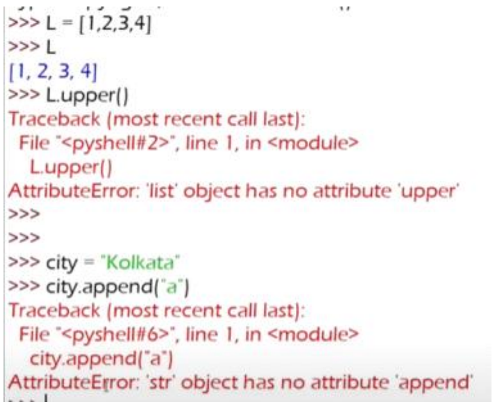
Importance Of OOP

01 March 2025 11:57



```
>>> L = [1,2,3,4]
>>> L
[1, 2, 3, 4]
>>> L.upper()
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    L.upper()
AttributeError: 'list' object has no attribute 'upper'
>>>
>>>
>>> city = "Kolkata"
>>> city.append("a")
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    city.append("a")
AttributeError: 'str' object has no attribute 'append'
```

Everything in Python is an Object ..!

OOP -> Object Oriented Programming

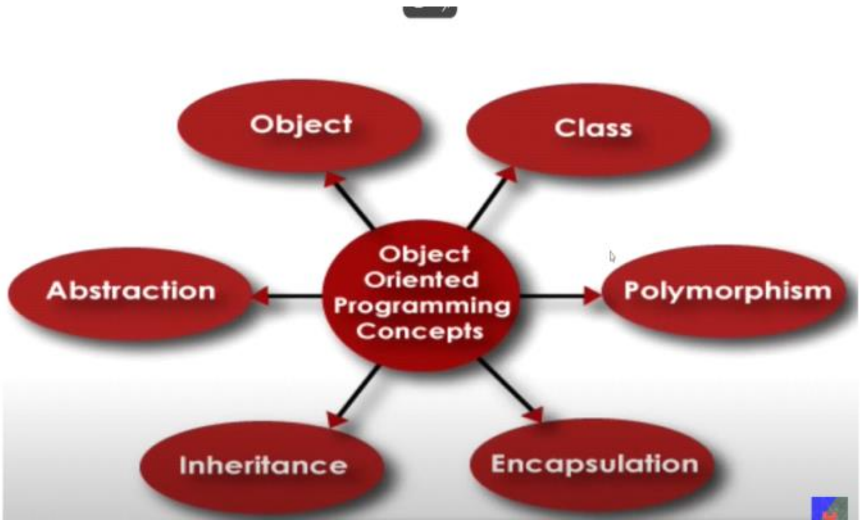
What is Object ?

What is attribute ?

The Problem:
Large applications develop => lengthy code => to add new feature => difficult to understand and alter the code -> because of clumsy and all functions in one place.

The Solution:
Generality to Specificality i.e. creating/using our own datatypes for better improvement when solving Real World Problems.

Properties of OOPS

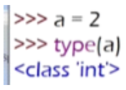


CLASS

01 March 2025 12:21

Class is a Blueprint. i.e. class defines the behavior of Object.

In python every datatype is a CLASS and variable is an OBJECT.

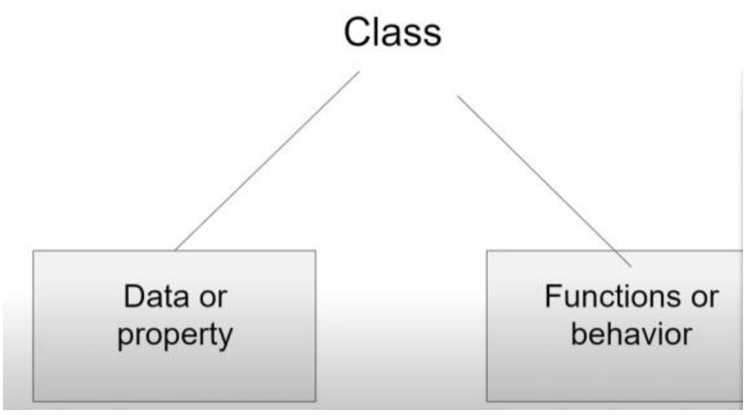


```
>>> a = 2
>>> type(a)
<class 'int'>
```

Class has only these 2 things --->

Data or Property:
The description of an object.

Functions or Behavior:
Behavior of an object



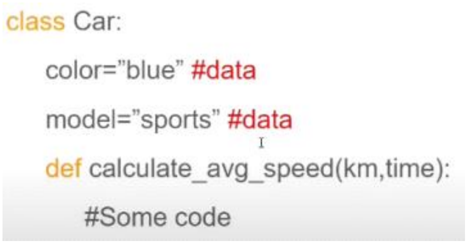
Class name should be written in **Pascal Case** Notation.

MyClass

Functions name should follow **snake case** notation.
my_function

myClass -> this is **camel case** notation

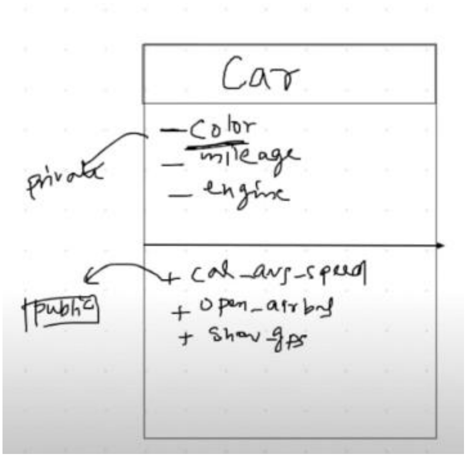
Below is basic structure of class



```
class Car:
    color="blue" #data
    model="sports" #data
    def calculate_avg_speed(km,time):
        #Some code
```

The diagrammatic representation of class would be as below:

- Sign -> indicates the data is private and can be accessed only inside class.
+ sign -> indicates the functions are open to access outside the class. i.e. Public

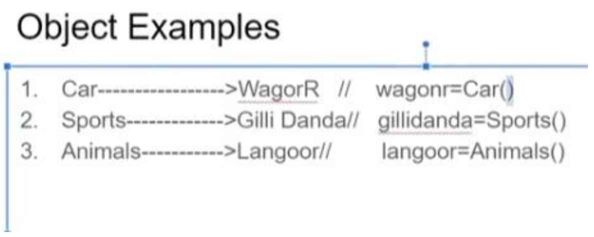


OBJECT

01 March 2025 14:36

It is the instance of a class.

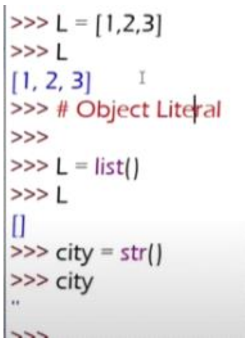
Syntax to create object: object_name = class_name()



Object Examples

1. Car------------------>WagorR // wagonr=Car()
2. Sports-------------->Gilli Danda// gillidanda=Sports()
3. Animals------------>Langoor// langoor=Animals()

For ease Python has provided **OBJECT LITERALS** for built in classes.



```
>>> L = [1,2,3]
>>> L
[1, 2, 3]
>>> # Object Literal
>>>
>>> L = list()
>>> L
[]
>>> city = str()
>>> city
''
>>>
```

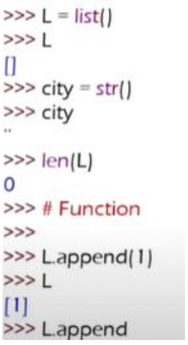
Practical Implementation Of Class & Object

01 March 2025 14:41

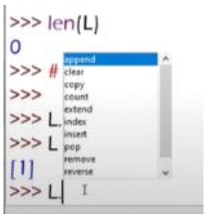
Functions (VS) Methods

Methods are nothing but functions (special functions) written inside a class.

There is a difference in calling these functions. Below len() is a normal function (so can be used by any object like string, array, list ..etc) but L.append() is a method. Append is a function written inside class List (so only objects of class list can access this function).



```
>>> L = list()
>>> L
[]
>>> city = str()
>>> city
''
>>> len(L)
0
>>> # Function
>>>
>>> L.append(1)
>>> L
[1]
>>> L.append
```

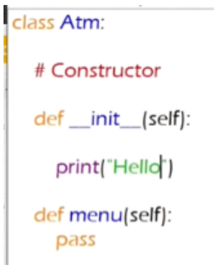


```
>>> len(L)
0
>>> #
>>>
>>> L.
>>> L
[1]
>>> L
```

Constructor

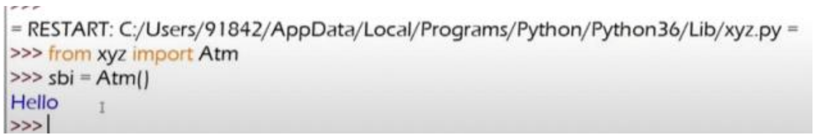
It is a special method which is called(code in this function is executed automatically) by default whenever the object of that particular class is created.

In other languages the name of constructor should be same as name of that class. In PYTHON the name of constructor will always be __init__ only.

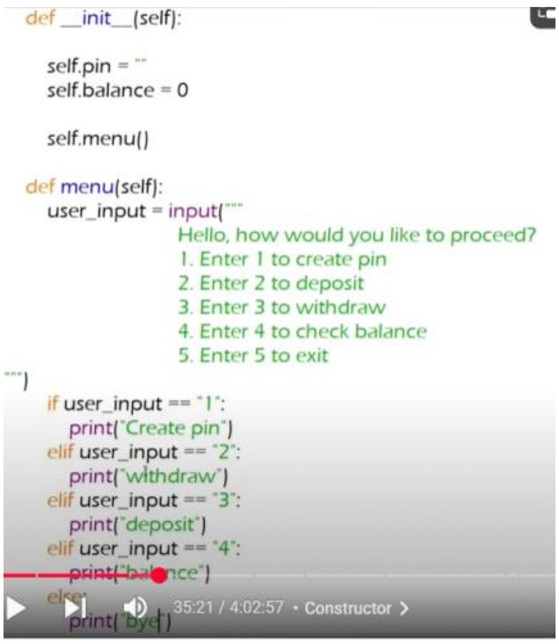


```
class Atm:

    # Constructor

    def __init__(self):

        print("Hello")

    def menu(self):
        pass
```

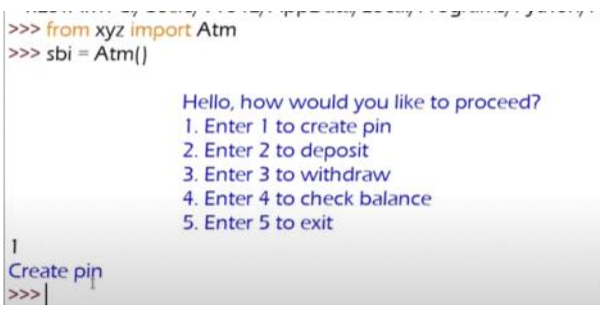
Create the object of that class
Syntax -> Obj_Name = Class_Name()



```
= RESTART: C:/Users/91842/AppData/Local/Programs/Python/Python36/Lib/xyz.py =
>>> from xyz import Atm
>>> sbi = Atm()
Hello
>>>
```

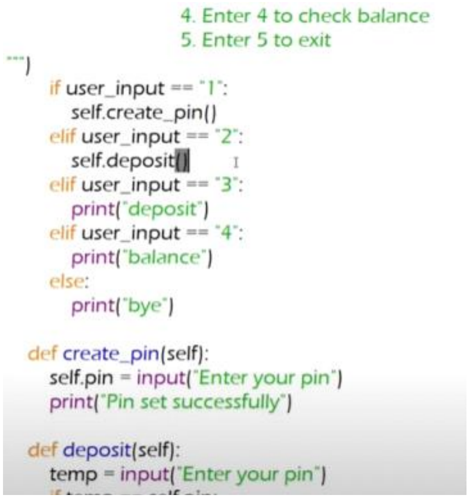


```
    def __init__(self):

        self.pin = ""
        self.balance = 0

        self.menu()

    def menu(self):
        user_input = input("""
                          Hello, how would you like to proceed?
                          1. Enter 1 to create pin
                          2. Enter 2 to deposit
                          3. Enter 3 to withdraw
                          4. Enter 4 to check balance
                          5. Enter 5 to exit
""")
        if user_input == "1":
            print("Create pin")
        elif user_input == "2":
            print("withdraw")
        elif user_input == "3":
            print("deposit")
        elif user_input == "4":
            print("balance")
        else:
            print("bye")
```

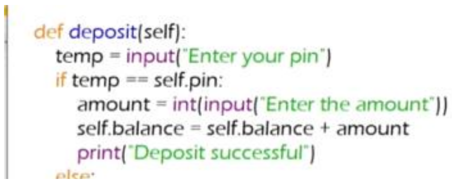


```
>>> from xyz import Atm
>>> sbi = Atm()

                          Hello, how would you like to proceed?
                          1. Enter 1 to create pin
                          2. Enter 2 to deposit
                          3. Enter 3 to withdraw
                          4. Enter 4 to check balance
                          5. Enter 5 to exit
1
Create pin
>>>
```

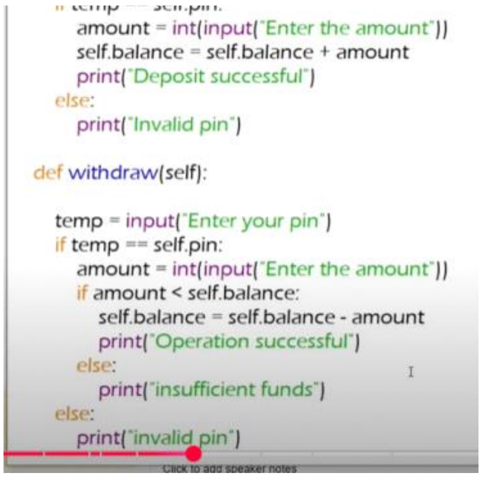
Basically we have created a navigation system here. Instead of that we will create methods for each menu there.



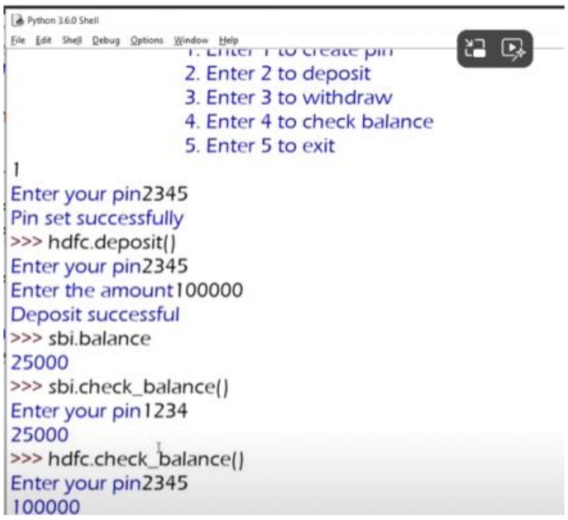
```
                          4. Enter 4 to check balance
                          5. Enter 5 to exit
""")
        if user_input == "1":
            self.create_pin()
        elif user_input == "2":
            self.deposit()
        elif user_input == "3":
            print("deposit")
        elif user_input == "4":
            print("balance")
        else:
            print("bye")

    def create_pin(self):
        self.pin = input("Enter your pin")
        print("Pin set successfully")

    def deposit(self):
        temp = input("Enter your pin")
```



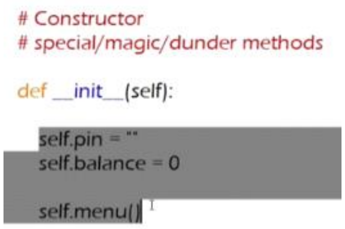
```
    def deposit(self):
        temp = input("Enter your pin")
        if temp == self.pin:
            amount = int(input("Enter the amount"))
            self.balance = self.balance + amount
            print("Deposit successful")
        else:
```



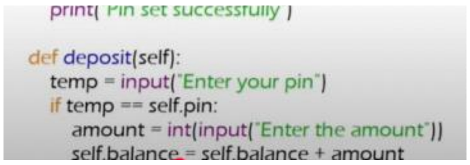
```
        if temp == self.pin:
            amount = int(input("Enter the amount"))
            self.balance = self.balance + amount
            print("Deposit successful")
        else:
            print("Invalid pin")

    def withdraw(self):

        temp = input("Enter your pin")
        if temp == self.pin:
            amount = int(input("Enter the amount"))
            if amount < self.balance:
                self.balance = self.balance - amount
                print("Operation successful")
            else:
                print("insufficient funds")
        else:
            print("invalid pin")
```



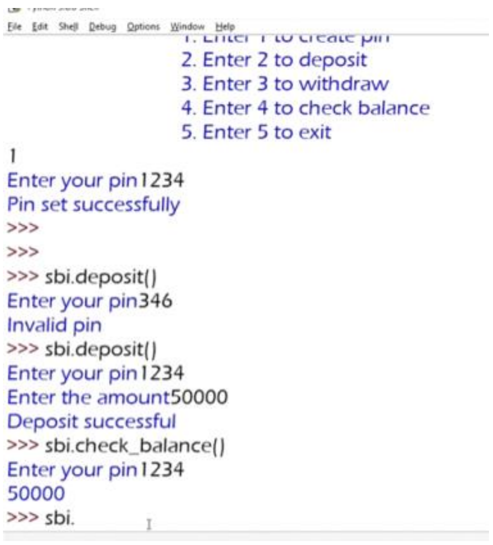
```
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
1
Enter your pin2345
Pin set successfully
>>> hdfc.deposit()
Enter your pin2345
Enter the amount100000
Deposit successful
>>> sbi.balance
25000
>>> sbi.check_balance()
Enter your pin1234
25000
>>> hdfc.check_balance()
Enter your pin2345
100000
```



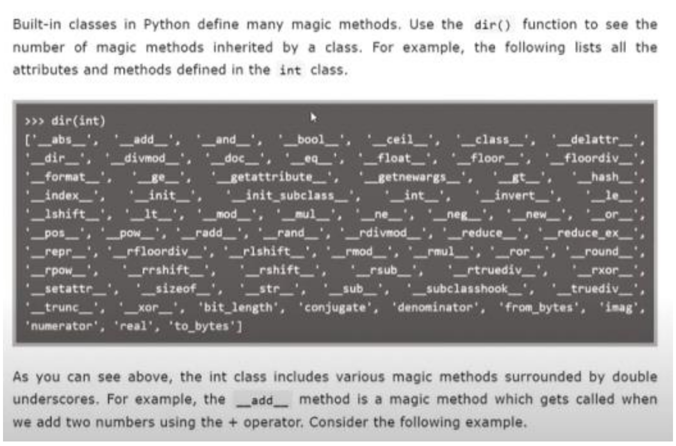
```
# Constructor
# special/magic/dunder methods

def __init__(self):

    self.pin = ""
    self.balance = 0

    self.menu()
```



```
        print("Pin set successfully")

    def deposit(self):
        temp = input("Enter your pin")
        if temp == self.pin:
            amount = int(input("Enter the amount"))
            self.balance = self.balance + amount
```



```
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
1
Enter your pin1234
Pin set successfully
>>>
>>>
>>> sbi.deposit()
Enter your pin346
Invalid pin
>>> sbi.deposit()
Enter your pin1234
Enter the amount50000
Deposit successful
>>> sbi.check_balance()
Enter your pin1234
50000
>>> sbi.
```



Built-in classes in Python define many magic methods. Use the `dir()` function to see the number of magic methods inherited by a class. For example, the following lists all the attributes and methods defined in the `int` class.

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
'__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

As you can see above, the int class includes various magic methods surrounded by double underscores. For example, the `__add__` method is a magic method which gets called when we add two numbers using the + operator. Consider the following example.

Objects cannot call the constructor directly and also there are some magic methods in python which cannot be called explicitly using objects.

When to use CONSTRUCTOR ?

Constructor is also a special/magic/dunder method which cannot be called using objects but can be called whenever object is created. That means there is no privilege given(cannot be given) to user to invoke constructor. So write the functionality in constructor that requires to execute without the intervention of user.

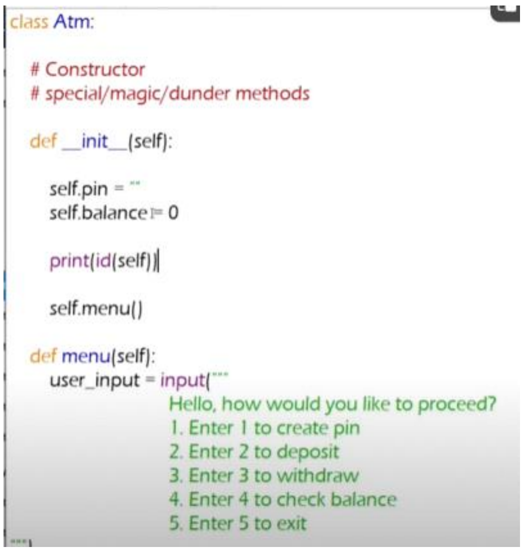
Eg: Food Delivery app -> as soon as the app is launched it will connect to internet/DB/Hardware by default. Rather than asking user if app can connect to internet ??

Self in Python

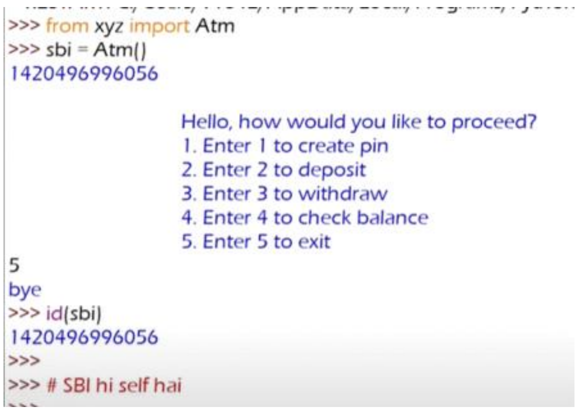
16 March 2025 17:55

Self is nothing but the object that we are currently dealing with.

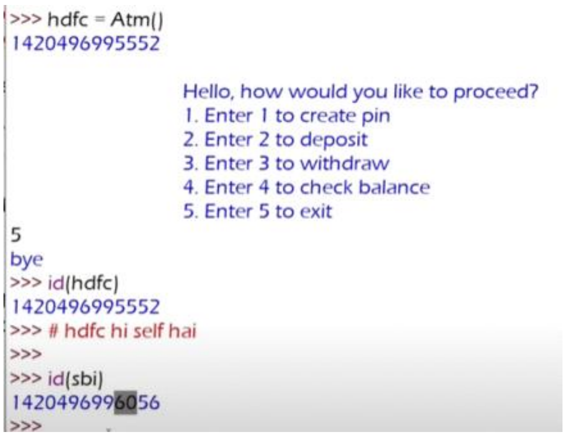
see that the memory location/address of self is same as object memory location/address.



```
class Atm:

  # Constructor
  # special/magic/dunder methods

  def __init__(self):

    self.pin = ""
    self.balance = 0

    print(id(self))

    self.menu()

  def menu(self):
    user_input = input("""
                    Hello, how would you like to proceed?
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
""")
```

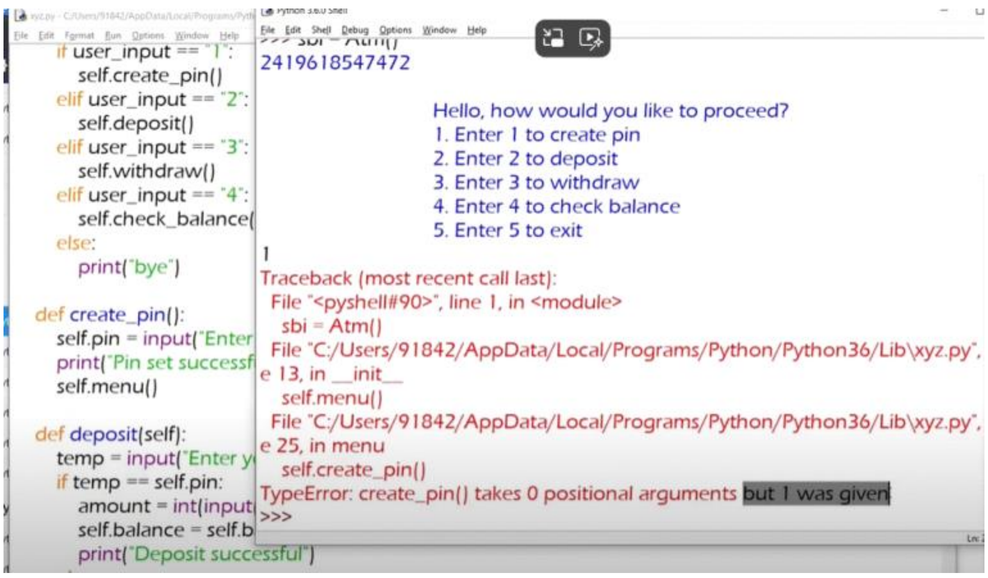


```
>>> from xyz import Atm
>>> sbi = Atm()
1420496996056

                    Hello, how would you like to proceed?
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
5
bye
>>> id(sbi)
1420496996056
>>>
>>> # SBI hi self hai
```



```
>>> hdfc = Atm()
1420496995552

                    Hello, how would you like to proceed?
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
5
bye
>>> id(hdfc)
1420496995552
>>> # hdfc hi self hai
>>>
>>> id(sbi)
1420496996056
>>>
```

What happens if self is removed from the method arguments. It will not work, app crashes. See below.



```
    if user_input == "1":
      self.create_pin()
    elif user_input == "2":
      self.deposit()
    elif user_input == "3":
      self.withdraw()
    elif user_input == "4":
      self.check_balance(
    else:
      print("bye")

  def create_pin():
    self.pin = input("Enter
    print("Pin set successf
    self.menu()

  def deposit(self):
    temp = input("Enter y
    if temp == self.pin:
      amount = int(input
      self.balance = self.b
      print("Deposit successful")
```

```
2419618547472

                    Hello, how would you like to proceed?
                    1. Enter 1 to create pin
                    2. Enter 2 to deposit
                    3. Enter 3 to withdraw
                    4. Enter 4 to check balance
                    5. Enter 5 to exit
1
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    sbi = Atm()
  File "C:/Users/91842/AppData/Local/Programs/Python/Python36/Lib\xyz.py",
e 13, in __init__
    self.menu()
  File "C:/Users/91842/AppData/Local/Programs/Python/Python36/Lib\xyz.py",
e 25, in menu
    self.create_pin()
TypeError: create_pin() takes 0 positional arguments but 1 was given
>>>
```

Why to use self in methods as argument ?

Rule of OOP: 1 method in any class do not have power to access any other method/data members in the same class.

Only object of that class has the power to access all the methods and data members of that class.

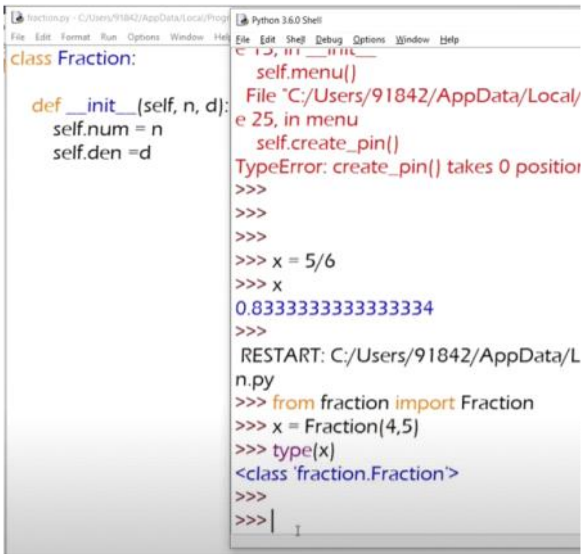
In python sbi.create_pin() => 1 argument i.e. sbi itself is being passed to create_pin() method by default. (refer error beside)

Since only way to access them is via Object i.e. via SELF.
=> Self is coming as an current object and hence accessing the data members and methods inside class.

Creating own data type

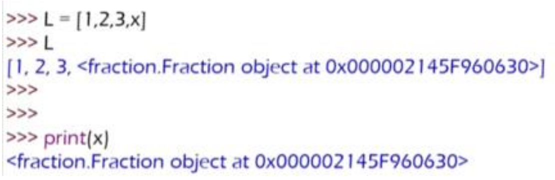
16 March 2025 18:16

Let's create Fraction since python(most programming languages) doesn't handle fraction datatype.



```
class Fraction:

    def __init__(self, n, d):
        self.num = n
        self.den =d
```

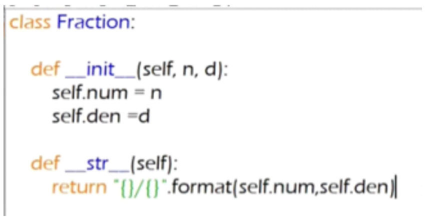
```
e 15, in __init__
    self.menu()
  File "C:/Users/91842/AppData/Local/
e 25, in menu
    self.create_pin()
TypeError: create_pin() takes 0 positio
>>>
>>>
>>>
>>> x = 5/6
>>> x
0.8333333333333334
>>>
 RESTART: C:/Users/91842/AppData/L
n.py
>>> from fraction import Fraction
>>> x = Fraction(4,5)
>>> type(x)
<class 'fraction.Fraction'>
>>>
>>>|
```



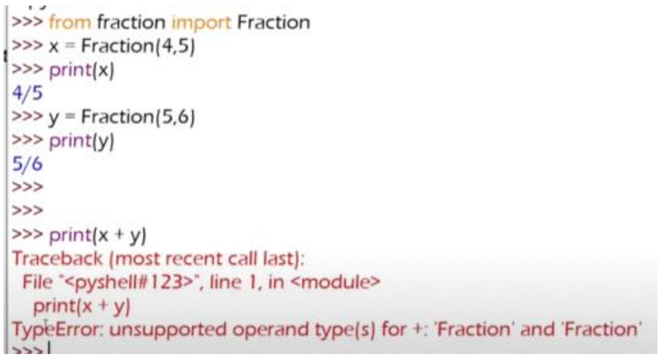
```
>>> L = [1,2,3,x]
>>> L
[1, 2, 3, <fraction.Fraction object at 0x000002145F960630>]
>>>
>>>
>>> print(x)
<fraction.Fraction object at 0x000002145F960630>
```

We didn't tell the python how this datatype fraction look like. Whenever print is written it will by default execute the magic method named __str__ inside that particular class where object is created.

So we will implement __str__ magic method accordingly and see.



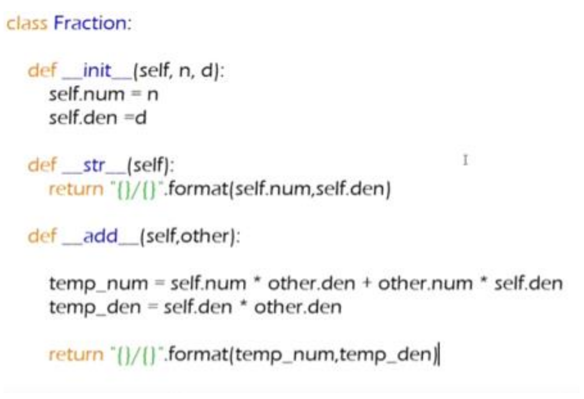
```
class Fraction:

    def __init__(self, n, d):
        self.num = n
        self.den =d

    def __str__(self):
        return "{}/{}".format(self.num,self.den)
```



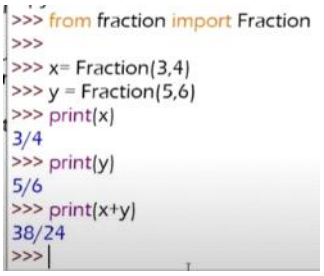
```
>>> from fraction import Fraction
>>> x = Fraction(4,5)
>>> print(x)
4/5
>>> y = Fraction(5,6)
>>> print(y)
5/6
>>>
>>>
>>> print(x + y)
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    print(x + y)
TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
>>>|
```

Here to the left we can see that add operation is not defined since we haven't mentioned the same in our class.

When + is operand in python, by default another magic method __add__ is called. So implement this to perform addition.

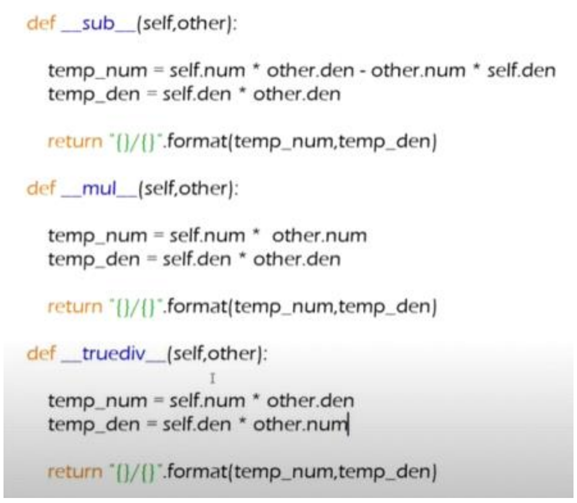


```
class Fraction:

    def __init__(self, n, d):
        self.num = n
        self.den =d

    def __str__(self):
        return "{}/{}".format(self.num,self.den)

    def __add__(self,other):

        temp_num = self.num * other.den + other.num * self.den
        temp_den = self.den * other.den

        return "{}/{}".format(temp_num,temp_den)
```

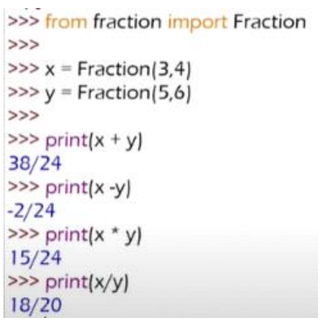


```
>>> from fraction import Fraction
>>>
>>> x= Fraction(3,4)
>>> y = Fraction(5,6)
>>> print(x)
3/4
>>> print(y)
5/6
>>> print(x+y)
38/24
>>>|
```

Similarly for subtraction, multiplication and division the magic methods can be implemented as below.



```
def __sub__(self,other):

    temp_num = self.num * other.den - other.num * self.den
    temp_den = self.den * other.den

    return "{}/{}".format(temp_num,temp_den)

def __mul__(self,other):

    temp_num = self.num *  other.num
    temp_den = self.den * other.den

    return "{}/{}".format(temp_num,temp_den)

def __truediv__(self,other):

    temp_num = self.num * other.den
    temp_den = self.den * other.num

    return "{}/{}".format(temp_num,temp_den)
```



```
>>> from fraction import Fraction
>>>
>>> x = Fraction(3,4)
>>> y = Fraction(5,6)
>>>
>>> print(x + y)
38/24
>>> print(x -y)
-2/24
>>> print(x * y)
15/24
>>> print(x/y)
18/20
```

Note: We can create any kind of custom datatype with all the necessary operations possible using magic methods and can be use d by anyone just by having them in their Lib folder.

Eg: datatype to deal with coordinate geometry, Matrices .. Etc (depending on our requirements).

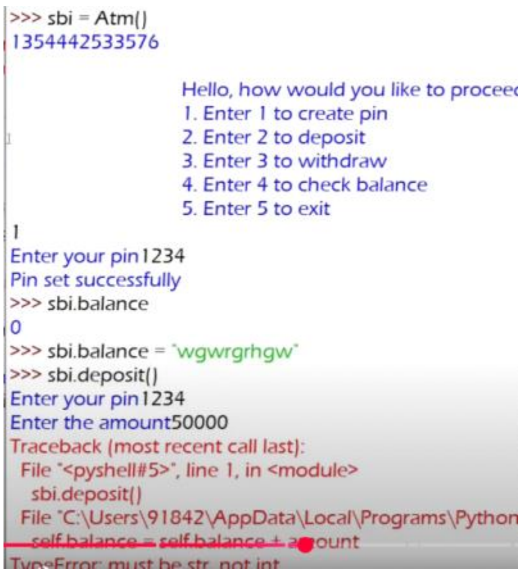
Encapsulation

17 March 2025 17:28

Instance Variable: The variables that are defined inside the constructor and their values varies(should vary) from object to object.

The problem is since the objects can access and modify the instance variables the app may crash and it's not desired (meaningless)

eg: customer changing his available balance in his account. **Solution:** hide the data members using access modifiers so that they can be accessible only inside the class.

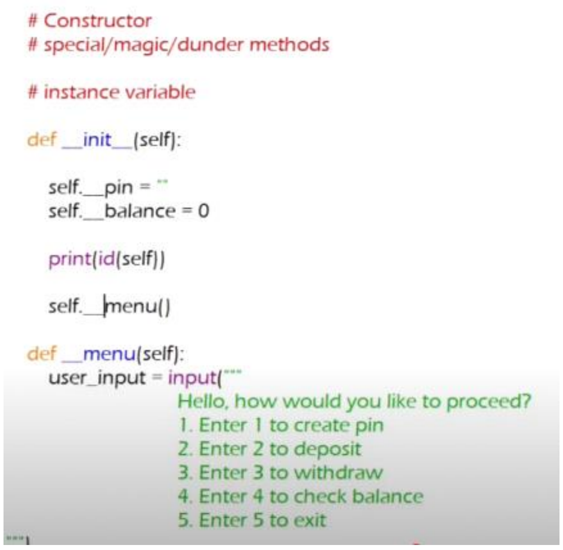


```
>>> sbi = Atm()
1354442533576

                     Hello, how would you like to procee
                     1. Enter 1 to create pin
                     2. Enter 2 to deposit
                     3. Enter 3 to withdraw
                     4. Enter 4 to check balance
                     5. Enter 5 to exit
1
Enter your pin1234
Pin set successfully
>>> sbi.balance
0
>>> sbi.balance = "wgwrgrhgw"
>>> sbi.deposit()
Enter your pin1234
Enter the amount50000
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    sbi.deposit()
  File "C:\Users\91842\AppData\Local\Programs\Python
    self.balance = self.balance + amount
TypeError: must be str, not int
```

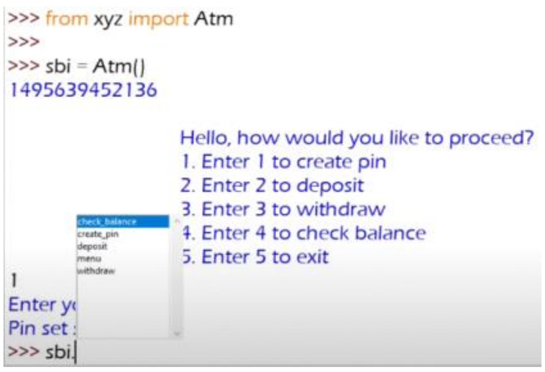
The process of hiding the data members in a class and restricting the object to change its value.
eg: private keyword in other languages (private => only accessible within the class)

In Python it can be done by placing __ (double _) before variable in all places where that variable is used.

Using __ we can hide the methods also i.e. restricting the object to access that method.



```
# Constructor
# special/magic/dunder methods

# instance variable

def __init__(self):

    self.__pin = ""
    self.__balance = 0

    print(id(self))

    self.__menu()

def __menu(self):
    user_input = input("""
            Hello, how would you like to proceed?
            1. Enter 1 to create pin
            2. Enter 2 to deposit
            3. Enter 3 to withdraw
            4. Enter 4 to check balance
            5. Enter 5 to exit
""")
```



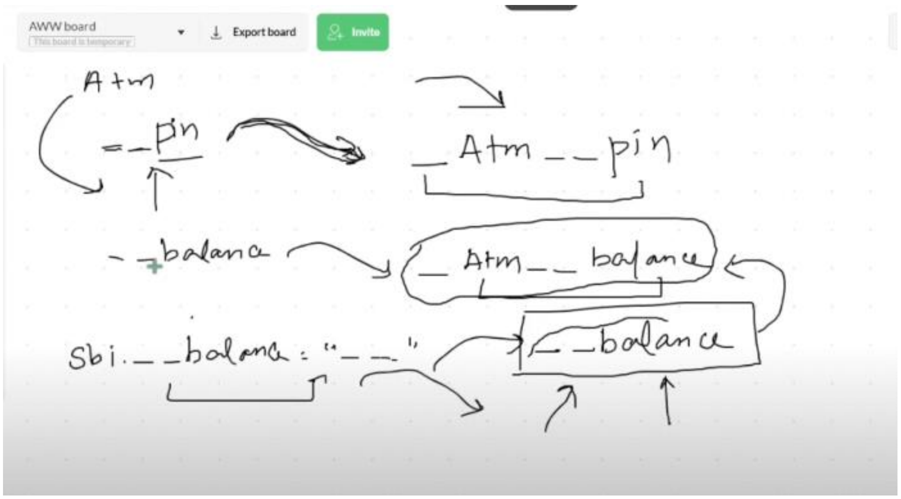
```
>>> from xyz import Atm
>>>
>>> sbi = Atm()
1495639452136

                     Hello, how would you like to proceed?
                     1. Enter 1 to create pin
                     2. Enter 2 to deposit
                     3. Enter 3 to withdraw
                     4. Enter 4 to check balance
                     5. Enter 5 to exit
1
Enter yo
Pin set
>>> sbi.
```

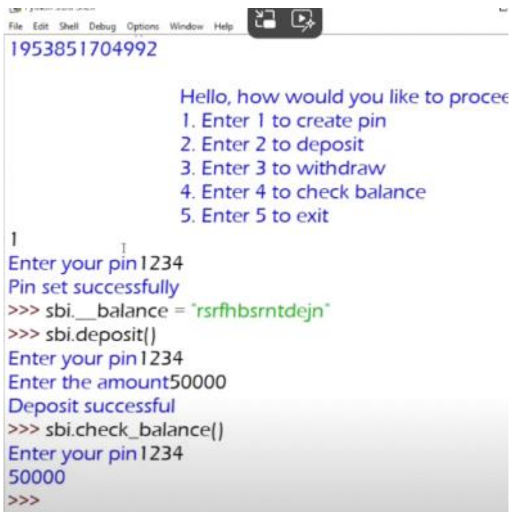
Note: Internally if we apply __pin in the class atm => python interpreter will convert this __pin -> _atm__pin

can we modify the values of data members using object.__data member ?

Though it will not throw the runtime error, The value of data member is not actually modified. Why ? -> above note

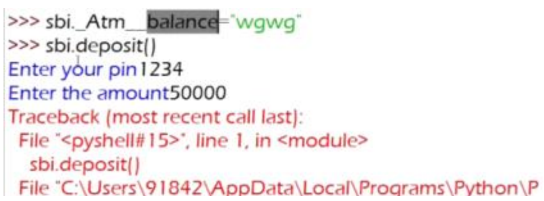
Actually a new variable/data member with name __balance is created in that class but that variable is not used anywhere. Why ? -> above note.





```
1953851704992

                     Hello, how would you like to procee
                     1. Enter 1 to create pin
                     2. Enter 2 to deposit
                     3. Enter 3 to withdraw
                     4. Enter 4 to check balance
                     5. Enter 5 to exit
1
Enter your pin1234
Pin set successfully
>>> sbi.__balance = "rsrfhbsrntdejn"
>>> sbi.deposit()
Enter your pin1234
Enter the amount50000
Deposit successful
>>> sbi.check_balance()
Enter your pin1234
50000
>>>
```

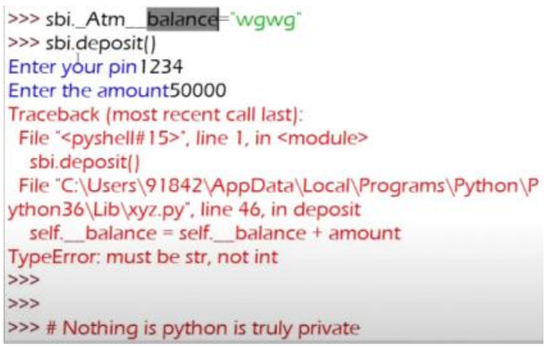
Observation: **Nothing in Python is truly Private.** Eg: if we know the class name and data member name, then we can modify the value of that variable though it is hidden.



```
>>> sbi._Atm__balance="wgwg"
>>> sbi.deposit()
Enter your pin1234
Enter the amount50000
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    sbi.deposit()
  File "C:\Users\91842\AppData\Local\Programs\Python\P
```

But why did python given this privilege of making nothing truly private: Python is for adults, kind of gentlemen's agreement. If it is said not to access you should not but still if you want to really do u can access. Eg: if senior programmer hides, junior programmer can access but they are working in same project only not enemies.

A kind of read only / write only stuff in oops is getter and setter method.

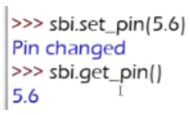


```
>>> sbi._Atm__balance="wgwg"
>>> sbi.deposit()
Enter your pin1234
Enter the amount50000
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    sbi.deposit()
  File "C:\Users\91842\AppData\Local\Programs\Python\P
ython36\Lib\xyz.py", line 46, in deposit
    self.__balance = self.__balance + amount
TypeError: must be str, not int
>>>
>>>
>>> # Nothing is python is truly private
```

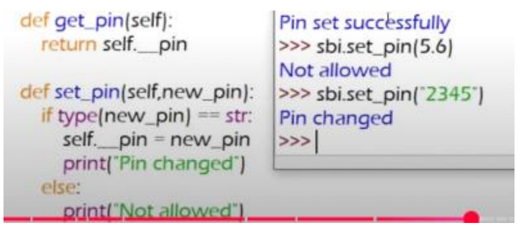
We have hidden the data members but created get set method to allow user to modify them -> why ?

Because we are getting the input from set method => we can put validations on the incoming value/data. i.e. we have control over this operation.

In earlier case directly accessing the data members and modifying their values => no control / scope for validation.



```
>>> sbi.set_pin(5.6)
Pin changed
>>> sbi.get_pin()
5.6
```

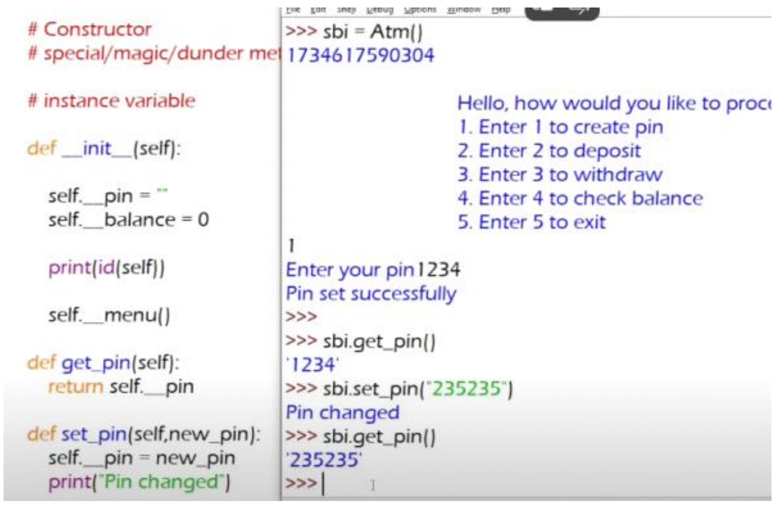


```
def get_pin(self):
    return self.__pin

def set_pin(self,new_pin):
    if type(new_pin) == str:
        self.__pin = new_pin
        print("Pin changed")
    else:
        print("Not allowed")
```

```
Pin set successfully
>>> sbi.set_pin(5.6)
Not allowed
>>> sbi.set_pin("2345")
Pin changed
>>>
```

But why did python given this privilege of making nothing truly private? Python is for adults, kind of gentlemen's agreement. If it is said not to access you should not but still if you want to really do u can access. Eg: if senior programmer hides, junior programmer can access but they are working in same project only not enemies.

A kind of read only / write only stuff in oops is getter and setter method.



```
# Constructor
# special/magic/dunder me

# instance variable

def __init__(self):

    self.__pin = ""
    self.__balance = 0

    print(id(self))

    self.__menu()

def get_pin(self):
    return self.__pin

def set_pin(self,new_pin):
    self.__pin = new_pin
    print("Pin changed")
```

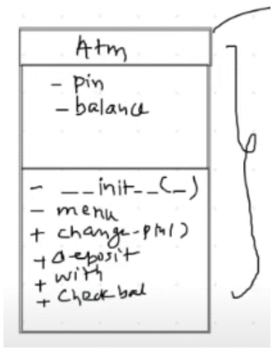
```
>>> sbi = Atm()
1734617590304

                Hello, how would you like to proc
                1. Enter 1 to create pin
                2. Enter 2 to deposit
                3. Enter 3 to withdraw
                4. Enter 4 to check balance
                5. Enter 5 to exit
1
Enter your pin1234
Pin set successfully
>>>
>>> sbi.get_pin()
'1234'
>>> sbi.set_pin("235235")
Pin changed
>>> sbi.get_pin()
'235235'
>>>
```

Encapsulation: The process of combining the two things in a class

1. Hiding the data members
2. Providing the access to read and write them with 2 functions get & set.

Why encapsulation ? To make your data members protected from accessing and modifying at the wish of user without our control.

How? Using private attributes (__) and getter/setter methods



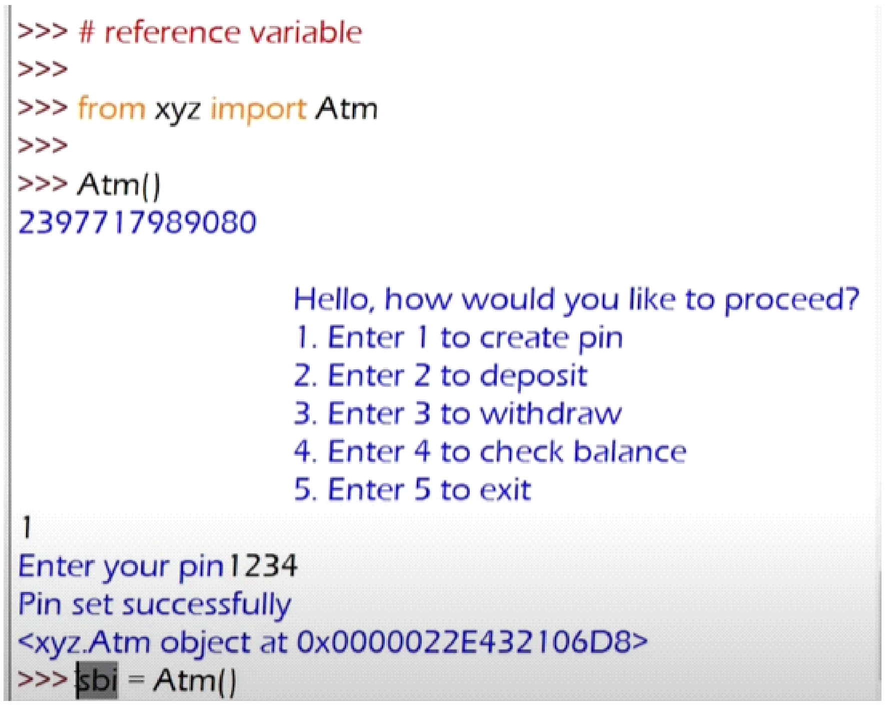
Beside is the class diagram. - => private, + => public

It can be useful while developing the application or for understanding the next steps on how to proceed with development based on the requirements gathered from client.

Reference Variable

21 March 2025 07:20

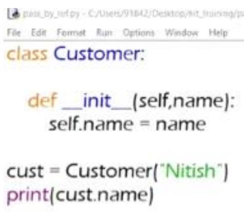
Using atm() the memory is lost at that address because no variable is assigned to that particular object and so after the execution, we cannot access that memory. So we write sbi = atm() where sbi is called as **reference variable.**



```
>>> # reference variable
>>>
>>> from xyz import Atm
>>>
>>> Atm()
2397717989080

                        Hello, how would you like to proceed?
                        1. Enter 1 to create pin
                        2. Enter 2 to deposit
                        3. Enter 3 to withdraw
                        4. Enter 4 to check balance
                        5. Enter 5 to exit
1
Enter your pin1234
Pin set successfully
<xyz.Atm object at 0x0000022E432106D8>
>>> sbi = Atm()
```

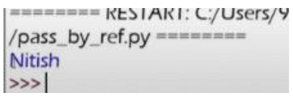
Pass By Reference

23 March 2025 11:07



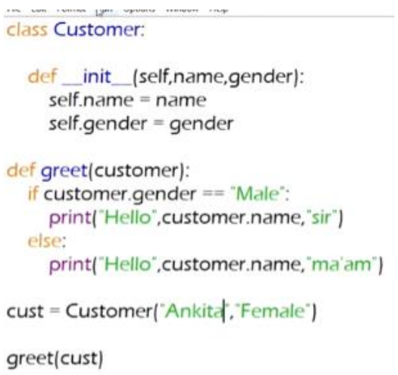
```
class Customer:

    def __init__(self,name):
        self.name = name

cust = Customer("Nitish")
print(cust.name)
```

O/P of beside code.

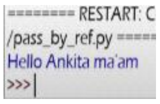


```
======== RESTART: C:/Users/9
/pass_by_ref.py ========
Nitish
>>>|
```

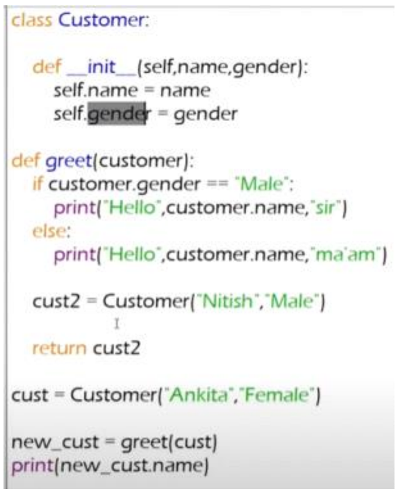
Passing the object of a class as an argument to a function written outside the class.



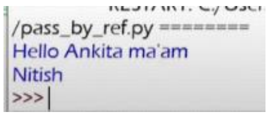
```
class Customer:

    def __init__(self,name,gender):
        self.name = name
        self.gender = gender

def greet(customer):
    if customer.gender == "Male":
        print("Hello",customer.name,"sir")
    else:
        print("Hello",customer.name,"ma'am")

cust = Customer("Ankita","Female")

greet(cust)
```



```
======== RESTART: C
/pass_by_ref.py =====
Hello Ankita ma'am
>>>|
```

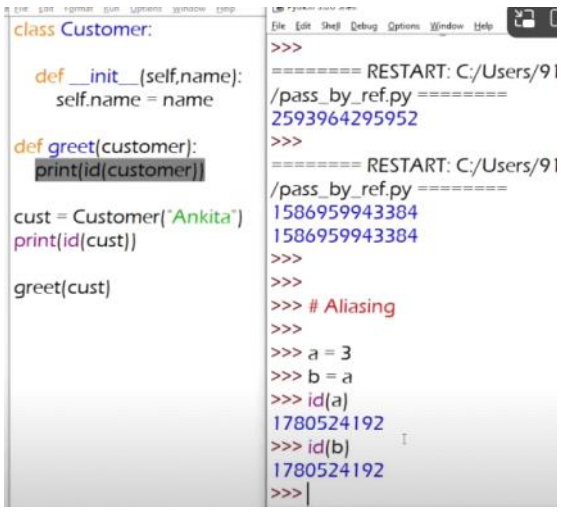


```
class Customer:

    def __init__(self,name,gender):
        self.name = name
        self.gender = gender

def greet(customer):
    if customer.gender == "Male":
        print("Hello",customer.name,"sir")
    else:
        print("Hello",customer.name,"ma'am")

    cust2 = Customer("Nitish","Male")

    return cust2

cust = Customer("Ankita","Female")

new_cust = greet(cust)
print(new_cust.name)
```



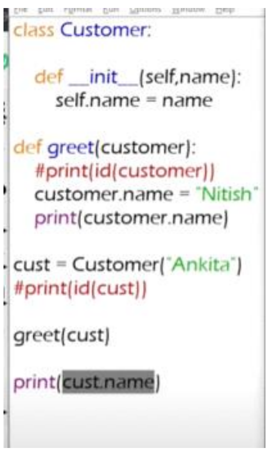
```
/pass_by_ref.py ========
Hello Ankita ma'am
Nitish
>>>|
```

Pass by reference is nothing but a kind of aliasing i.e. using the same memory address of the already created variable/object for further use. Refer below.

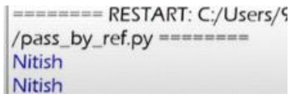


```
class Customer:

    def __init__(self,name):
        self.name = name

def greet(customer):
    print(id(customer))

cust = Customer("Ankita")
print(id(cust))

greet(cust)
```

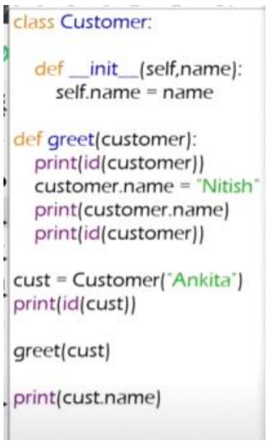
```
>>>
======== RESTART: C:/Users/91
/pass_by_ref.py ========
2593964295952
>>>
======== RESTART: C:/Users/91
/pass_by_ref.py ========
1586959943384
1586959943384
>>>
>>>
>>> # Aliasing
>>>
>>> a = 3
>>> b = a
>>> id(a)
1780524192
>>> id(b)
1780524192
>>>|
```



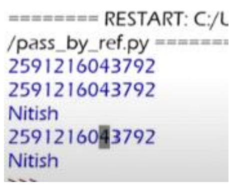
```
class Customer:

    def __init__(self,name):
        self.name = name

def greet(customer):
    #print(id(customer))
    customer.name = "Nitish"
    print(customer.name)

cust = Customer("Ankita")
#print(id(cust))

greet(cust)

print(cust.name)
```



```
======== RESTART: C:/Users/9
/pass_by_ref.py ========
Nitish
Nitish
```

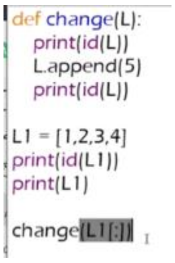


```
class Customer:

    def __init__(self,name):
        self.name = name

def greet(customer):
    print(id(customer))
    customer.name = "Nitish"
    print(customer.name)
    print(id(customer))

cust = Customer("Ankita")
print(id(cust))

greet(cust)

print(cust.name)
```

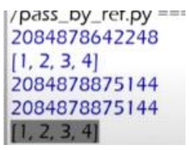


```
======== RESTART: C:/L
/pass_by_ref.py ======
2591216043792
2591216043792
Nitish
2591216043792
Nitish
```

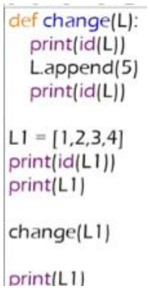
Observation: Objects of the class are mutable datatypes like LIST, DICTIONARY, SET.



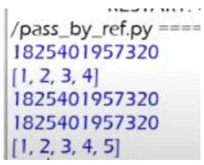
```
def change(L):
    print(id(L))
    L.append(5)
    print(id(L))

L1 = [1,2,3,4]
print(id(L1))
print(L1)

change(L1[:])
```



```
/pass_by_ref.py ===
2084878642248
[1, 2, 3, 4]
2084878875144
2084878875144
[1, 2, 3, 4]
```



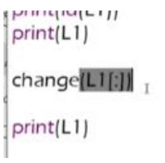
```
def change(L):
    print(id(L))
    L.append(5)
    print(id(L))

L1 = [1,2,3,4]
print(id(L1))
print(L1)

change(L1)

print(L1)
```



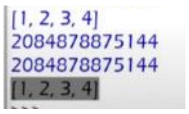
```
/pass_by_ref.py ====
1825401957320
[1, 2, 3, 4]
1825401957320
1825401957320
[1, 2, 3, 4, 5]
```

Here by directly passing the outside list to the function, with the operations inside a function, there will be permanent changes taking place in outside list.

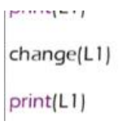
How to avoid ? => use **CLONING.** **(**refer left side**)**



```
print(id(L1))
print(L1)

change(L1[:])

print(L1)
```



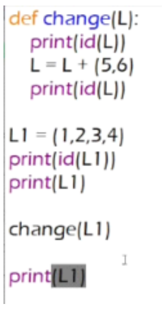
```
[1, 2, 3, 4]
2084878875144
2084878875144
[1, 2, 3, 4]
```



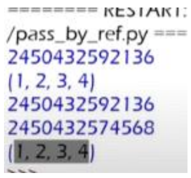
```
change(L1)

print(L1)
```

Here by directly passing the outside list to the function, with the operations inside a function, there will be permanent changes taking place in outside list.

How to avoid ? => use **CLONING. (**refer left side**)**



```
def change(L):
    print(id(L))
    L = L + (5,6)
    print(id(L))

L1 = (1,2,3,4)
print(id(L1))
print(L1)

change(L1)

print(L1)
```



```
======== RESTART:
/pass_by_ref.py ===
2450432592136
(1, 2, 3, 4)
2450432592136
2450432574568
(1, 2, 3, 4)
```

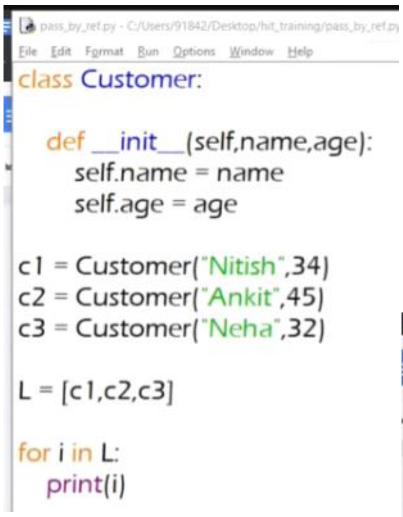
Key Points:
When passing the objects/data types to a function by reference, if they are mutable datatypes then the changes will be permanent in the original object. Else if it is immutable datatype then there will not be changes in the original object and for any operation new object will be created at new memory location.

(refer above and beside).

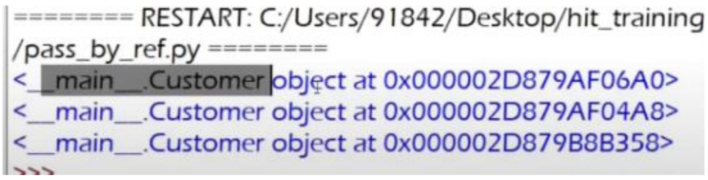
Collection of Objects

23 March 2025 17:03

The same with objects will apply like with datatypes in python. The loops, print .. Etc

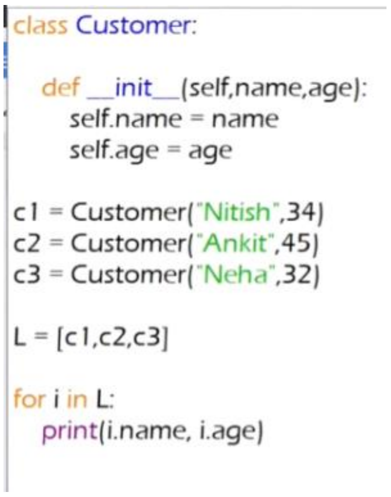


```
pass_by_ref.py - C:/Users/91842/Desktop/hit_training/pass_by_ref.py
File Edit Format Run Options Window Help
class Customer:

    def __init__(self,name,age):
        self.name = name
        self.age = age

c1 = Customer("Nitish",34)
c2 = Customer("Ankit",45)
c3 = Customer("Neha",32)

L = [c1,c2,c3]

for i in L:
    print(i)
```

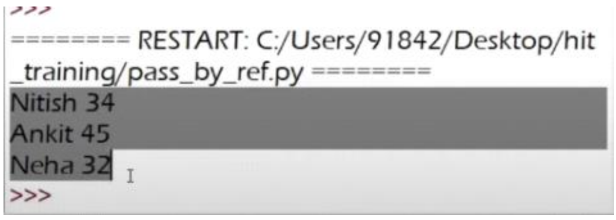


```
======== RESTART: C:/Users/91842/Desktop/hit_training
/pass_by_ref.py ========
<__main__.Customer object at 0x000002D879AF06A0>
<__main__.Customer object at 0x000002D879AF04A8>
<__main__.Customer object at 0x000002D879B8B358>
>>>
```

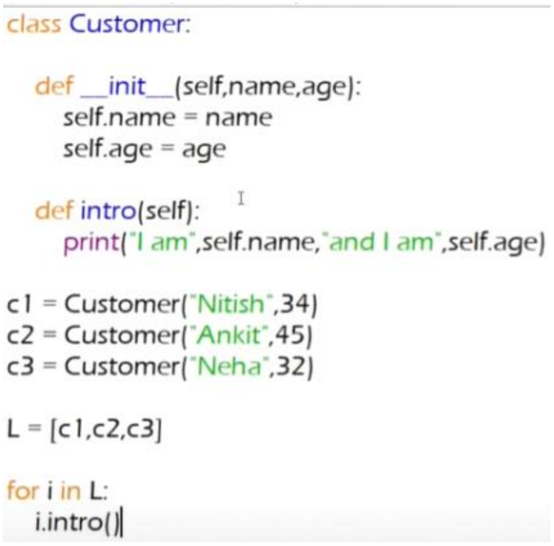
We didn't write method for str and int, so the memory location of the 3 objects has been printed. Rather try print with data members.



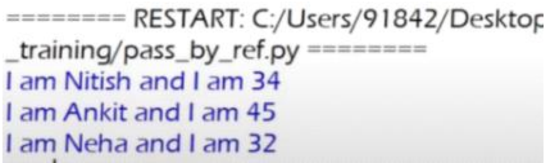
```
class Customer:

    def __init__(self,name,age):
        self.name = name
        self.age = age

c1 = Customer("Nitish",34)
c2 = Customer("Ankit",45)
c3 = Customer("Neha",32)

L = [c1,c2,c3]

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



```
>>>
======== RESTART: C:/Users/91842/Desktop/hit
_training/pass_by_ref.py ========
Nitish 34
Ankit 45
Neha 32
>>>
```



```
class Customer:

    def __init__(self,name,age):
        self.name = name
        self.age = age

    def intro(self):
        print("I am",self.name,"and I am",self.age)

c1 = Customer("Nitish",34)
c2 = Customer("Ankit",45)
c3 = Customer("Neha",32)

L = [c1,c2,c3]

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

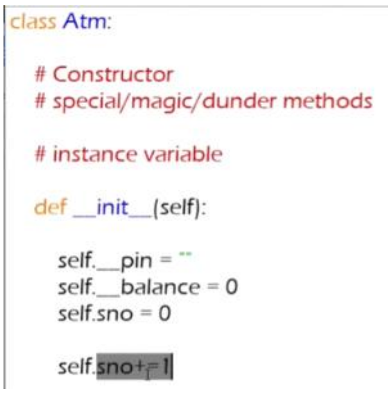


```
======== RESTART: C:/Users/91842/Desktop
_training/pass_by_ref.py ========
I am Nitish and I am 34
I am Ankit and I am 45
I am Neha and I am 32
```

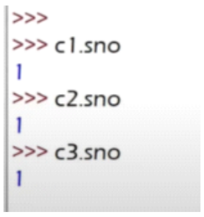
Static Variables & Methods

23 March 2025 17:28

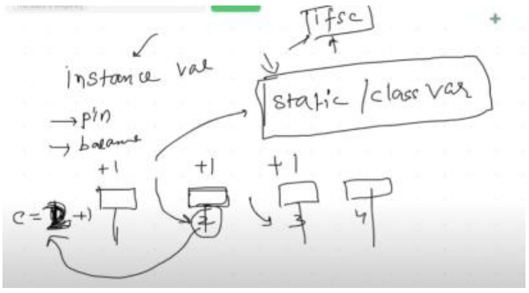
Example scenario: maintain a unique serial num for every object i.e. for 1st obj s.no = 1 and increment on creating more objects further.



```
class Atm:

    # Constructor
    # special/magic/dunder methods

    # instance variable

    def __init__(self):

        self.__pin = ""
        self.__balance = 0
        self.sno = 0

        self.sno+=1
```



```
>>>
>>> c1.sno
1
>>> c2.sno
1
>>> c3.sno
1
```

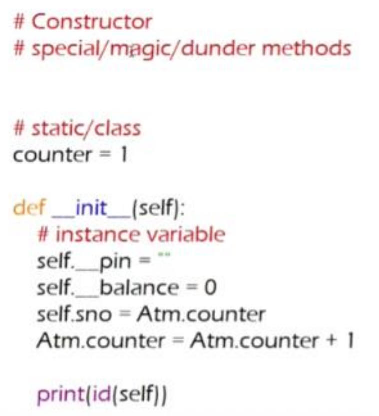


Here each time obj is created, sno is set to zero then incremented by 1 => sno = 1 for every obj (customer).

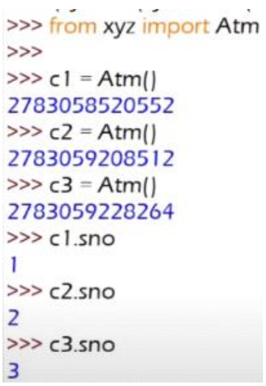
Solution -> **Static / Class Variable:** it's a variable in python whose value will be same for every object unlike instance variable.

Eg: pin & balance -> instance variable | IFSC -> static variable.

Note: In a class, instance variables are always written inside the constructor. Static variables are written outside the constructor. To access static variable => class_name.static_variable_name

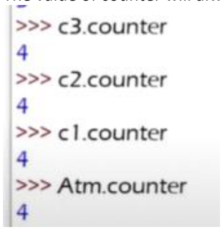


```
# Constructor
# special/magic/dunder methods


# static/class
counter = 1

def __init__(self):
    # instance variable
    self.__pin = ""
    self.__balance = 0
    self.sno = Atm.counter
    Atm.counter = Atm.counter + 1

    print(id(self))
```

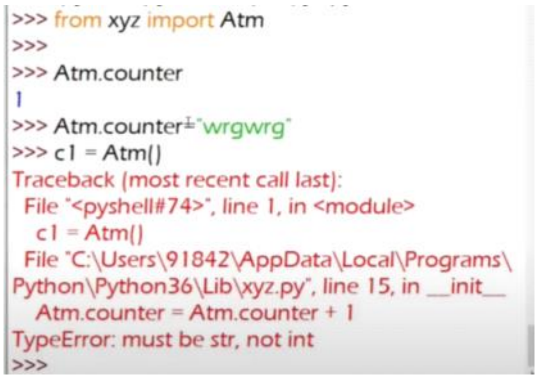


```
>>> from xyz import Atm
>>>
>>> c1 = Atm()
2783058520552
>>> c2 = Atm()
2783059208512
>>> c3 = Atm()
2783059228264
>>> c1.sno
1
>>> c2.sno
2
>>> c3.sno
3
```

The value of counter will always be the latest incremented value.

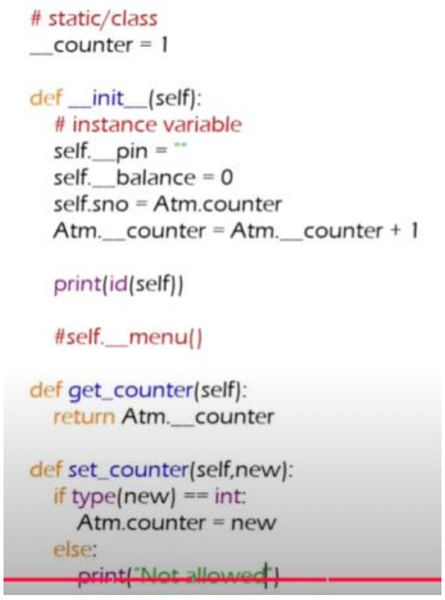


```
>>> c3.counter
4
>>> c2.counter
4
>>> c1.counter
4
>>> Atm.counter
4
```

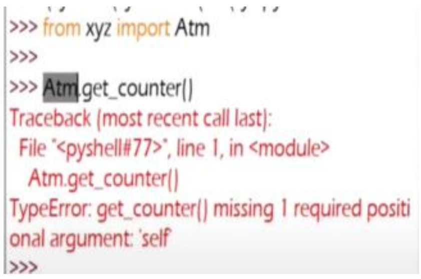


```
>>> from xyz import Atm
>>>
>>> Atm.counter
1
>>> Atm.counter="wrgwrg"
>>> c1 = Atm()
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    c1 = Atm()
  File "C:\Users\91842\AppData\Local\Programs\Python\Python36\Lib\xyz.py", line 15, in __init__
    Atm.counter = Atm.counter + 1
TypeError: must be str, not int
>>>
```

The code will again crash if we try to assign new value to the static variable. Solution ? => make the static variable as private.



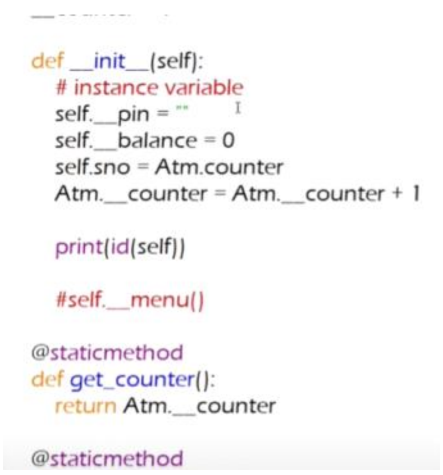
```
# static/class
__counter = 1

def __init__(self):
    # instance variable
    self.__pin = ""
    self.__balance = 0
    self.sno = Atm.counter
    Atm.__counter = Atm.__counter + 1

    print(id(self))

    #self.__menu()

def get_counter(self):
    return Atm.__counter

def set_counter(self,new):
    if type(new) == int:
        Atm.counter = new
    else:
        print('Not allowed')
```



```
>>> from xyz import Atm
>>>
>>> Atm.get_counter()
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    Atm.get_counter()
TypeError: get_counter() missing 1 required positional argument: 'self'
>>>
```

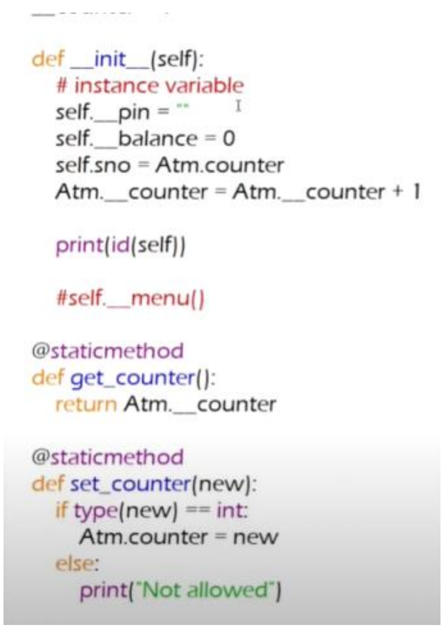
Here we are dealing with static variables and hence method will also become static. Hence self is not required to pass as an argument. Because object is not used/passed here.

Modified code below. @staticmethod -> indicates that the method is static.

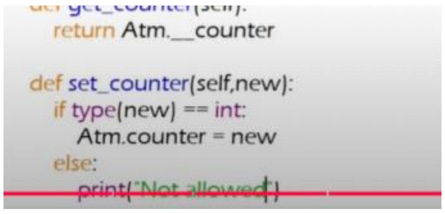


```
def __init__(self):
    # instance variable
    self.__pin = ""
    self.__balance = 0
    self.sno = Atm.counter
    Atm.__counter = Atm.__counter + 1

    print(id(self))

    #self.__menu()

@staticmethod
def get_counter():
    return Atm.__counter

@staticmethod
```

Beside image, in the set_counter method it should be Atm.__counter = new



```
    def __init__(self):
        # instance variable
        self.__pin = ""
        self.__balance = 0
        self.sno = Atm.counter
        Atm.__counter = Atm.__counter + 1

        print(id(self))

        #self.__menu()

    @staticmethod
    def get_counter():
        return Atm.__counter

    @staticmethod
    def set_counter(new):
        if type(new) == int:
            Atm.counter = new
        else:
            print("Not allowed")
```



```
    def get_counter(self):
        return Atm.__counter

    def set_counter(self,new):
        if type(new) == int:
            Atm.counter = new
        else:
            print("Not allowed")
```

method will also become static. Hence self is not required to pass as an argument. Because object is not used/passed here.

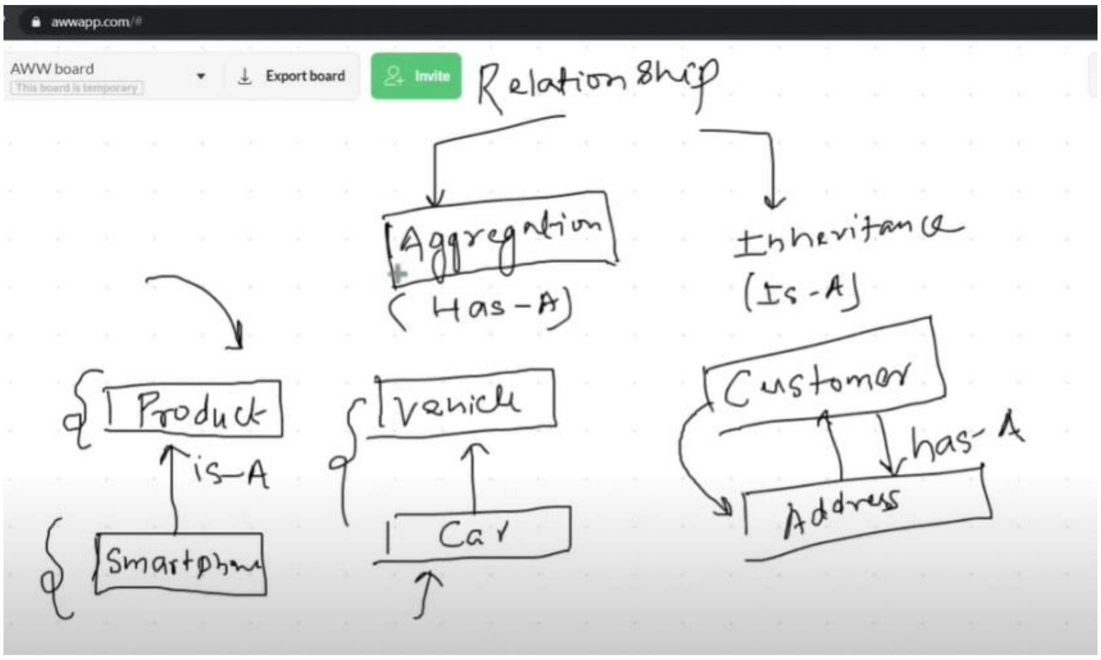
Modified code below. @staticmethod -> indicates that the method is static.

Beside image, in the set_counter method it should be Atm.__counter = new

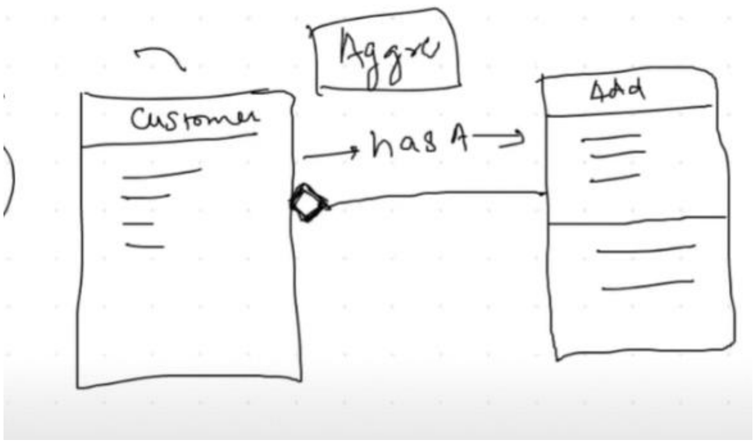
Class Relationship - Aggregation

23 March 2025 21:58

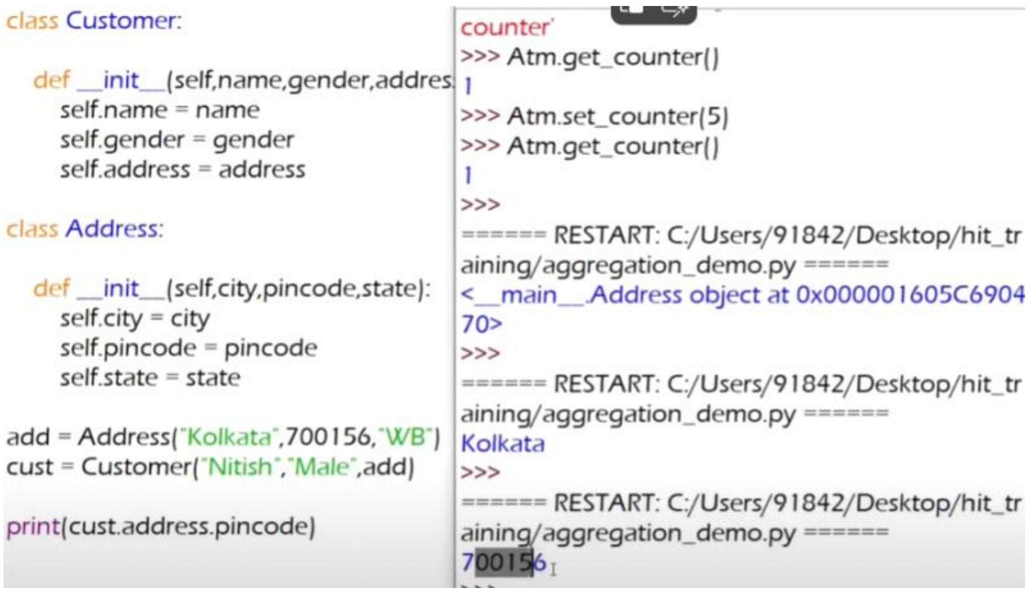
In practice there will more number of classes. It is said these classes can have 2 types of relationships -> 1. Aggregation (has - a - relation) 2. Inheritance (is - a - relation)



A basic class diagram would look like below:

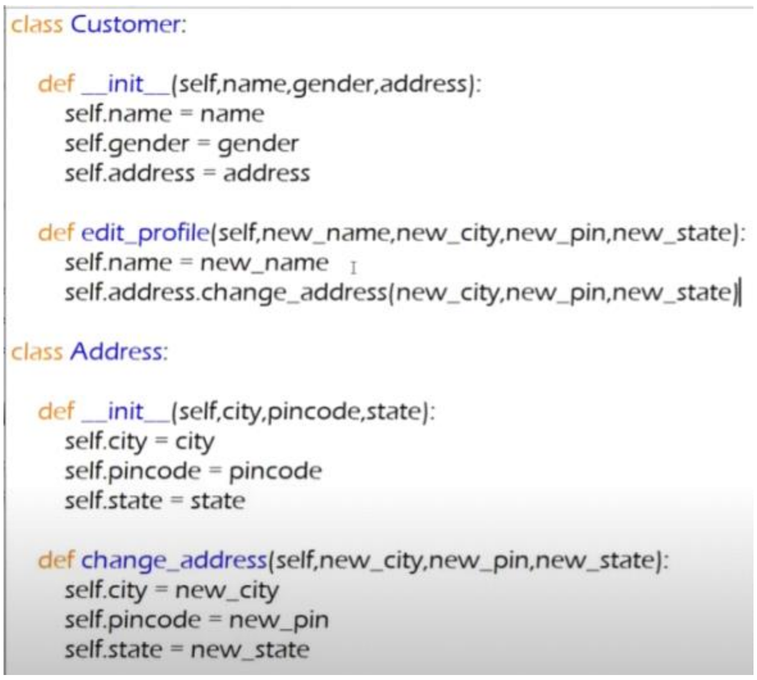


A sample program using aggregation



```
class Customer:

    def __init__(self,name,gender,addres
        self.name = name
        self.gender = gender
        self.address = address

class Address:

    def __init__(self,city,pincode,state):
        self.city = city
        self.pincode = pincode
        self.state = state

add = Address("Kolkata",700156,"WB")
cust = Customer("Nitish","Male",add)

print(cust.address.pincode)
```

```
counter'
>>> Atm.get_counter()
1
>>> Atm.set_counter(5)
>>> Atm.get_counter()
1
>>>
====== RESTART: C:/Users/91842/Desktop/hit_tr
aining/aggregation_demo.py ======
<__main__.Address object at 0x000001605C6904
70>
>>>
====== RESTART: C:/Users/91842/Desktop/hit_tr
aining/aggregation_demo.py ======
Kolkata
>>>
====== RESTART: C:/Users/91842/Desktop/hit_tr
aining/aggregation_demo.py ======
700156
```



```python
class Customer:

  def __init__(self,name,gender,address):
    self.name = name
    self.gender = gender
    self.address = address

  def edit_profile(self,new_name,new_city,new_pin,new_state):
    self.name = new_name
    self.address.change_address(new_city,new_pin,new_state)

class Address:

  def __init__(self,city,pincode,state):
    self.city = city
    self.pincode = pincode
    self.state = state

  def change_address(self,new_city,new_pin,new_state):
    self.city = new_city
    self.pincode = new_pin
    self.state = new_state
```

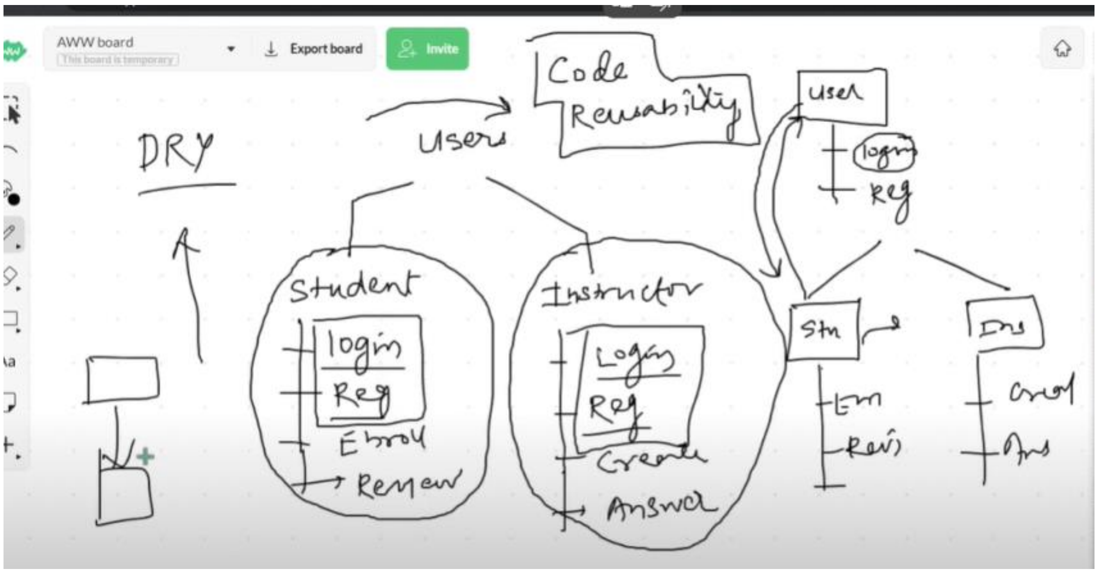
To modify the address, customer class is simply making use of Address class to update the value. Which is nothing but aggregation.

Inheritance

24 March 2025 10:44

Inheritance is a real word concept like legally, biologically, genetically … etc. **DRY** -> Don't Repeat Yourself => do not write the same code twice. Write once use many times.

The main intention of applying Inheritance is **Code Reusability.**

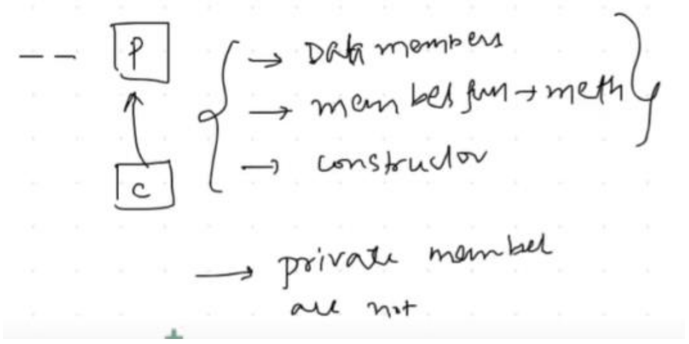


Beside shown is the example of Udemy platform where there are two main roles for user. Student and instructor. Both share some of the common functionalities such as login, registration.

So there we can make use of inheritance concept because same functionality is re used.

Always **Child** class inherits **Parent** class **.**

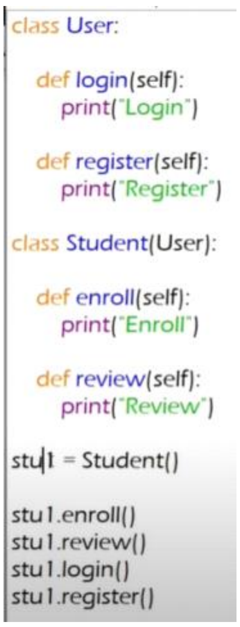
We can inherit the properties and data members of a class. Inheritance is always bottom to top but not top to bottom. Eg: child inherits father vice versa is false.



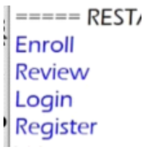
A Inherited class will have the access to Data Members, Methods, Constructor.

Note: However, Inherited class do not have Access to Private Members.

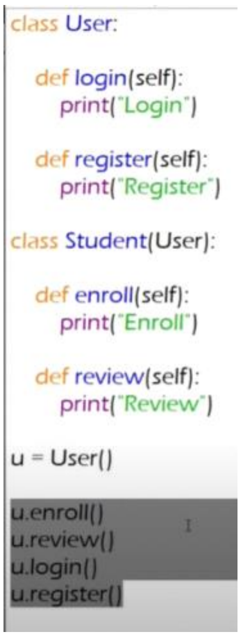
Syntax to inherit the class is class child_class(parent_class):



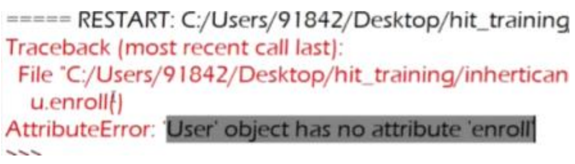
```
class User:

    def login(self):
        print("Login")

    def register(self):
        print("Register")

class Student(User):

    def enroll(self):
        print("Enroll")

    def review(self):
        print("Review")

stu1 = Student()

stu1.enroll()
stu1.review()
stu1.login()
stu1.register()
```



```
===== REST
Enroll
Review
Login
Register
```

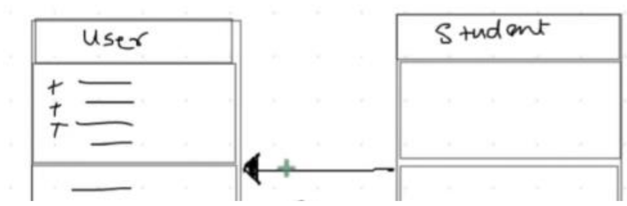


```
class User:

    def login(self):
        print("Login")

    def register(self):
        print("Register")

class Student(User):

    def enroll(self):
        print("Enroll")

    def review(self):
        print("Review")

u = User()

u.enroll()
u.review()
u.login()
u.register()
```

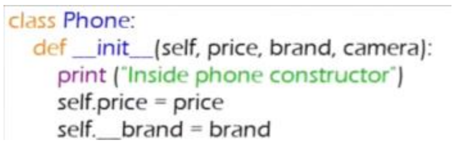


```
===== RESTART: C:/Users/91842/Desktop/hit_training
Traceback (most recent call last):
  File "C:/Users/91842/Desktop/hit_training/inhertican
    u.enroll()
AttributeError: 'User' object has no attribute 'enroll'
```

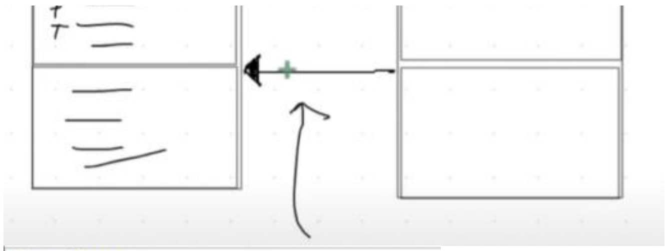
Child class can access the methods and members in parent class but parent class cannot access the methods of child class.

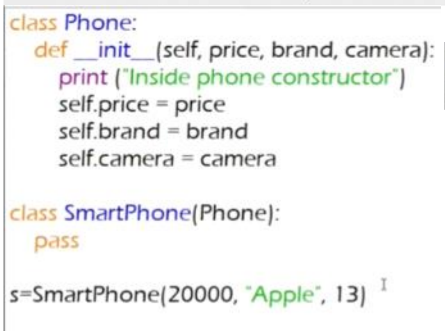


Let's see some of the code examples on inheritance below.

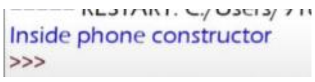


```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.price = price
        self.__brand = brand
```

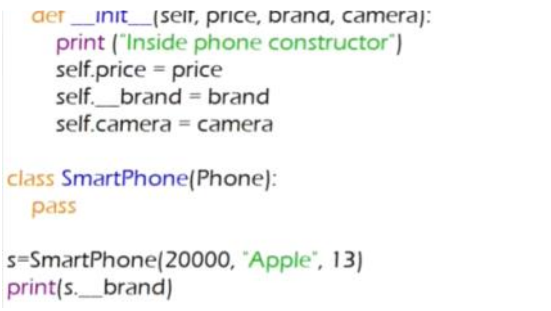




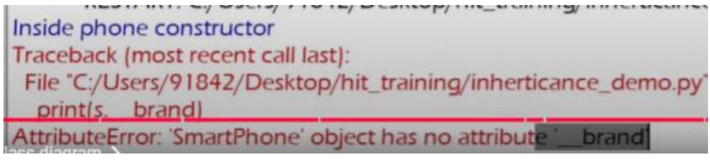
```
class Phone:
   def __init__(self, price, brand, camera):
      print ("Inside phone constructor")
      self.price = price
      self.brand = brand
      self.camera = camera

class SmartPhone(Phone):
   pass

s=SmartPhone(20000, "Apple", 13)
```



```
Inside phone constructor
>>>
```



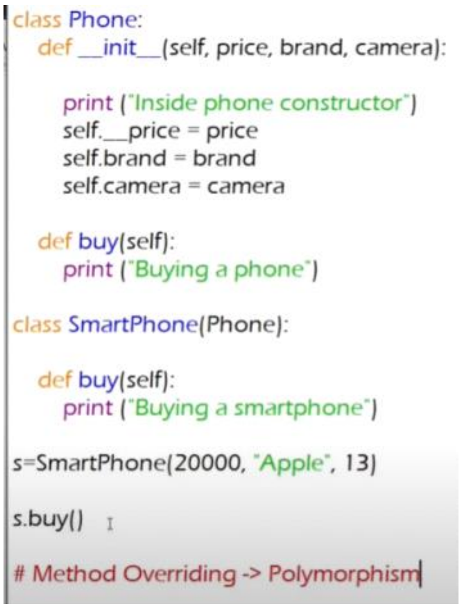
```
   def __init__(self, price, brand, camera):
      print ("Inside phone constructor")
      self.price = price
      self.__brand = brand
      self.camera = camera

class SmartPhone(Phone):
   pass

s=SmartPhone(20000, "Apple", 13)
print(s.__brand)
```



```
Inside phone constructor
Traceback (most recent call last):
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py"
    print(s.__brand)
AttributeError: 'SmartPhone' object has no attribute '__brand'
```

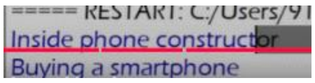
Polymorphism

27 March 2025 07:26



```
class Phone:
    def __init__(self, price, brand, camera):

        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):

    def buy(self):
        print ("Buying a smartphone")

s=SmartPhone(20000, "Apple", 13)

s.buy()

# Method Overriding -> Polymorphism
```

Here we have created a object of class Smartphone, buy is the method with same name in the parent and child classes. See which class method is executed on calling s.buy()

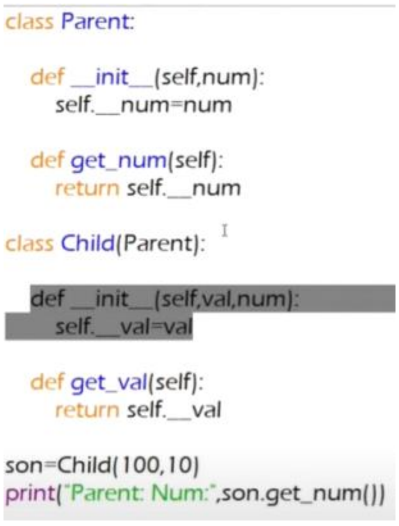


```
===== RESTART: C:/Users/91
Inside phone constructor
Buying a smartphone
```

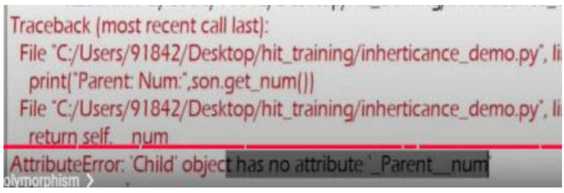
This process is also called **" Method Overriding".**

There are 3 concepts in Polymorphism -> 1. Method Overloading 2.Method Overriding 3.Operator Overloading.

If child class do not have Constructor, The Constructor of parent class gets invoked. If child class do have its own constructor, then the constructor of child class is invoked but not parent class as soon as the object of child class is created.
(Refer example below)



```
class Parent:

    def __init__(self,num):
        self.__num=num

    def get_num(self):
        return self.__num

class Child(Parent):

    def __init__(self,val,num):
        self.__val=val

    def get_val(self):
        return self.__val

son=Child(100,10)
print("Parent: Num:",son.get_num())
```



```
Traceback (most recent call last):
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", li
    print("Parent: Num:",son.get_num())
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", li
    return self.__num
AttributeError: 'Child' object has no attribute '_Parent__num'
```

Child constructor has invoked -> val = 100 but self.
_num = not defined because
Parent constructor not invoked => attritube error

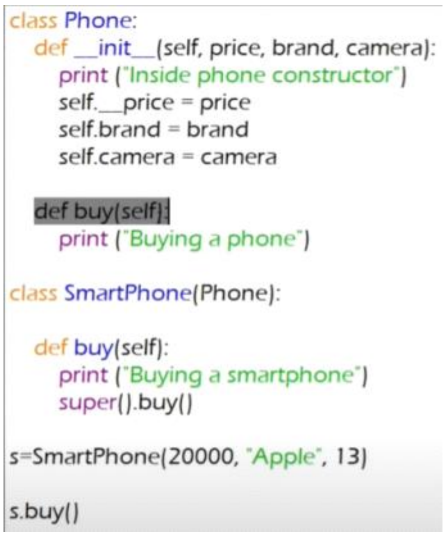
Super Keyword - User of Super()

27 March 2025 07:47

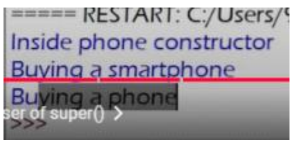
Super keyword is used to invoke the constructor and method of a parent class from the child class.

Note: Super keyword do not work outside the class. It has to be written in the method inside the class only.

Super keyword cannot access the attributes of the parent class. Only constructor and method can be invoked using super.



```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):

    def buy(self):
        print ("Buying a smartphone")
        super().buy()

s=SmartPhone(20000, "Apple", 13)

s.buy()
```

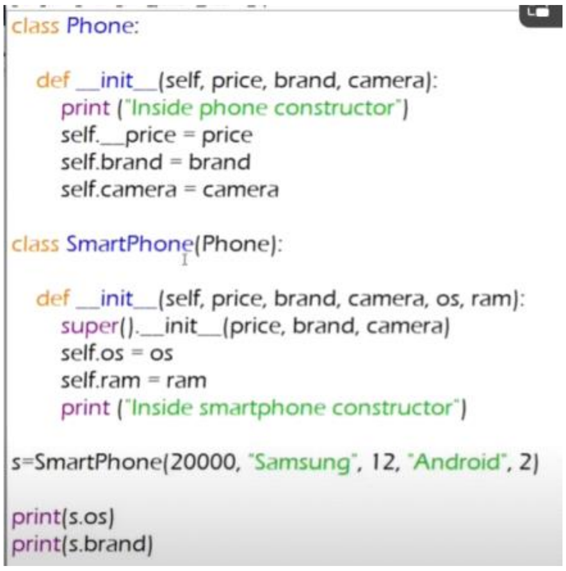


```
===== RESTART: C:/Users/
Inside phone constructor
Buying a smartphone
Buying a phone
>>>
```

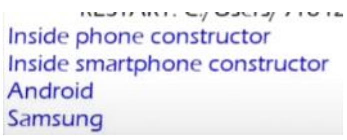
Observation:

Super keyword has to be written in the first place inside a constructor of child class (refer right)

Else it will not work.



```
class Phone:

    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):

    def __init__(self, price, brand, camera, os, ram):
        super().__init__(price, brand, camera)
        self.os = os
        self.ram = ram
        print ("Inside smartphone constructor")

s=SmartPhone(20000, "Samsung", 12, "Android", 2)

print(s.os)
print(s.brand)
```



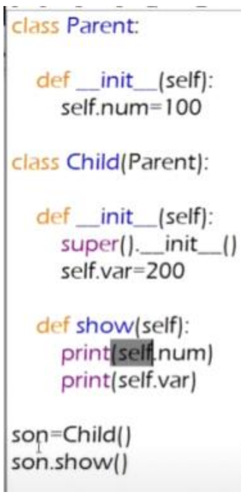
```
Inside phone constructor
Inside smartphone constructor
Android
Samsung
```

The use of super keyword:

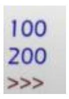
When we need to initialize half of the variables in parent class and half in child class, using the super keyword we can access the constructor of parent class when required. (code reusability)

Here in the beside example, phone class has basic details applicable for any kind of phone, in the smartphone (child class), it is specific to smartphone only.

When we want to show all the details, we can make use of super keyword.



```
class Parent:

    def __init__(self):
        self.num=100

class Child(Parent):

    def __init__(self):
        super().__init__()
        self.var=200

    def show(self):
        print(self.num)
        print(self.var)

son=Child()
son.show()
```



```
100
200
>>>
```

Q: Can we access the num of parent class using self.num from child class ?

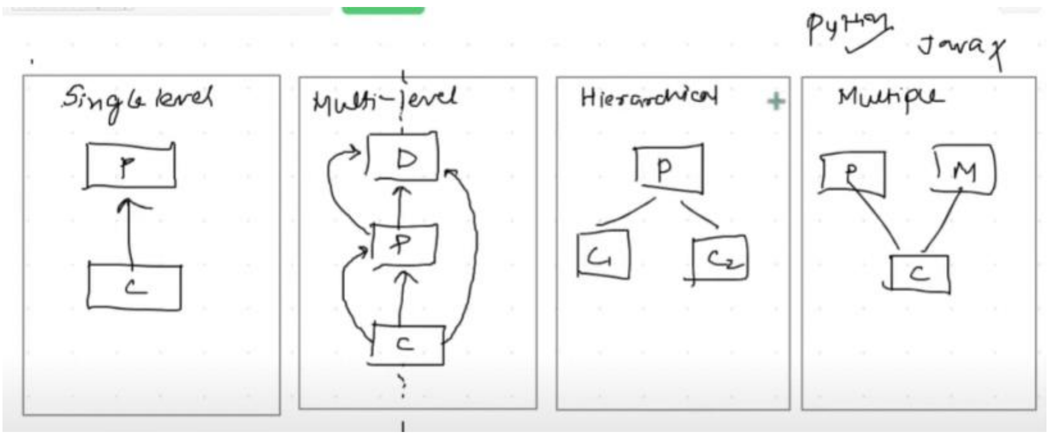
A: Yes, because self is nothing but the object at that instance, if object outside the class can actually access the num then using the self.num, we can print the value of num.

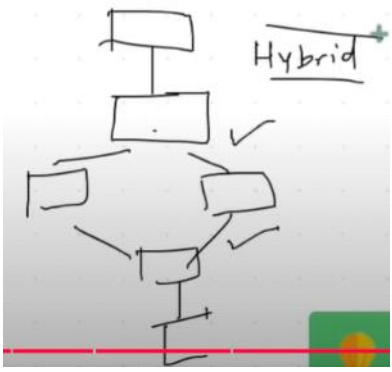
(refer image left)

Types Of Inheritance

28 March 2025 08:13

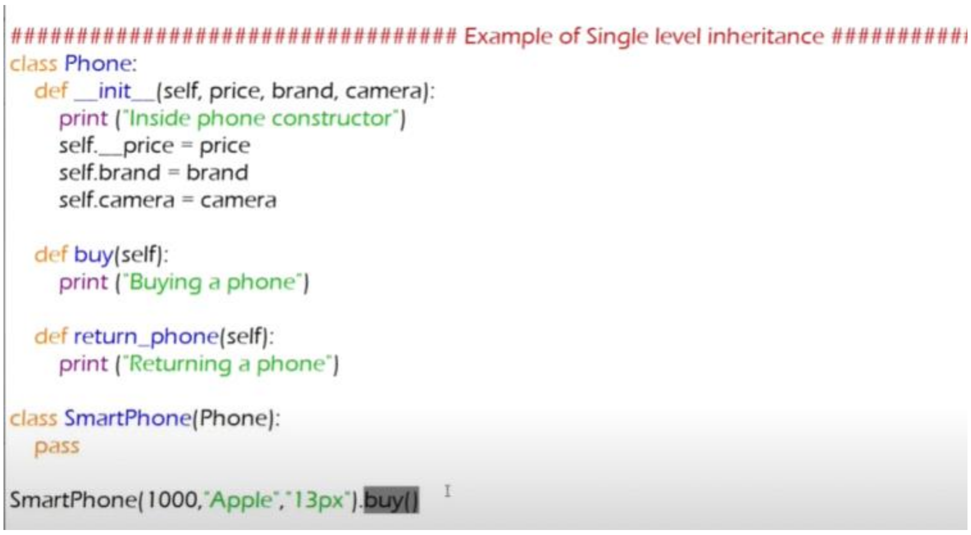
There are 4 types of inheritance in python -> 1. Single level 2. Multi-level 3.Hierarchal 4.Mutliple





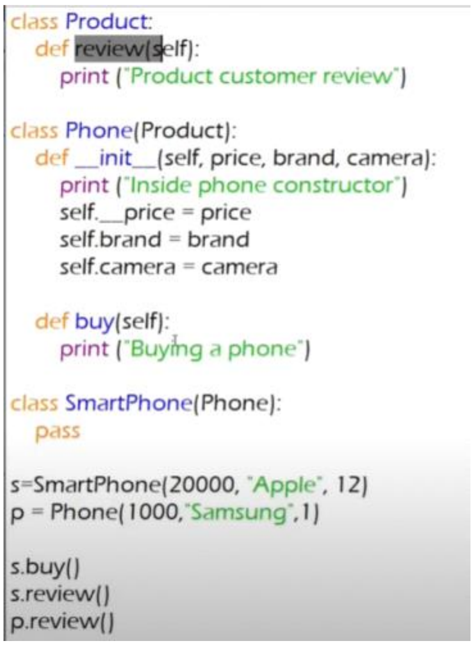
Multiple Inheritance is not present in Java.

We can have another possible type of inheritance -> Hybrid (Eg: combo of Multi-level + Hierarchal + Multiple).

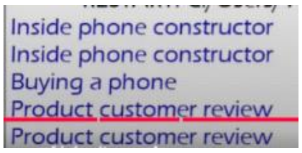


```
################################### Example of Single level inheritance ##########
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

    def return_phone(self):
        print ("Returning a phone")

class SmartPhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

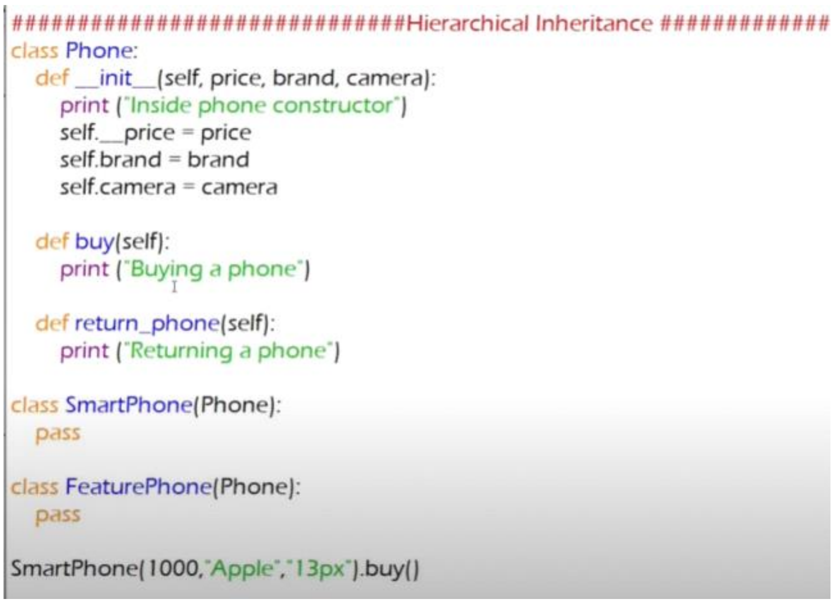
Example for Multi-Level Inheritance below:



```
class Product:
    def review(self):
        print ("Product customer review")

class Phone(Product):
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class SmartPhone(Phone):
    pass

s=SmartPhone(20000, "Apple", 12)
p = Phone(1000,"Samsung",1)

s.buy()
s.review()
p.review()
```

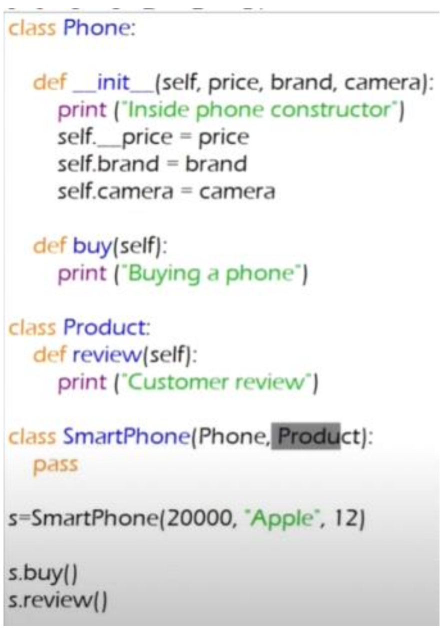


```
Inside phone constructor
Inside phone constructor
Buying a phone
Product customer review
Product customer review
```

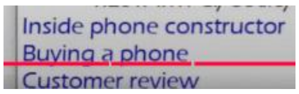


```
###############################Hierarchical Inheritance ############
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

    def return_phone(self):
        print ("Returning a phone")

class SmartPhone(Phone):
    pass

class FeaturePhone(Phone):
    pass

SmartPhone(1000,"Apple","13px").buy()
```

Example for Multiple Inheritance below:



```
class Phone:

    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:
    def review(self):
        print ("Customer review")

class SmartPhone(Phone, Product):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
s.review()
```



```
Inside phone constructor
Buying a phone
Customer review
```

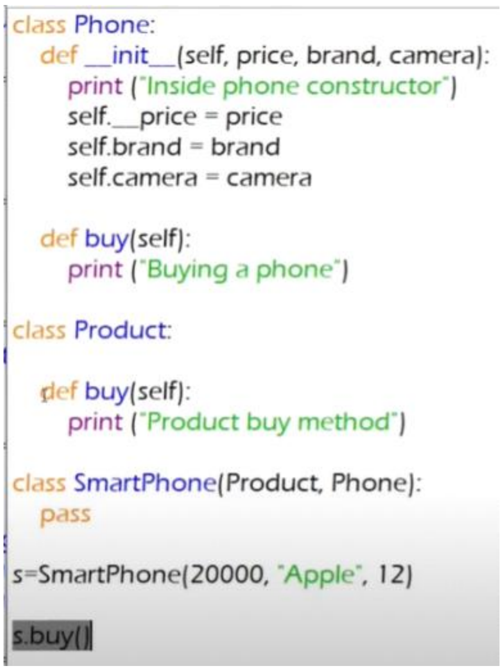
Here in this example, Q is which class constructor will be invoked ?

A: It will invoke the constructor of first parent (here Phone), if not present in 1st parent class, will invoke the constructor of 2nd parent class.

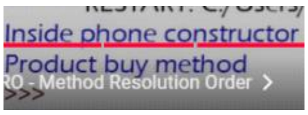
MRO - Method Resolution Order

28 March 2025 10:55

Possible Conflict with Multiple Inheritance: if there is method with same name in both the parent classes. Which method will be invoked when called with the object of child class.



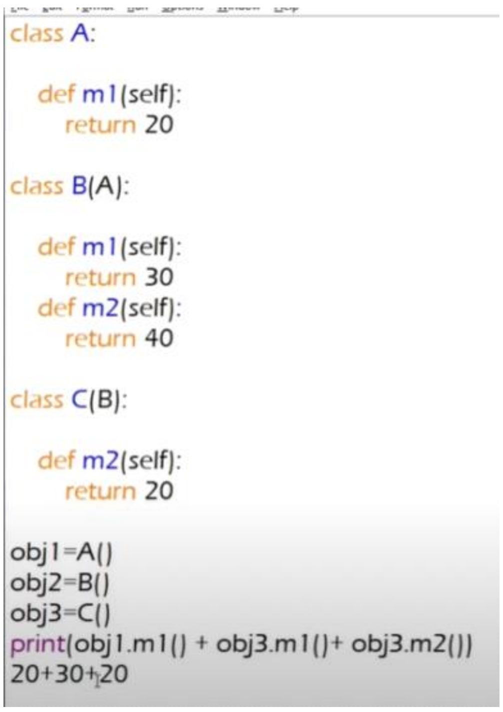
```
class Phone:
    def __init__(self, price, brand, camera):
        print ("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print ("Buying a phone")

class Product:

    def buy(self):
        print ("Product buy method")

class SmartPhone(Product, Phone):
    pass

s=SmartPhone(20000, "Apple", 12)

s.buy()
```



```
Inside phone constructor
Product buy method
>>>
```

Here buy method of product is invoked because Product is inherited first in the smartphone class.

i.e. it follows the order in which the class is being inherited.



```
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        return 30
    def m2(self):
        return 40

class C(B):

    def m2(self):
        return 20

obj1=A()
obj2=B()
obj3=C()
print(obj1.m1() + obj3.m1()+ obj3.m2())
20+30+20
```

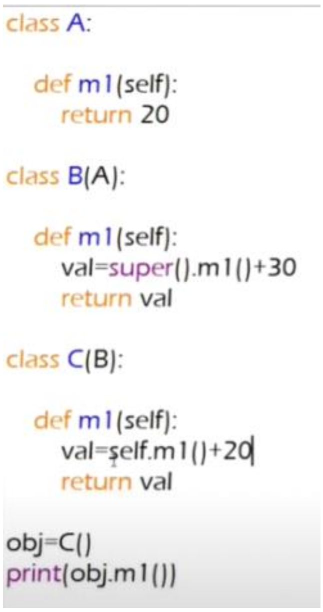
Here beside example, obj1.m1 will invoke m1 method of class A (own method).

Obj3.m2 will invoke m2 method of class C (own method, method overriding)

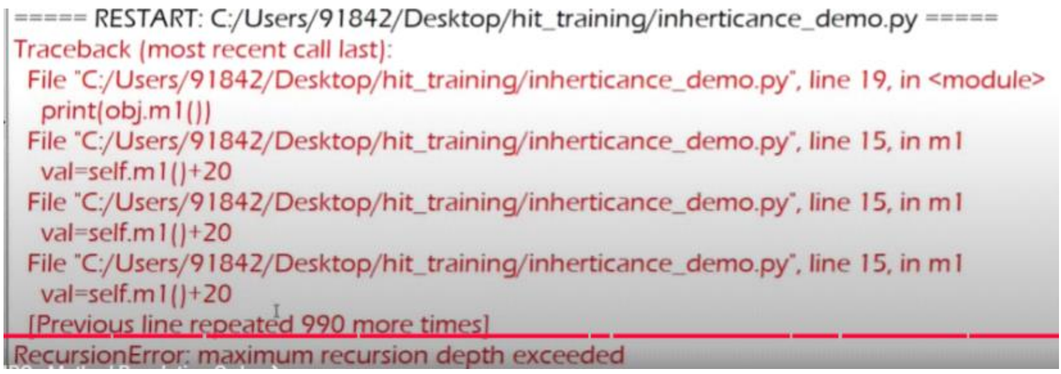
Q is obj3.m1 will invoke m1 of A or m1 of B ?

Ans: If available child class will invoke the m1 method of parent class B rather than grandparent class A.

This is the method resolution order.



```
class A:

    def m1(self):
        return 20

class B(A):

    def m1(self):
        val=super().m1()+30
        return val

class C(B):

    def m1(self):
        val=self.m1()+20
        return val

obj=C()
print(obj.m1())
```



```
===== RESTART: C:/Users/91842/Desktop/hit_training/inherticance_demo.py =====
Traceback (most recent call last):
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", line 19, in <module>
    print(obj.m1())
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", line 15, in m1
    val=self.m1()+20
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", line 15, in m1
    val=self.m1()+20
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", line 15, in m1
    val=self.m1()+20
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
```

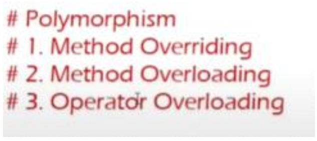
Here self.m1() in m1 method of class C is nothing but obj.m1() because self => obj

So it is going into infinite recursive loop i.e. calling same method infinite times.

Method Overloading & Operator Overloading

28 March 2025 11:47

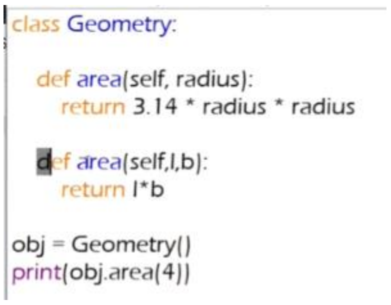
Polymorphism: Poly => Multiple + Morph => Faces; => Same thing behaving in multiple ways i.e. 1 function with many behaviors.



```
# Polymorphism
# 1. Method Overriding
# 2. Method Overloading
# 3. Operator Overloading
```

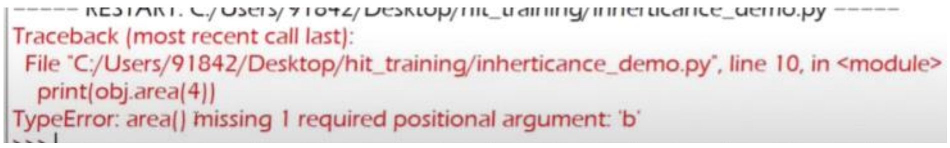
Method Overloading:

Same method name + different num of parameters => behavior different based on input parameters.



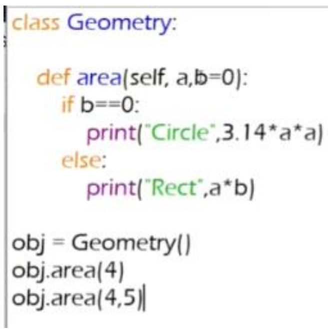
```
class Geometry:

    def area(self, radius):
        return 3.14 * radius * radius

    def area(self,l,b):
        return l*b

obj = Geometry()
print(obj.area(4))
```

This code may work in java but in python it doesn't. See error below:

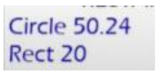


```
Traceback (most recent call last):
  File "C:/Users/91842/Desktop/hit_training/inherticance_demo.py", line 10, in <module>
    print(obj.area(4))
TypeError: area() missing 1 required positional argument: 'b'
```

Note: Technically method overloading as a definition will not work in python (because method overriding happened), but can achieve similar behavior by making some modifications.(refer right)



```
class Geometry:

    def area(self, a,b=0):
        if b==0:
            print("Circle",3.14*a*a)
        else:
            print("Rect",a*b)

obj = Geometry()
obj.area(4)
obj.area(4,5)
```



```
Circle 50.24
Rect 20
```

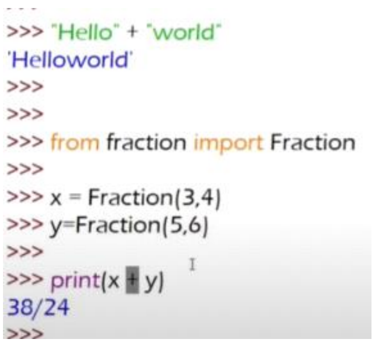
Here we can route the function and so achieve various behavior with same function.

Operator Overloading:

+ is used in string, int, fraction datatypes separately which means it is not an integer addition but pre-defined for each of the data types. This is nothing but "Operator Overloading".

In the fraction example, + operator function is overridden with the custom method defined for the fraction addition.

Operator Overloading can be achieved by using magic methods as studied earlier.



```
>>> "Hello" + "world"
'Helloworld'
>>>
>>>
>>> from fraction import Fraction
>>>
>>> x = Fraction(3,4)
>>> y=Fraction(5,6)
>>>
>>> print(x + y)
38/24
>>>
```