28-3-23.

Recursion: It is a phenomenon of a function calling itself until the given condition is satisfied.

-> Functions that are satisfying the recursion are called as recursive functions.

Syntax:

```
def func_name(args, intitialization):
        Statements/instructions
        fun_name(args, increment/decrement)
func_name(args, args)
```

-> If in case we are using above syntax, we are not writing any conditional statement with return keyword inside the function.

-> It will execute 'n' no of times (the self-call will happen until stack memory is full).

-> for each and every function call an instance memory will be allocated in memory segmentation.

Syntax:-

with return statement + func call

```
def func_name(args,):
    if <condition>:
        statement/instruction
        return value
    Statements/instructions.
```

return func_name(args, increment/decrement)

func_name(args)

Without return Statement + func call

```
def func_name(args,):
    if <Condition>:
        Statement/instruction
        return value
    Statements / instructions
    func_name(args, increment/decrement)
func_name(args)
```

other way:

With return Statement + func call

```
def func_name(args,):
    if <condition>:
        Statement/instruction
        return func_name(args, increment/decrement)
    Statements / instructions
    return value
func_name(args).
```

29-3-23

WAP: to generate sequence of values within range function with for loop:

```
def fetch (start, end)
    for i in range (start, end+1):
        print (i)

fetch (1, 10)
```

O/P :-

1
2
.
.
.
10.

WAP to generate sequence of values within range function with while loop:

```
def fetch (start, end)
    while start < end+1 :
        print (start, end = ' ')
        start += 1
```

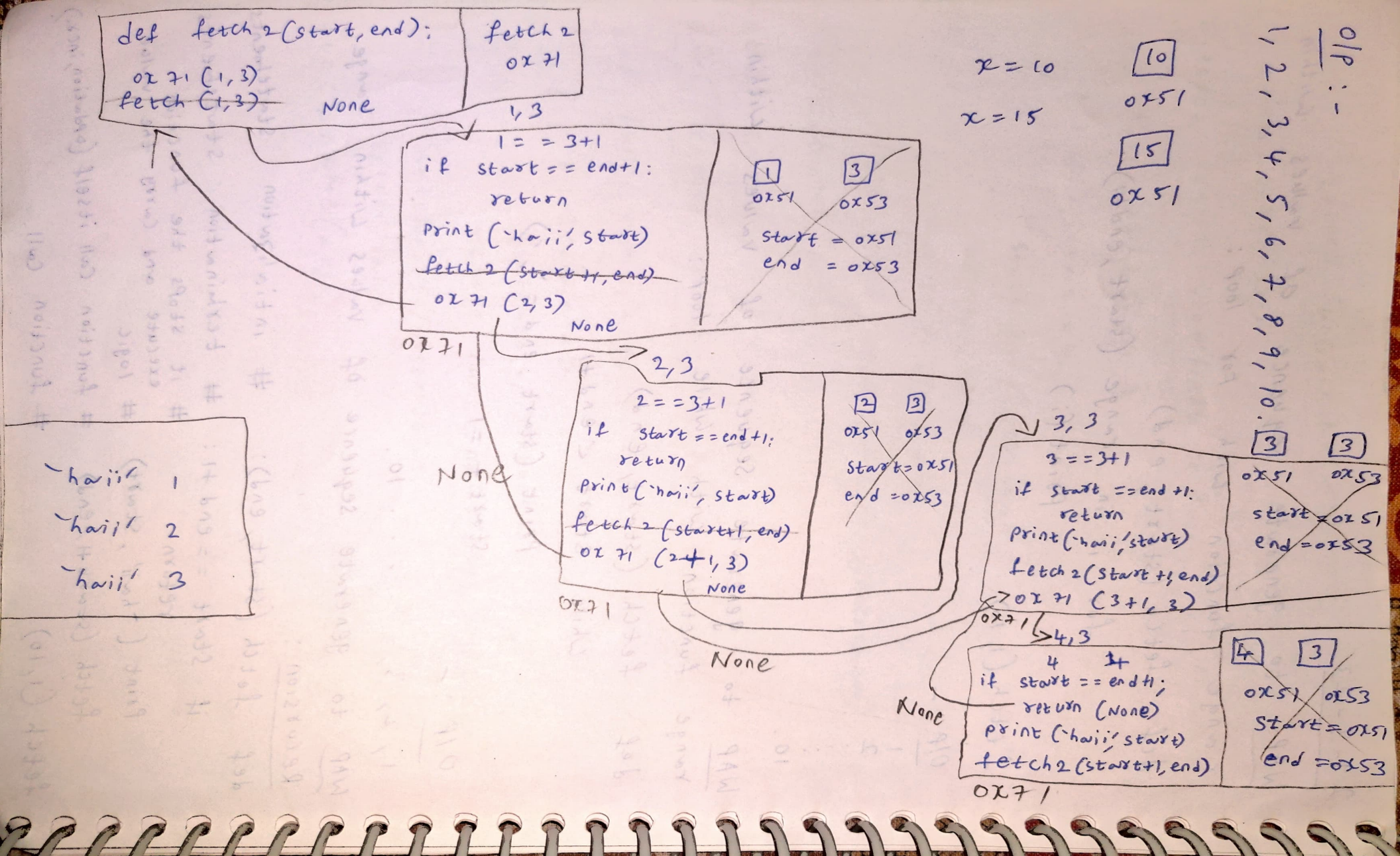
O/P :-

1, 2, 3, ... 10.

WAP to generate sequence of values within range.

Recursion :

```
def fetch (start, end):          # intialization statements
    if start == end+1:           # termination statements
        return                   # it stops the function execute and carry the value.
    Print ("hai", start)         # logic
    fetch (start+1, end)         # function call itself (updation, incr.)
fetch (1, 10)                    # function call.
```

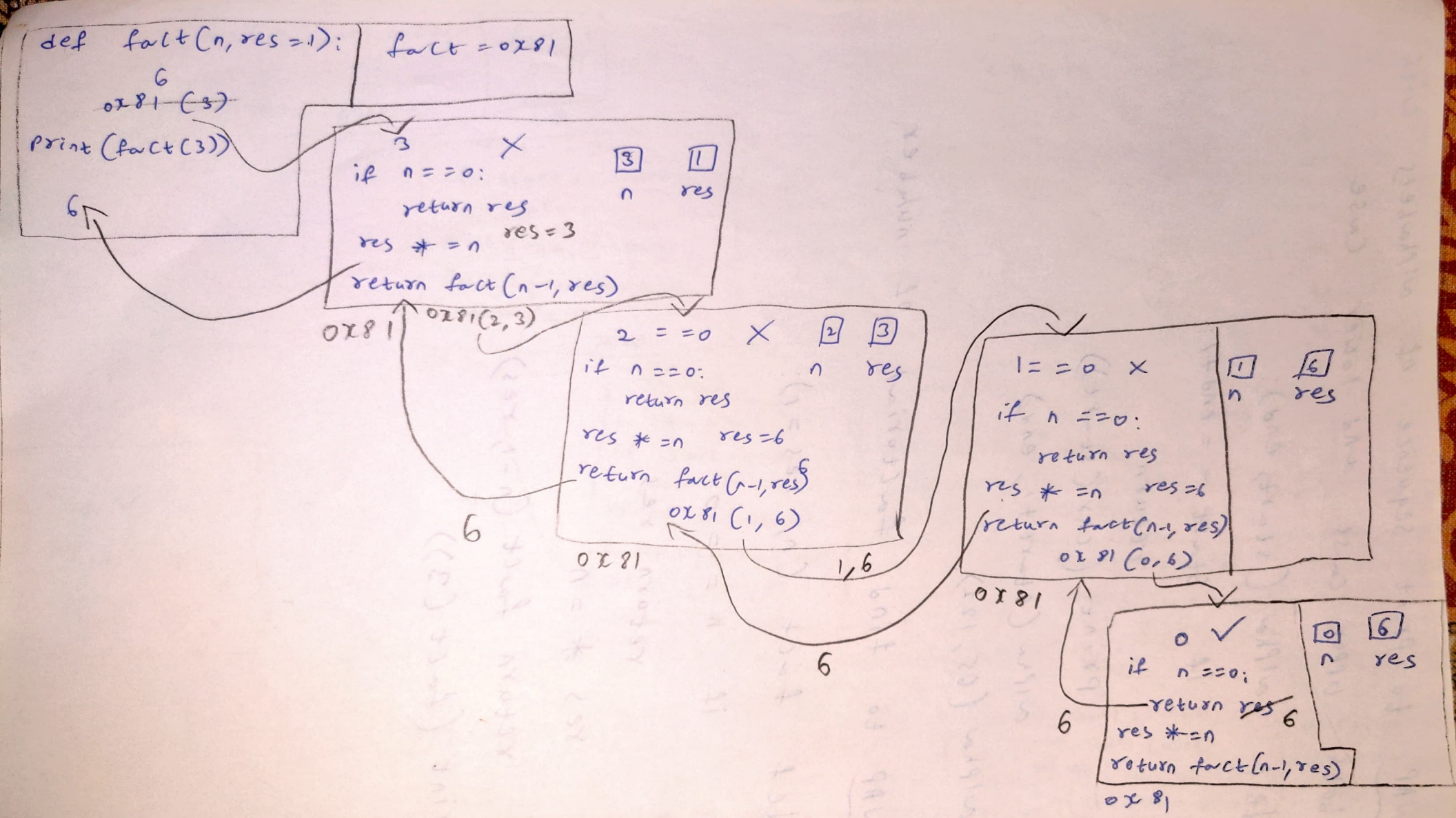

def fetch2(start, end):
0x71 (1, 3)
fetch (1, 3)
None
fetch2
0x71
1, 3
1 == 3+1
if start == end+1:
return
Print ('haii', start)
fetch2 (start+1, end)
0x71 (2, 3)
None
1
0x51
3
0x53
start = 0x51
end = 0x53
0x71
2, 3
2 == 3+1
if start == end+1:
return
Print ('haii', start)
fetch2 (start+1, end)
0x71 (2+1, 3)
None
2
0x51
3
0x53
start = 0x51
end = 0x53
None
0x71
3, 3
3 == 3+1
if start == end+1:
return
Print ('haii', start)
fetch2 (start+1, end)
0x71 (3+1, 3)
None
3
0x51
3
0x53
start = 0x51
end = 0x53
0x71
4, 3
4 == 4
if start == end+1:
return (None)
print ('haii', start)
fetch2 (start+1, end)
None
4
0x51
3
0x53
start = 0x51
end = 0x53
0x71
'haii' 1
'haii' 2
'haii' 3
x = 10
x = 15
10
0x51
15
0x51
O/P :-
1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

WAP to print sequence of alphabets with both upper case and lower case

```
def alpha (start, end):
    if start == end+1:
        return
    print (chr (start))
    alpha (start+1, end)
alpha (65, 122).
```

WAP to find factorial of number.

```
def fact (n, res = 1):
    if n == 0:
        return res
    res *= n
    return fact (n-1, res)
print (fact (3))
```


def fact(n, res=1):
if n==0:
return res
res *= n
return fact(n-1, res)
print(fact(3))
fact = 0x81
6
0x81 (3)
3
res=3
n
res
0x81
0x81(2,3)
2==0
1,6
0x81 (1, 6)
1==0
0x81 (0,6)
0
6
0x81

30-3-23

WAP to sum of integers in given collection.

```
l = [10, 15, 16, 'a', 'b', 'c', 15.05, True, [1,2],
     100, 200]
def sumof(l, sum=0, i=0):
    if i==len(l):
        return sum
    if type(l[i])==int:
        sum+=l[i]
    return sumof(l, sum, i+1)
res = sumof(l)
print(res)
```

O/P :-

341

WAP to swapcase.

```
l = 'PySpIdEr S*143'
def swapcase(l, s='', i=0):
    if i==len(l):
        return s
    if 'a'<=l[i]<='z':
        s+=chr(ord(l[i])-32)
    elif 'A'<=l[i]<='Z':
        s+=chr(ord(l[i])+32)
    else:
        s+=l[i]
    return swapcase(l, s, i+1)
swapcase(l)
```

O/P :-

'PYsPiDeRs *143'

WAP fibonocci series (while loop)

```
l = [0,1]
a = 0
b = 1
Start = 3
end = 10
while start <= end:
    c = a+b
    a = b
    b = c
    l+ = [c]
    Start += 1
print(l)
```

O/P :-

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

program :-

```
def fibo (l=[0,1], a=0, b=1, Start=3, end=10):
    if Start == end+1:
        return l
    c = a+b
    a = b
    b = c
    l+ = [c]
    return fibo (l, a, b, Start+1, end)
fibo()
```

O/P :-

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

armstrong number

```
t = 153
n = str(t)
sum = 0
i = 0
while i < len(n):
    sum += int(n[i] ** len(n))
    i += 1
print(sum)
if t == sum:
    print('armstrong')
else:
    print('not a arm strong')
```

O/P :-

153

armstrong.

Program :- (recursion)

```
def armstrong(t, sum = 0, i = 0):
    n = str(t)
    if i == len(n):
        return sum.
    sum += int(n[i]) ** len(n)
    return armstrong(t, sum, i+1)
num = 153
if num == armstrong(num):
    print("armstrong")
else:
    print("not armstrong")
```

O/P :-

153

armstrong.

Wap to display series of armstrong numbers in given range.

```
start = 1
end = 2500
while start <= end:
    t = start
    n = str(t)
    sum = 0
    i = 0
    while i < len(n):
        sum += int(n[i]) ** len(n)
        i += 1
    if t == sum:
        print(sum)
    start += 1
```

Olp :-

```
1
2
3
4
5
6
7
8
9
153
370
371
407
1634
```

Program :-

```
def armstrong(t):
    n=str(t)
    sum=0
    i=0
    while i<len(n):
        sum+=int(n[i])**len(n)
        i+=1
    return sum
# armstrong(153)
```

O/P :-

1
2
3
4
5
6
7
8
9
153
370
371
407
1634.

```
def series(start=1, end=1000):
    while start<=end:
        if start==armstrong(start):
            print(start)
        start+=1
series(1,2500)
```

armstrong using recursion :- (in given range)

```
def armstrong(t, sum=0, i=0):
    n=str(t)
    if i==len(n):
        return sum
    sum+=int(n[i])**len(n)
    return armstrong(t, sum, i+1)
# num=153
```

```
def series(start=1, end=1000):
    if start == end+1:
        return
    if start == armstrong(start):
        print(start)
    series(start+1, end)
series()
```

O/P:-

1
2
3
4
5
6
7
8
9
153
370
371
407.

31-3-23.

program :-

```
def sample(i=0, end=5):
    if i==end+1:
        return
    print(i, end=" ")
    sample(i+1, end)
    print(i, end=" ")
sample(1, 8)
```

O/P :- 1 2 3 4 5 6 7 8 8 7 6 5 4 3 2 1.

program :-

```
def sample(i=0, end=5):
    Print(i, end=" ")
    if i == end:
        return
```

```
    sample(i+1, end)
    print(i, end="")
sample(0,5)        O/P :- 0 1 2 3 4 5 4 3 2 1 0.
```

Program :- Case ①

```
def demo(st, i=0):
    if i==len(st):
        return
    if 'a'<=st[i]<='z':
        print(chr(ord(st[i])-32))
    else:
        print(st[i])
    demo(st, i+1)
    if 'a'<=st[i]<='z':
        print(chr(ord(st[i])-32))
    else:
        print(st[i])
demo("hello")
```

O/P :-

HELLOOLLEH

program : Case (2) :

```
# heLLo
# HELLooLLEH

res = ""
def demo (st, i=0):
    global res
    if i == len(st):
        return
    if 'a' <= st[i] <= 'z':
        res += chr(ord(st[i]) - 32)
    else:
        res += st[i]
    demo(st, i+1)
    res += res[i]
demo("heLLo")
print(res)
```

o/p :-

HELLooLLEH.

Program :-

```
St1 = "Pots  read  saW  Pot  tab"
St2 = "stoP  dear  WAs  toP  Bat"

def ulen(st):
    count = 0
    for i in st:
        count += 1
    return count

def iupper(st, res = "", i=0):
    if i == len(st):
        return res
    if 'a' <= st[i] <= '}':
        res += chr(ord(st[i]) - 32)
    else:
        res += st[i]
    return iupper(st, res, i+1)

def isorted(st):
    st = list(st)
    for i in range(0, len(st)):
        for j in range(i, len(st)):
            if st[i] > st[j]:
                st[i], st[j] = st[j], st[i]
```

```
if  ulen(st1) == 4 len (st2):
    if isorted (iupper(st1)) ==
                    isorted (iupper (st2)):
        print ("anagram")
    else :
        print ("not anagram")
else:
    print ("not anagram").
```

Shortcuts method for converting a while loop into a recursive program.

1. initially first write down the while loop program.

```
a = "helloworld"
st = ""
i = 0
while  i! = len(a):
    if  'a' <= a[i] <= 'z':
        st += chr(ord(a[i]) - 32)
    else:
        st += a[i]
    i+=1
Print(st)
```

(i) in the above programs we are taking some inputs, whatever input is there, those inputs written inside the function definition as formal arguments.

ex:

```
def upper (a='helloworld', st="",
           i =0):
```

2. In the while loop we are written the condition, whatever condition is there. opposite of that condition arite inside if statement (termination statement) and return the value inside the recursion function. (it in case we return the output of that programs than give the variable name in return statement.

ex:

```
if i ==len(a):          # termination
    return st           condition
```

3. In the while loop whatever true block statement is there, those statements copy and paste it after termination Condition inside the

recursion function (except increment/decrement statement).

ex:

```
if 'a' <= a[i] <= 'z':
    st += chr(ord(a[i])-32)
else:
    st += a[i]
```

4. whatever inc. and dec. statement is there; those statements write it inside the function call.

note (first give normal args and after inc./dec. statement).

ex:

```
return upper(a, st, i+1)
```

5. Finally give the normal function call inside the main space.

example:

```
def upper(a = "hello world",
          st = "", i=0):
    if i == len(a):          # termination
        return st            condition
```

```
if 'a'<=a[i]<='z':
        st += chr(ord(a[i])-32)
else:
    st += a[i]
return upper(a, st, i+1)
upper()
def upper(a, st, i):
    if i==len(a):          # termination condition
        return st
    if 'a'<=a[i]<='z':
        st += chr(ord(a[i])-32)
    else:
        st += a[i]
    return upper(a, st, i+1)
upper("hello world", "", 0)
def upper(a, st, i):
    if i!=len(a):
        if 'a'<=a[i]<='z':
            st += chr(ord(a[i])-32)
        else:
            st += a[i]
        return upper(a, st, i+1)
    return st                  # termination condition.
upper("hello world", "", 0)
```

1-4-23.

Object oriented programming Structure/ system

→ OOPS is used for storing the data in systematic and structural format. (or) manner.

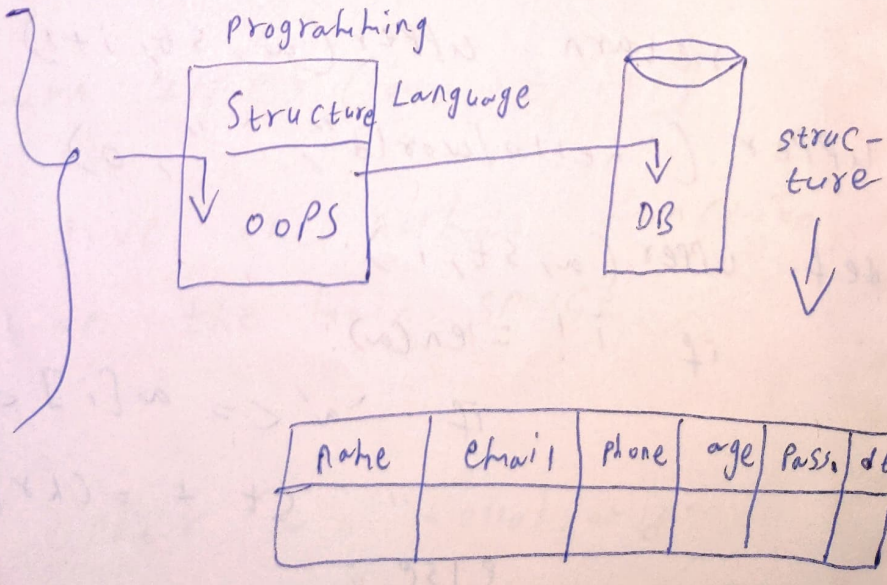
→ By the help of OOPS we can design or create the user defined collections.

→ By using OOPS we are ~~follow~~ solving the real time object related problems.

Structure

① procedural PL ② functional PL ③ OOPS

name
email
Phone
age
password
repassword



name	email	phone	age	Pass.	dt

-> The phenomenon of working with real world entities or real time objects will lead us to work with the concept called object oriented programming language.

-> It is used for solving the real-time entity problems.

-> It is used for storing and manipulating the real-time entity data.

-> As we are working with real world entities, the built in data types are not sufficient, enough to store the details of real world entities and because real world entities will have various types, various properties, various structure, various functionality are present, this all are not suitable to store or retrieve the information by using built-in data types.

-> so that's why we are going to use the user-defined data types called Class and object.

-> through class and object, we are creating our own states (properties) and behaviour (functionalities) related to real world entities.

Why is oops used ?

-> organize code and classify data and functions providing a clear structure.
-> easy maintanance and modification.
-> reduce code ; reduce code difficulty, by enabling code reusability.
-> inheritance which helps us to inherit data from one class to another class.
-> encapsulation , where secure data by implementing private variables.

Class :

-> It is a structure that specifies what to store and what to do.
-> Class is a blueprint of an object; Class is a plan (structure) to create an object.
-> Class is a user-defined data-type that specifies the states and behaviours of a real world entity.

-> States are nothing but properties/ attributes/ members or data about real world entities.

-> Behaviour is nothing but the functionality / operations of a real world entity.

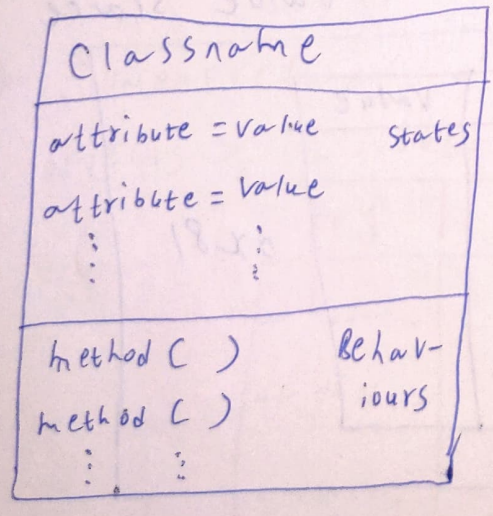
-> Behaviour is nothing but a feature that is provided for the entity.

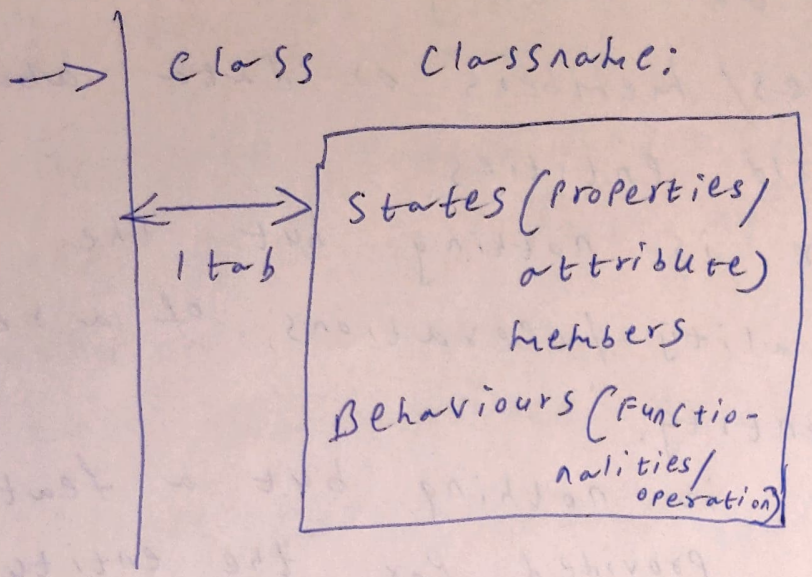
-> A class can be created using the keyword called 'class'

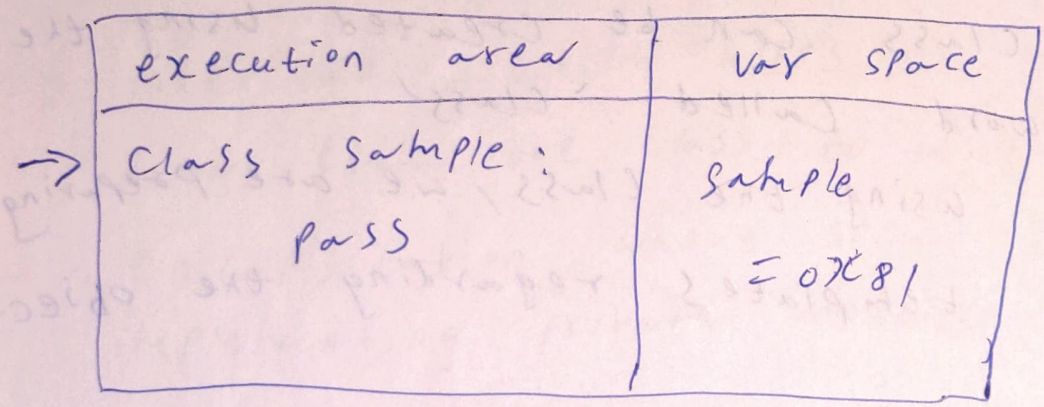
-> By using the class, we are preparing the templates regarding the object.

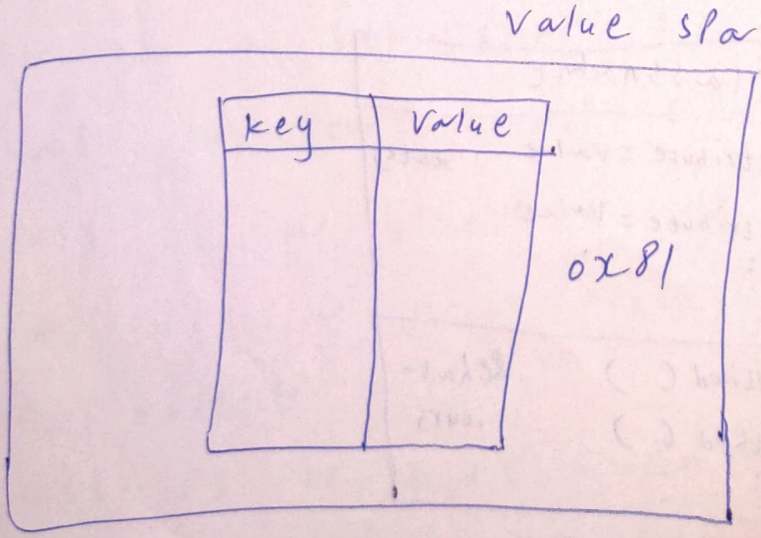
Class model:

UML




class Classname:
1 tab
States (properties/ attribute)
members
Behaviours (Functionalities/ operation)


execution area
Var space
class sample:
pass
sample
= 0x81


value space
key
value
0x81

3-4-23.

```
print(globals())
class person:
    pass
print(globals())
>>> type(person)
<class 'type'>
person
<class '__main__.person'>
>>> person.__dict__
mappingproxy({'__module__': '__main__', '__dict__': <attribute '__dict__' of 'person' objects>, '__weakref__': <attribute '__weakref__' of 'person' objects>, '__doc__': None})
```

key	value
__module__	__main__
__dict__	address
__weakref__	address
__doc__	address

Object :

-> An object is a variable of type Class.

-> Class will specify what to store and what to do, whereas the object will store what is specified by the class and does what it said object in real world entity.

Syntax :

object_name = class_name (arguments)

-> When we call the class then duplicate, class memory (instance memory) is created.

-> Arguments are nothing but the data that we wanted to store it inside the object memory (sometimes arguments are not mandatory).

-> When we create an object after execution of the program, as soon as the control reaches the object creation statement then it creates the dictionary memory in the value_space. the address of the dictionary will be stored in the object name inside the variable space.

-> from the class all the class members will be derived to the object dictionary.

-> checks whether there exists a method called __init__ or not if it exists then it automatically calls for that constructor method.

-> All the object members from the constructor will be stored in the object dictionary.

Example :

```
obj = A()
print(obj)
print(type(obj))
```

<__main__.A object at 0x7f1e15e7768>

<class '__main__.A'>

```
class Person:
    name = 'rahma'
obj1 = Person()
print(obj1)
print(Person.__dict__)
print(obj1.__dict__)
```

O/P :-

{ }

person.name
'rahma'
obj1.name
'rahma'.

Default functionalities for object states :-

States or properties : states are the data that defines the real world entity. states is nothing but attributes / properties of a real world entity.

ex : if we consider a bank as a class then bankname, regno, MBL (main branch location) .... these all comes under the category of states of bank.

-> If we consider a employees as a class then company_name, regno, MBL, these all are common for all the employees.

-> employee name, email, phone, sal ..... these are all data under the category of states of the emlpoyee.

States are segregated into 2 categories:

1. generic States (static states / class variables / class states) (Static States / Class Member's / Class data / generic data / generic attributes / Class attributes)
2. Specific States (non - Static states / Object variables / object states) (specific data / object data / object member's / object attribute / object properties)

1. Generic States: These are the properties / attributes that are real world entities, this states are common for all the objects and which belong to the Class; these will be stored inside the class dictionary and can be accessed by using the class name or object names. This States is common for class and objects

ex: If we consider a bank as an example bank name, location, Phone, ifsc, loan rate or FD rate of interest will be common for all the customers of the bank so bank name, location, Phone, loan rate and FD rate are called as generic states.

-> these are the data that are stored in the variable which is called as static variable.

-> static variables / attributes or class members: these are the variables that contains the generic data that defined for all the real world entities which are stores inside class memory and can be accessed by all the members (either class name or objects name)

-> Modification / deletion of generic states by using class-name will have an impact on class dictionary and also all the derived object dictionaries.

-> modification of generic states by

using object_name will have an impact on that particular object dictionaries only, and it will not impact on class dict and some other object dictionaries.

creating the generic attributes of the class:

syntax:

```
class Cname:
    var1 = val1
    var2 = val2
    var3 = val3
```

accessing the generic attributes of the class:

All these static members can be accessed by using the classname or objectname.

syntax:

```
class_name.membername or class_name.
variablename
```

object_name.membername or object_name.variablename.

```
Class College:
    college_name = "Pyspiders"
    college_phone = 9988772455
    college_address = "HYD"
obj1 = college()
print(college.college_name, college.college_phone, college_address)
print(obj1.college_name, obj1.college_phone, obj1.college_address)
```

o/p :-

```
Pyspiders
9988772455
HYD
```

obj1.__dict__
{ }

```
obj1.name = 'sai'
obj1.phone = 9988665522
obj1.email = 'sai@gmail.com'
obj1.degree = 'Btech'
obj1.YOP = 2023
```

obj1.__dict__

{'name': 'sai', 'phone': 9966332211,
'email': 'sai@gmail.com', 'degree':
'Btech', 'yop': 2023}

Modification :-

obj1.phone = 9966332211

deletion :-

del obj1.yop

4-4-23.

class 0x11

key	value
col_name	"pys"
col_phone	996688
col_address	~~"Hyd"~~ "Bangalore"
website	True

object 0x22

key	value
col_address	"chennai"
col_phone	776677
estd	1901

Fetch the values by using Class address :-

(Static states) :-

>>> college.college_name

'Pyspiders'

>>> College.college_phone

9988774455

>>> College.college_address

"HyD"

Fetch the values by using object address :-

>>> obj1.College_name

'Pyspiders'

>>> obj1.college_Phone

9988774455

>>> obj1.College_address

"HYD"

Modifying the Static states by using Class address :-

>>> College.college_address = 'Bangalore'

>>> college.college_address

'Bangalore'

-> Modification will happen inside the class memory only but the modification will impact on class memory as well as object memory.

```
>>> obj1.College_address
'Bangalore'
```

Modifying the static states by using object address :-

```
>>> obj1.college_address = "Bangalore"
>>> obj1.college_address = "chennai"
"Chennai"
```

-> Modification will happen inside the object memory only and the modification will impact on only object memory.

```
>>> obj1.__dict__
{'college_address': 'Chennai'}
>>> College.__dict__
{'college_address': 'Bangalore'
.....}
```

deleting the static states by using Class address :-

```
>>> college.college_phone
    9988774455
>>> obj1.college_phone
    9988774455
>>> del college.college_phone
```

-> deletion operation will happen in class memory but it ~~will~~ impacts on both class memory and object memory.

deleting the static states by the help of object address

```
>>> college.college_name
    'pyspiders'
>>> obj1.college_name
    'pyspiders'
>>> del obj1.college_name
attributeerror : 'college' object has no attribute 'college_name'
```

Initializing Static States by using Class address :-

```
>>> College.website = True
>>> College.website
True
>>> obj1.website
True
```

By using object address :-
-> It is not possible to create static or initialization states by using object address.

-> Still if we try to initialize the states will be initialized inside the object memory, than that state is known as non static state.

2. Non-Static States :- (specific states) :-

-> These are the data that are specific for each and every object which is getting created in the object memory.

-> This states are un-common for the class and other object.

-> As this data is specific to an object so this will be stored inside the object memory itself.

-> how to store the data inside an object memory?

Syntax:

Objname.var_name = value

Ex: let us consider a college management application where we will have students as objects. Each student will have his own raw data like student name, age, phone, regno etc..

one more example: if we consider the bank application in the customer raw data like name, age, bal, pan, aadhar, all these properties are different for each and every customer.

example: Method 1:

```
Class Bank;
    B_name = "SBM"
    MBL    = "Delhi"
    LROI   = 13
Reeta = Bank()
Reeta.name = "Reeta",
```

```
Reeta.age = 27
Reeta.phno = 9123123123
Reeta.bal = 10000

print(Reeta.name, Reeta.age, Reeta.phno,
      Reeta.bal)
# these kinds of members are called
  object members or instance members.

class College:
    college_name = "pyspiders"
    College_phone = 9988774455
    College_address = "HYD"
ravi = college()
print(ravi.__dict__)
ravi.name = "ravi kumar"
ravi.roll = '14125a1201'
ravi.section = 'B'
ravi.phone = 9988774455
ravi.email = 'ravi@gmail.com'
print(ravi.__dict__)
```

o/p :-

```
{ }
{'name': 'ravi kumar', 'roll': '14125a1201',
 'section': 'B', 'phone': 9988774455, 'email':
 'ravi@gmail.com'}
```

fetch the non-static states by using object address :-

```
>>> ravi
<__main__.College object at 0x000>
>>> ravi.college_name
'pyspiders'
>>> ravi.name
'ravi kumar'
>>> ravi.roll
'14125a1201'
>>> ravi.email
'ravi@gmail.com'
>>> ravi.phone
9988774455
>>> ravi.section
'B'
>>> ravi.lover
attributeerror : 'college' object has
no attribute 'lover'
```

fetch the non-static states by using class address :-

->It is not possible to fetch non-static states by using class address : because

All the non-static states are stored inside the object memory.

Modify the non-static states using object address :-

>>> ravi.email = 'ravikumar@ghail.com'

>>> ravi.email

'ravikumar@ghail.com'

Modify the non-static states using Class address :-

-> It is not possible to modify the non-static states by using class address, still if we try to modify the values then it will initialize the state inside the class memory this state we call it as static state.

Delete the non-static states by using object address :-

>>> ravi.phone

9988774455

>>> del ravi.phone

>>> ravi.phone

attributeerror :

Delete the non static states by using class address :-

It is not possible to delete the non static states by using class address because non static states are stored in object memory.

Initialization :-

-> If we initialize the state by using class address ~~so~~ this ~~that~~ state we call as static state.

-> this state is initialized ~~by~~ in class memory.

-> If we initialize the state by using object address this state we call it as non-static state.

-> this state is initialized in object memory.

5-4-23

Method 2 :-

```
Class college :                    # class definition
    college_name = "pyspiders"
    college_phone = 9988774455
    college_address = "HYD"
# college_name, college_phone, college_address
ravi = College()  # object initialization / object creation by using class call,
                                    └> static states.
print(ravi.__dict__)  -> # display the object memory

def initialize(self):  # self argument holds the object address
    self.name = eval(input('enter the name'))
    self.roll = eval(input('enter the roll'))
    self.section = eval(input('enter the section'))
    self.phone = eval(input('enter the mobile'))
    self.email = eval(input('enter the email'))
                                    -># object attribute names
initialize(ravi)  # initializing the nonstatic states by using function.
print(ravi.__dict__)
```

o/p :-

```
{ }
enter the name = 'ravi'
 //     //  roll = '14125a1205'
 //     //  section = 'B'
 //     //  mobile = 9988774411
 //     //  email = 'ravi@gmail.com'
```

{'name' : 'ravi', 'roll' : '14125a1201',
'section' : 'B', 'phone' : 9988724411,
'email' : 'ravi@gmail.com'}

gopi = college()

gopi

<__main__.college object at
0x000>

gopi.__dict__

initialize (gopi)

enter the name gopi

roll 14125a1202

section b

mobile 9988774455

email 'gopi@gmail.com'

gopi.__dict__

{'name': 'gopi', 'roll': '14125a1202',
'section' : 'b', 'phone' : 9988724455,
'email' : 'gopi@gmail.com'}

-> class College :

    Col.name = 'pys'
    col. phone = 99677
    Col. address = 'HYD'

-> Ravi ~~college~~ ( )

    ~~0x71 ( )~~
    0x61

-> def initialize (self):

    self. name = input ("....")
    self. roll = input ("....")
    self. email = input ("....")

-> ~~initialize (Ravi)~~
0x51 (0x61)

College 0x71

0x71

Ravi

0x61

initialize

0x51

0x71 Class

key	value
Col.name	"pys"
Col.Phone	99677
Col. address	"HYD"

0x61 object.

key	value
name	"Ravi"
roll	"14125A1201"
email	"Ravi@gmail.com"

0x61

0x51

0x61	Ravi	
-> ~~self~~.name = ~~input ( )~~		
0x61	14125A1201	0x61
-> ~~self~~.roll = ~~input ( )~~		
0x61	Ravi@gmail.com	self
-> ~~self~~ email = ~~input ( )~~		

Method 3 :-

```
class college :
    college_name = 'pyspiders'
    college_phone = 9988274455
    college_addresss = 'HYD'
    def initialize(self):
        self.name = eval(input("enter the name:"))
        self.roll = eval(input("enter the roll:"))
        self.section = eval(input("enter the section:"))
        self.Phone = eval(input("enter the mobile:"))
        self.email = eval(input("enter the email:"))
obj1 = college()
college.initialize(obj1)
'gopi'
'14125a1203'
'b'
9663322255
'gopi@gmail.com'
obj1.__dict__
{'name': 'gopi', 'roll': '14125a1203',
 'section': 'b', 'phone': 9663322255,
 'email': 'gopi@gmail.com'}
```

College.initialize(obj1)

-> Method calling by using class address

note: we should assign the object inside the function call.

obj1: initialize() #-> class_address.method_address(object_address)

-> Method calling by using object address

note: we don't want to assign the object address inside the function call.

if we try to assign the object address, it raise the expection.

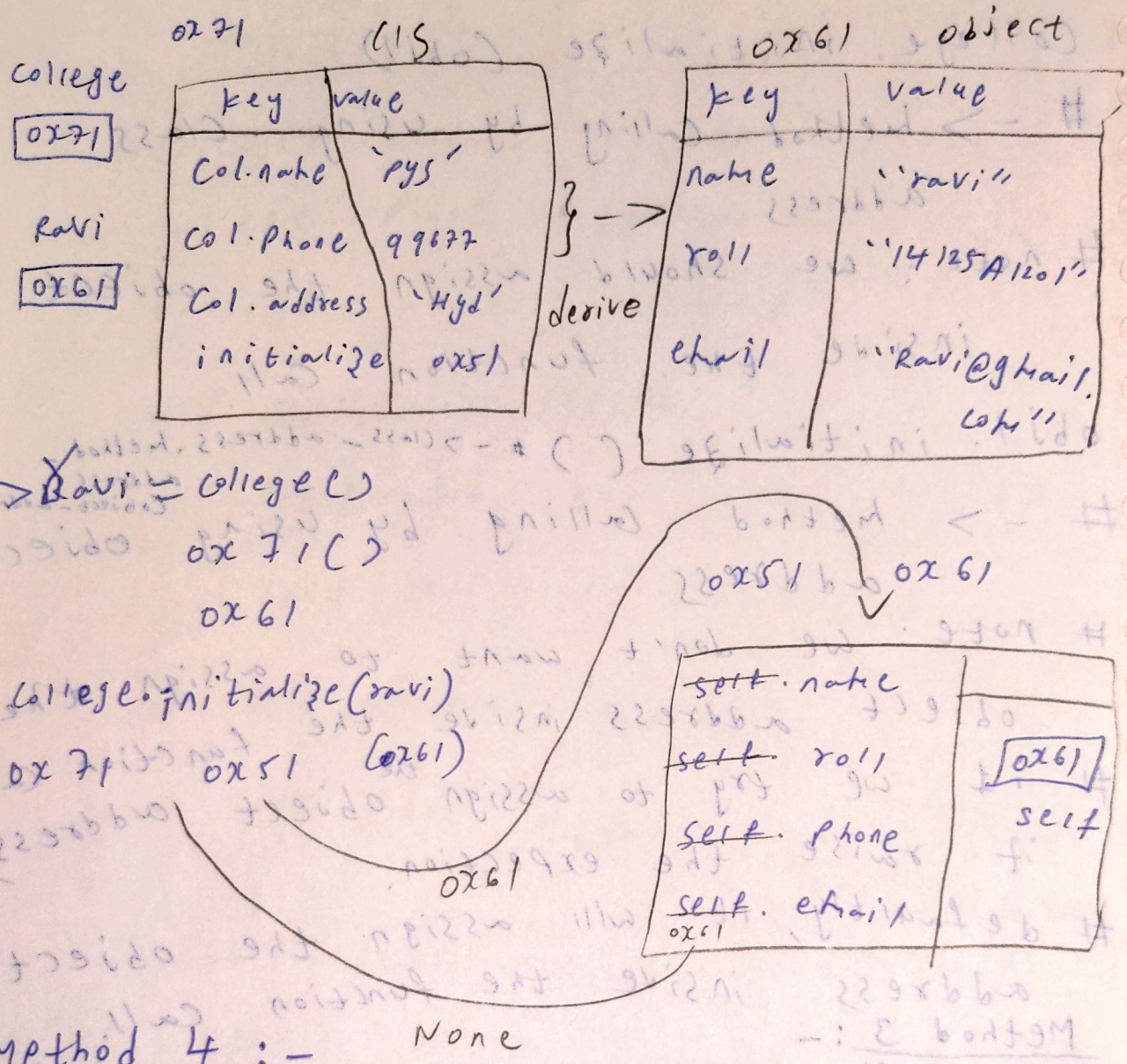
defaultly, it will assign the object address inside the function call.

Method 3:-

-> class College:

```
    Coll.name = "Pys"
    Coll.Phone = 99677
    Coll.address = "Hyd"
    def initialize(self):
        self.name = input("")
        self.roll = input("")
        self.email = input("")
```

(or) -> init -



Method 4 :-

```
class college :

    college_name = "pyspiders"
    College_Phone = 9988776455
    College_address = "Hyd"

    def __init__(self):
        self.name = eval(input(" "))
        self.roll = eval(input(" "))
        self.section = eval(input(" "))
        self.Phone = eval(input(" "))
```

self.email = eval(input(":: "))

obj1 = college()

o/p :- enter the details :-

"linga"

"125a1201"

"B"

"9966778811"

"linga@gmail.com"

Constructor or Initialization method (__init__):

-> In the world all the real-world entities will have both specific states and generic states to avoid initializing the specific states (non-static) separately, we came up with the concept of writing a special method that will do the initialization of specific states, this method we will call the initialization method or constructor.

-> we can write the specific states outside of the class and call whenever we wanted, but each time calling it will be a hectic task to overcome this problem we came up with an approach called constructor.

-> At the time of object creation itself how the application will get to know which method need called --init-- which is created inside the class and dedicated for the initialization of object members as because of this method application will get to know this method should be called at the time of object creation.

-> Constructor or initialization method is a type of method created to do the initialization of object members when we say to store the given details in an object information needs to give the object address in which data needs to be stored.

-> So this method must have at least one argument to store the address of an object in memory, according to industrial standards we call that argument with the name called self.

Syntax:

```
class classname:
    def __init__(self, arguments):
        self.var = val
obj = classname(args)
self.var = val
```

6-4-23.

example for Constructor method :-

```
Class College :
    college_name = 'pyspiders'
    college_address = 'hyd'
    def __init__(self, name, age, roll, Phone, email):
        # non static constructor method.
        print(self, name, age, roll, phone, email)
        self.name = name
        self.age = age
        self.roll = roll
        self.Phone = phone
        self.email = email
Obj = College('rama', 26, '12154210', 9988665577,
              'rama@gmail.com')
```

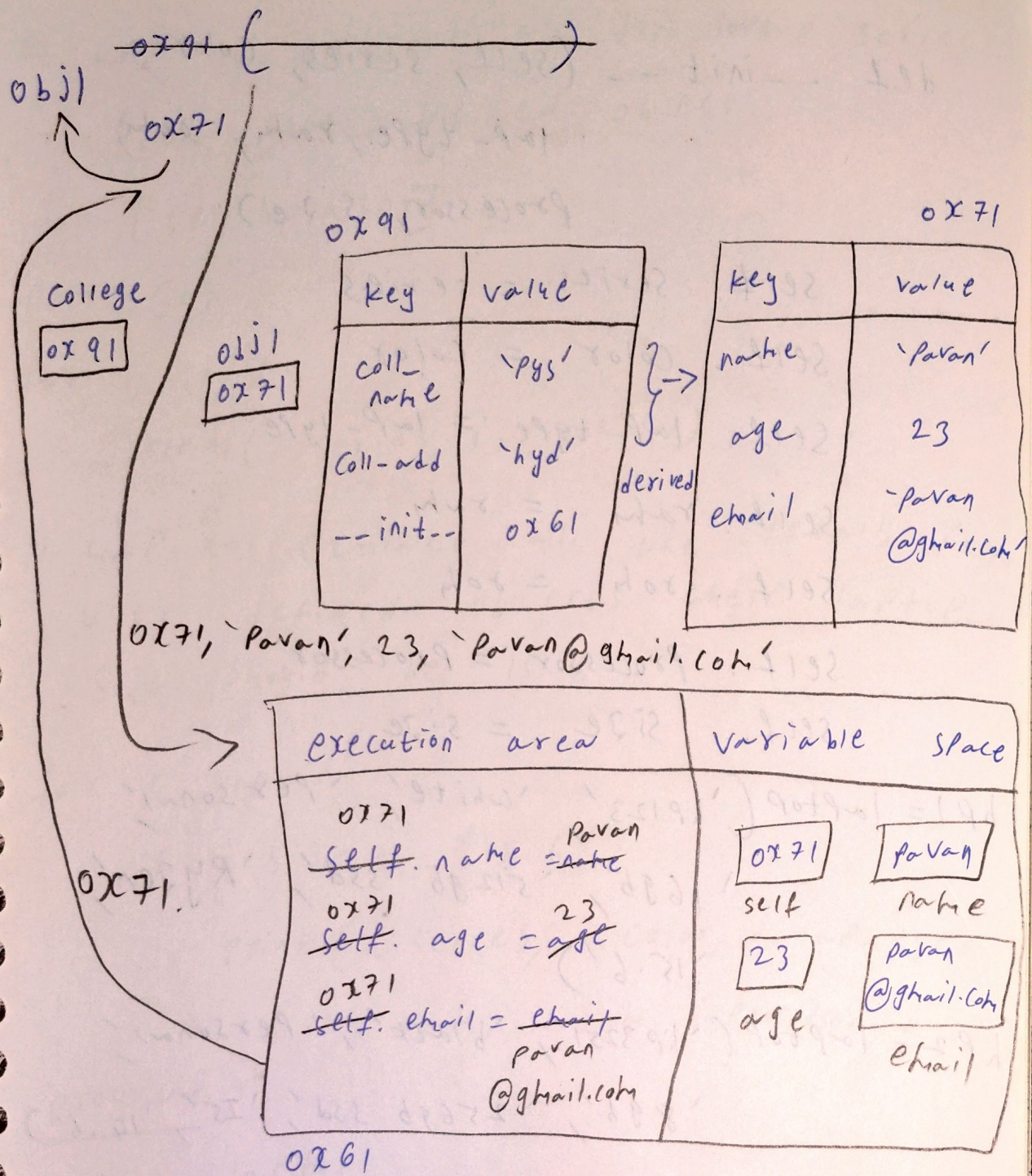
example :-

```
Class College:
    college_name = 'pyspiders'
    College_address = 'hyd'
    def __init__(self, name, age, roll, Phone, email):
        # non static constructor method.
        print(self, name, age, roll, Phone, email)
        self.name = name
        self.age = age
        self.roll = roll
        self.Phone = Phone
        self.email = email
```

```
def create(self, lovername):
    # non static initialization method
    self.lover = lovername

obj1 = college('rahu', 26, '12154210',
               9988665577, 'rahu@gmail.com')
obj1.address = 123   # default initialization

def add(self, new):
    # non static initialization function
    self.section = new
add(obj1, 'A')
obj1.create('sita')
```

memory management of constructor method :-

```
class college:
    coll_name = 'pys'
    coll_add = 'hyd'
    def __init__(self, name, age, email):
        self.name = name
        self.age = age
        self.email = email
obj1 = college('pavan', 23, 'pavan@gmail.com')
```



Program :-

```
Class laptop:
    Company_name = 'HP'
    Company-tollfree = 18001600401
    branch = 'hyd'
```

```
def __init__(self, series, color,
             lap_type, ram, rom,
             processor, size):
    self.series = series
    self.color = color
    self.lap_type = lap_type
    self.ram = ram
    self.rom = rom
    self.processor = processor
    self.size = size

hp1 = laptop('hp123', 'white', 'personal',
             '16gb', '512gb ssd', 'Ryzen',
             '15.6')

hp2 = laptop('hp3251', 'black', 'personal',
             '8gb', '256gb ssd', 'I5', '14.6')

hp3 = laptop('hp1562', 'pink', 'gaming', '16gb',
             '512gb ssd', 'Ryzen', '15.6')
```

WAP to extract all the laptop series from user defined objects.

```
l = [hp1, hp2, hp3]
print (l)
for i in l:
    print (i.series, i.processor)
```

wap to extract all the details from user defined objects where laptop size should be 15.6.

```
for i in l:
    if i.size == '15.6':
        print (i.series, i.color, i.lap-type,
               i.ram, i.rom, i.processor, i.size)
```

program :-

```
class laptop:
    company-name = "HP"
    company-tollfree = 18001600401
    branch = 'hyd'
    def --init-- (self, series, color,
                  lap-type, ram, rom,
                  processor, size):
```

```
        self. series = series
        self. color = color
        self. lap_type = lap-type
        self. ram = ram
        self. rom = rom
        self. processor = processor
        self. size = size

def display (self):
    print (`Laptop series_no:', self. series)
    print (`Laptop color :', self. color)
    print (`Laptop lap-type :', self. lap_type)
    print (` Laptop ram :', self. ram)
    print (` Laptop rom :', self. rom)
    print (` Laptop processor :', self. processor)
    print (` Laptop size :', self. size)
    return `haii'

hp1 = laptop (`hp123', `white', `personal', `16gb',
              `512gb ssd', `Ryzen', `15.6')

hp2 = laptop (`hp3251', `black', `personal', `8gb',
              `256gb ssd', `i5', `14.6')

hp3 = laptop (`hp1562', `pink', `gaming', `16gb',
              `512gb ssd', `Ryzen', `15.6')

print (hp1. display())   # laptop. display
                           (hp1)
```

7-4-23.

Behaviours or functionalities :-

-> These are the methods that are used to perform some operation such as stored ~~for~~ or manipulate the data (states) inside the class or object.

-> These are the functionalities that are done by objects or classes whenever it is needed. the functionalities are nothing but either create the states/ access the states/ modify the states/ delete the states.

-> These are the features that are given to the class or an object to perform some operation (behaviours) are written in the form of methods (functions).

-> There are three different types of methods that are there in this concept based on the data (states).

(A) object method

(B) class method

(C) static method.

Class Methods :-

-> This is one type of method that performs the operation on static states (class states)

-> It is a method type in which you can create, access, modify, delete the members (states/data) of class attributes.

-> Here we have to use argument name called cls to specify the address of the class memory in class method.

-> ex: if we look at the bank every day the rate of interest will change. This change should have impact on all its customers. In such a case we do the modification of the write class like Classname.variable.

syntax :

Classname.variablename

=new value.

-> If we wanted to do the same purpose multiple times then instead of using the previous approach, we use a method ~~the~~ called classmethod.

-> whenever you wanted to do the modification to use classmethod.

-> To make a classmethod, we need to decorate a method class @classmethod.

Syntax : (to write the class method)

```
@ classmethod
def method_name (cls, arguments):
    | any type of operation
    |
```

Note: arguments are not mandatory

Syntax : (to calling the class method)

```
Classname.method_name (arguments)
objectname.method_name (arguments)
```

-> In class methods whenever we do function call, first control will go to the @classmethod decorator

and then it goes to a method called @classmethod.

-> @classmethod will assign the class address to the given method. and internally it took a class address to assign a cls argument in the class method.

Program :-

```
Class employee:
    com_name = 'DELL'

    @classmethod
    def fetch(cls):
        print(cls)
        print(cls.com_name)

    @classmethod
    def modify(cls):
        cls.com_name
            =input("enter the
                company_name")
```

```
@classmethod
def delete(cls):
    del cls.com_name
@classmethod
def initialize(cls, new):
    cls.address = new
```

```
obj1 = employeeC()
# employee.fetch()
```

-> Calling the classmethod by using Class address.

obj1.fetch() # -> calling the Class method by using object address.

employee.modify() # -> modify the static states by using class address

obj1.modify() # -> modify the static states by using object address.

employee.deleteC() # -> delete the static states by using class address.

obj1.delete() # -> delete the static states by using object address.

employee.initialize('hyd') # -> initialize the static states by using class address.

obj1.initialize('hyd') # -> initialize the static states by using object address.

Object Method :-

-> This is one type of method that perforths the operation on non-static states (object states)

-> It is used to create, access, modify/ delete the members of object states.

-> This method utilizes the data of an object for all its operations.

Syntax: (define the object method)

```
def method_names (self, args):
    |
    | any type of operations
    |
```

Note: we are doing inside the class.

Syntax : (calling the object method)

```
object_name.method_name
        (arguments)
Class_name.method_name
        (object_name, arguments)
```

Note : arguments are not mandatory

Note : We are doing outside of the class.

```
Class employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def fetch(self):
        print(self.name)
        print(self.age)
    def modify_age(self):
        self.age = int(input("enter the age"))
    def delete_age(self):
        del self.age
```

```
def initialize (self):
    self.email = input("enter the email")

obj1 = employee ('chinnu', 23)

# employee.fetch (obj1) # -> fetching the non-static states by using Classaddress with object-address

# obj1.fetch () # -> fetching the non-static states by using object-address.

# employee.modify_age (obj1)
# obj1.modify-age ()

# employee.delete_age (obj1)
# obj1.delete_age ()

# employee.initialize (obj1)
# obj1.initialize ().
```

8-4-23.

Static methods:-

-> This is the type of method that is neither related with class states nor with object states.

-> This method is just written inside the class and can be used an additional functionality.

-> Ex: when you look at the ATM functionality when you want to withdraw money. It will ask the amount at the same time, if u want to deposit money the same machine will ask you for the amount. if you look at this both it asking you with the same message and same functionality is that receiving the amount value, so instead of writing 2-2 input statement, we create a one method called Static method only one item we written the method and n no of times, we can call it in anywhere inside the class,

-> we call it ~~whether~~ whenever we want to get the amount. The get amount functionality is not related to bank or

Customer but we are using a supporting method to reduce code redundancy.

-> To make a method as a static method we decorate a method with @ staticmethod

Syntax : (define a static method)

```
@ static method
def method_name(arguments):
```

Syntax : (calling a static method)

```
obj_name.method_name(arguments)
```

```
class employee:
    company_name = 'DELL'

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

```
@staticmethod
def upper(coll):
    st = ""
    for i in coll:
        if 'a' <= i <= 'z':
            st += chr(ord(i) - 32)
        else:
            st += i
    return st
obj1 = employee('kiran', 18)
print(employee.upper('pyspiders*12'))
print(obj1.upper('pyspiders*12'))
```

o/p :-

PYSPIDERS*12.

PYSPIDERS*12.

Program :-

```
class Bank:
    B_name = 'SBI'
    Branch = 'Mysore'
    Ifsc = 'SBIN00123'
    pincode = 521301
    B_address = 'GANDHI BAZAR Mysore'
    cust_count = 0
```

```
cust_details = { }

def __init__(self, name, phone, email,
             dob, adhaar, pan, address, bal=0):
    self.c_name = name
    self.c_phone = phone
    self.c_email = email
    self.c_dob = dob
    self.c_adhaar = adhaar
    self.c_pan = pan
    self.c_address = address
    self.bal = bal
    self.cust_count_inc()
    self.c_id = str(123) + str(self.cust
                               _count)
    self.add_cust_details(self.c_id, self)

@classmethod
def cust_count_inc(cls):
    cls.cust_countt += 1

@classmethod
def add_cust_details(cls, custid, self):
    cls.cust_details[custid] = self
```

```
def deposit(self):
    amount = int(input("enter the deposit
                        amount:"))
    self.bal += amount
    print("your transcation is success")
    print("MR/MS:", self.C_name,
          "\n your account balance is"
          self.bal)

def withdraw(self):
    amount = int(input("enter the
                        withdraw amount:"))
    if self.bal > amount:
        self.bal -= amount
        print("your transaction is
                success")
        print("MR/MS:", self.C-name,
              "\n your account balance is"
              self.bal)
    else:
        print("your transaction is declined")
        print("MR/MS:", self.C-name,
              "\n your account balance
                is", self.bal)
```

```
        print ("insufficient balance")

    def changePhone(self):
        print ("Mr/Ms: ", self.c_name,
               "\n your mobile number
                is ", self.c_phone)
        num = int(input("enter the old mobile number:"))
        if self.c_phone == num:
            num = int(input("enter the new
                  mobile number:"))
            num1 = int(input("re-enter the
                   new mobile number:"))
            if num == num1:
                self.c_phone = num
                print("your transaction
                       is success")
                print("Mr/Ms:", self.c_name,
                      "\n your new mobile
                       number:", self.c_phone)
            else:
                print("your transaction
                       is declined")
```

```
else:
    Print("your transaction is
          declined")

Obj1 = Bank('RAMA', 9988774455, 'rama@
       gmail.com', '08-06-1999',
       1234567899, 'abcd125an',
       'mysore', 1000)
Obj2 = Bank('SEETHA', 9988774433,
       'SEETHA@gmail.com', '08-06-2000',
       1234567899, 'abcd128an',
       'mysore')
```

10-4-23.

Inheritance :-

→ It is a phenomenon of deriving or acquiring the properties (functionalities) of one class to another class as known as inheritance.

→ we are going for inheritance to avoid the code redundancy problem because code redundancy will increase the turnaround time in programming.

-> When we do the inheritance, all the members that are stored inside the Parent Class will be derived to the child class to which we are calling it as inheriting.

-> The class from which we derive the properties are called parent class or base class or super class.

-> the class to which the properties derive are called child class or derived class or subclass.

-> In heritance concept deriving the properties from parent class to child class as a same time adding extra new features to child class.

How to write the parent and child class?

Syntax :-

```
Class Parent Class: -> parent
                       Class
    |
    | states/Behaviours
    |
    |
```

class Child_Class (Parent_Class1, Parent_Class2):

States/Behaviours

Child Class.

The process of inheritance will happen in following steps:

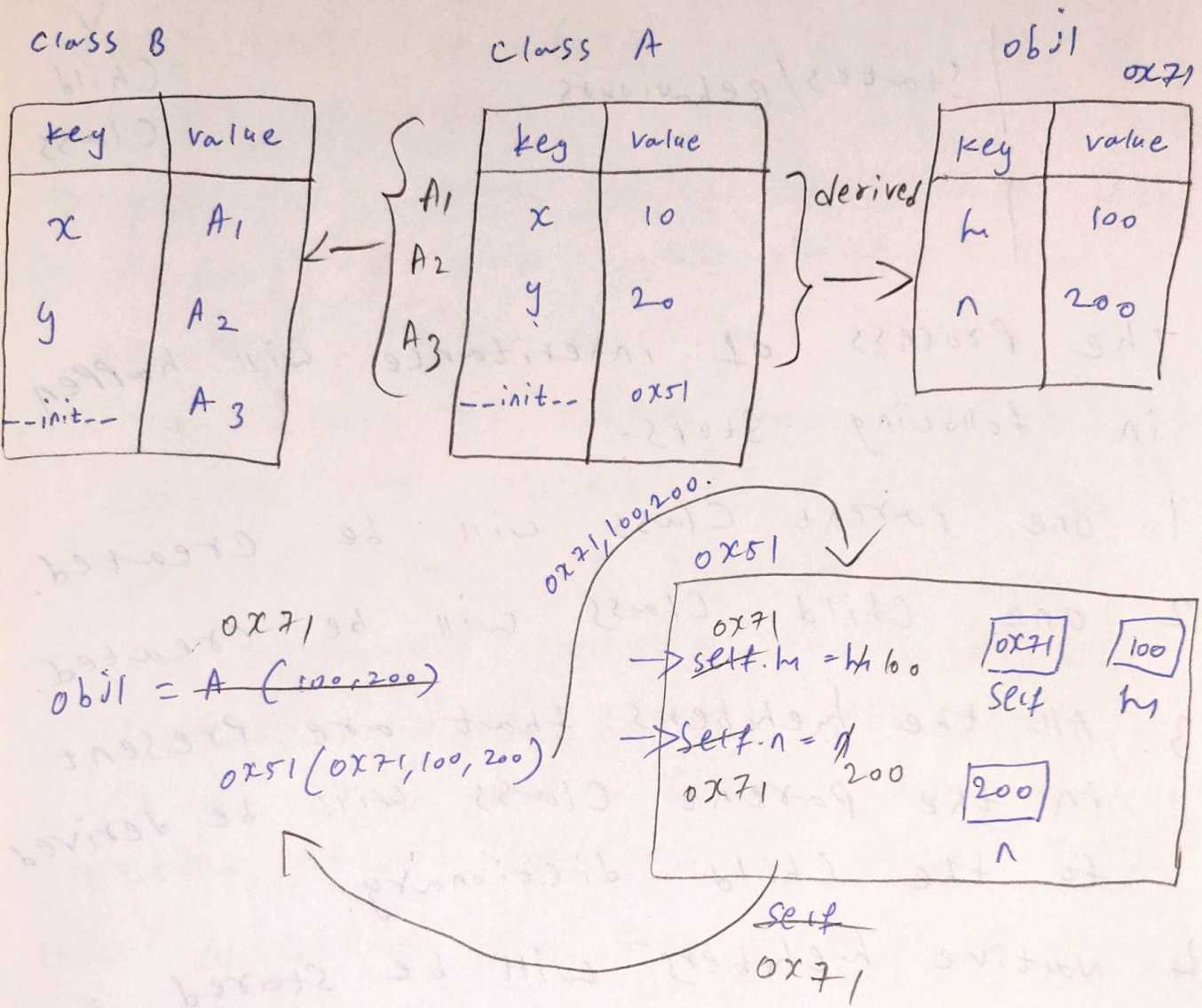
1. one parent Class will be created.
2. one Child Class will be created.
3. All the members that are present in the parent Class will be derived to the Child dictionary.
4. Native members will be stored in Child Class by the help of inheritance.

example :-

```
Class A:    # Parent Class
    x = 10
    y = 20
    def __init__(self, m, n):   # child
        self.m = m
        self.n = n
```

Class B(A): # Child Class.

z = 30



Creating the Parent members and Child class members :-

1. if we want to create members for the parent class, we should create through parent Class names only.
2. If we want to create members for the child class, we should create through Child class names only.

3. The Child class members we cannot create through the parent class names.
4. The parent class members we cannot create through the Child class names.

Accessing the parent members and Child class members:

1. The parent class members we can access through the parent class name as well as the child class name.
2. The child class members we can access through the Child class name only.
3. The Child class members we cannot access through the parent class name.

Modification the parent members and Child class members :-

1. The Parent class members we can only modify through the Parent class name only. (If we try to modify it through the parent class name. It will impact on Parent class dictionary and also all the derived classes dictionary or derived objects dictionary)

Note :- If we try to modify the Parent class members by using the Child class name, it will impact the Child class dictionary only. It does not impact the parent class dictionary or any other class object dictionaries.

2. The Child class members we can modify through the Child class name only. (if in case we are doing this process it will impact the same class dictionary and derived child object dictionary).

3. If we try to modify it through the Parent class name. It will impact the parent class dictionary and also all the derived classes dict. or derived objects dict. The Child class members we cannot modify through the parent class name.

Deletion the parent members and Child class members:

1. The parent class members we can delete through the parent class name only. (if we try to delete the parents class members by through parent class name, it will impact parent class dictionary and all the derived classes dictionary and derived objects dictionary)

Note: The parent class members we cannot delete through the child class name. If in case we are doing it shows attribute error.

2. The Child class members we can delete through the child class name only. (if we try to delete the child class members through the child class name. it will impact the same class dict. and derived object dictionary).

Note:- The child class members we cannot delete through the parent class name. if in case we are doing it shows attribute error.

11-4-23

Fetch the values by using Parent class address

>>> A.X
10

>>> A.Y
20

>>> B.X
10

>>> B.Y
20

Fetch the entire Class dict block's

>>> A.__dict__

>>> B.__dict__

A.X = 100

>>> B.X
100

100 ↳ modification using Parent Class address.

A.__dict__

>>> B.Y - Fetch the values by using Child Class address
20

>>> B.Y = 200 - modification using Child Class address

del B.X

attributeerror : X

del A.X - delete Parent Class attribute by using Parent Class address.

>>> A.x

attribute error.

>>> B.x

attribute error.

A.m = 1000 - intialize the Static States by using parent class address.

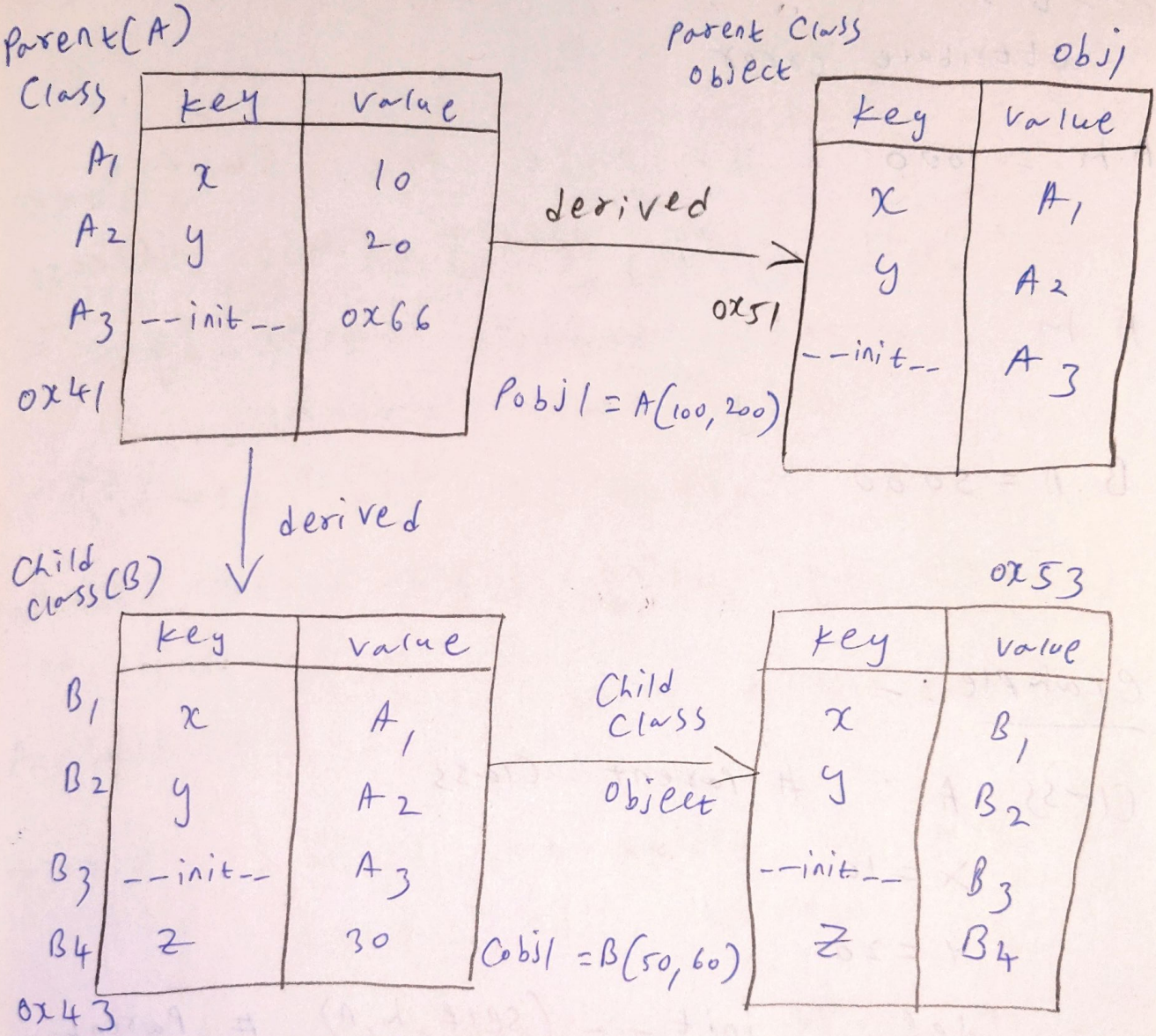
A.m -> Fetch static states by using parent class address

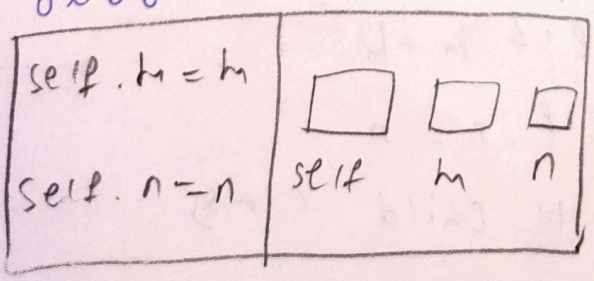
B.n = 5000 - intialize the static states by using child class address.

Example :-

```
class A :  # parent class
    x = 10
    y = 20
    def __init__(self, m, n):  # Parent Class __init__ method
        self.m = m
        self.n = n
class B(A):  # child class
    z = 30
Pobj1 = A("hai", "bye")
Cobj1 = B([10,20], [30,40])
```

```
>>> Pobj1.__dict__
{'m': 'hai', 'n': 'bye'}
```





Value Space
A = 0x41
Pobj1 = 0x51
B = 0x43
Cobj1 = 0x53

example :- Constructor overriding

```
class A:  # Parent Class
    x = 10
    y = 20
    def __init__(self, m, n):
        print("parent class constructor")
        self.m = m
        self.n = n
class B(A):  # Child Class
    z = 30
    def __init__(self, a, b):
        print("child class constructor")
        self.a = a
        self.b = b
pobj = A('hai', 'bye')
cobj = B([10,20], [30,40])
```

o/p :-

parent class constructor

child class constructor

Parent (A)
Class

	key	value
A_1	x	10
A_2	y	20
A_3	--init--	0x66

0x41

derived

Parent class object Obj1

key	Value
x	A_1
y	A_2
--init--	A_3

0x51

pObj1 = A(100, 200)

Child class (B)

	key	value
B_1	x	A_1
B_2	y	A_2
B_3	--init--	B_0
B_4	z	30

0x43

Child class object

0x53

key	Value
x	B_1
y	B_2
--init--	B_3
z	B_4

Cobj1 = B(50, 60)

0x66 -> Parent Class Constructor

self.m = m self.n = n	self m n

Value space	
A =	0x41
Pobj1 =	0x51
B =	0x43
Cobj1 =	0x53

0x76 -> Child Class Constructor

self.a = a self.b = b	self a b

Constructor Chaining

```
Class A:
    x = 10
    y = 20
    def __init__(self, m, n):
        print("parent class constructor=")
        self.m = m
        self.n = n
Class B(A):
    z = 30
    def __init__(self, a, b):
        print("child class constructor=")
        A.__init__(self, 500, 600)
        # child class constructor is calling
        to parent class constructor
        # this concept we will call it as
        constructor chaining.
        self.a = a
        self.b = b
Pobj1 = A("hai", "bye")
Cobj1 = B([10,20], [30,40])
```

12-4-23.

Program :- Question

Register the student details;

1. Student name 2. Ph.no 3. email.id

4. branch. 5. password 6. re-enter password.

Validation :- 1. user name should be uppercase. (by using static method)

2. mobile number starts with either 9 or 8 or 7 or 6 and mobile number should be 10 digits. (by using static method).

3. email.id should be more than ~~one~~ 10 charac. and only one @ should be present and min one dot(.).

4. ~~It~~ Branch should be uppercase.

5. password should be min five charac. min one special symbol, min one asci; number and 1st character should be uppercase.

-> All the constraints are satisfy then only store the data.

Student login operation :-

→ if password is match display the user details. (except password).

→ if password is not matched provide attempts.

Program :-

```
Class Student :
    C_name = "sree vidyanikethan"
    C_address = "Tirupath"
    C_Phone = 9933117755
    C_Code = "SVEC"
    S_count = 0

    def __init__(self):
        self.S_name = input("enter the
                             student Name")
        self.S_Phone = input("enter the
                              Student Phone")
        self.S_gender = input("enter the
                               Student gender")
```

```
self.s_dob = input("enter the student
                    dob")
self.s_section = input("enter the student
                        section")
self.s_class = input("enter the student
                      class")
self.s_count = inc()
self.s_id = self.c_code +
            str(self.s_count)

@classmethod
def s_count_inc(cls):
    cls.s_count += 1

@classmethod
def change_c_phone(cls):
    if cls.c_phone == int(input
                          ("enter the old
                           mobile number")):
        m = int(input("enter the new
                       mobile number"))
        n = int(input("reenter the new
                       mobile number"))
        if m == n:
            cls.c_phone = m
```

```
        print ("mobile number sucesstfully
                updated")
    else:
        print ("mobile number and
                reenter mobile number should
                be same")
        CLS.Change_C_Phone()
else:
    print ("mobile number is not
            matched")
    CLS.Change_C_Phone()

def Change_S_Phone(self, count=0):
    if count == 3:
        print ("exceeded number of
                3 attempts")
        return
    if int(self.S_Phone) == int(input
            ("enter the old mobile
            number"))
        m = int(input("enter the new
                mobile number"))
        n = int(input("re enter the new
                mobile number"))
```

```
        if m==n:
            self.S_Phone=str(m)
            print("mobile number
                  successfully updated")
        else:
            count+=1
            print(count,"attempt is done")
            print("mobile no and reenter
                  no should be same")
            self.Change_S_Phone(count)
else:
    count+=1
    print(count,"attempt is done")
    print("mobile no is not matched")
    self.Change_S_Phone(count)
```

example :-

```
class Student:
    C_name="SVEC"
```

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

@staticmethod
def upper(coll):
    st = ""
    for i in coll:
        if 'a' <= i <= 'z':
            st += chr(ord(i) - 32)
        else:
            st += i
    return st

obj = student('sai', 15).
```

17-4-23.

Constructor Chaining :-

-> It is a phenomenon invoking the constructor from parent class to the constructor of child class is called constructor chaining.

-> Invoking from one constructor to another constructor is known as constructor Chaining.

-> by through constructor chaining to reduce code redundancy and eliminate repetitive task.

-> There are 3 ways to call to achieve the constructor chaining.

1. Calling by using the super statement.

Syntax :-

super().__init__(args)

-> can be used in single, multilevel and hierarchical inheritance.

Note : super() refers to parent class data/address.

2. Calling by using super containing arguments.

Syntax :-

super(ChildClassName, self).__init__(arguments)

-> Can be used in single and multilevel inheritance.

3. Calling by using Parent Class name

Syntax :-

Parent.__init__(self, arguments)

-> Can be used in any type of inheritance.

Program :-

```
Class sample:
    x=100
    def __init__(self,i,j): # parent class constructor
        self.i=i
        self.j=j
Class demo(sample):
    y=200
    def __init__(self,i,j,k,l): # child class constructor
        # super().__init__(i,j)
        # super(demo,self).__init__(i,j)
        sample.__init__(self,i,j) : # invoking to child constructor
        self.k=k
        self.l=l
obj2=demo(10,20,30,40)
```

Method Chaining :-

-> It is the phenomenon of calling a parent class method into a child class method to avoid code redundancy problems.

-> There are 3 ways to call to achieve the method chaining.

1. Calling by using the super statement.

Syntax :-

```
super().methodname(args)
```

Note : super refers to parent class data / address

2. Calling by using super containing arguments

Syntax :-

```
super(childClassName, self).
    methodname(arguments)
```

3. Calling by using Parent class name

Syntax :-

```
parent.methodname(self, arguments)
```

Program :-

```
class sample:
    x = 100
    def __init__(self, i, j):
        self.i = i
        self.j = j
    def display(self):
        print("i value:", self.i)
        print("j value:", self.j)
```

```
class demo (sample):
    y=200
    def __init__(self, i, j, k, l):
        super(demo, self).__init__(i, j)
        self.k = k
        self.l = l

    def display(self):
        # sample.display(self)
        # super().display()
        super(demo, self).display()
        print("k value:", self.k)
        print("l value:", self.l)
obj2 = demo(10, 20, 30, 40)
```

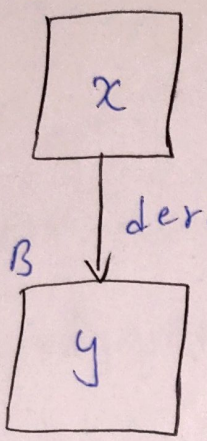
~~Program~~ :-

Inheritance :-

There are five types of inheritance.

1. single level inheritance.
2. multi level inheritance.
3. ~~Hei~~ Hierarchical inheritance.
4. multiple inheritance.
5. Hybrid inheritance.

single level Inheritance :- deriving the properties from one class to another class, only one parent class and only one A child class.



parent class / super class / Base class x members

child class / sub class / derived class

x+y members.

x+y members.

example for single level inheritance, Constructor Chaining, Method Chaining :-

```
Class car :
    Company = "BMW"
    def __init__(self, model, price, Color):
        self.C_model = model
        self.C_price = price
        self.C_Color = Color.
```

```
def display(self):
    print("car model number:", self.c_model)
    print("car price:", self.c_price)
    print("car color:", self.c_color)

obj1 = car('BMW01', 19999999, 'white')

class car1(car):
    branch = "HYD"
    def __init__(self, model, price, color,
                 ctype, fuel, capacity):
        car.__init__(self, model, price, color)
        self.c_ctype = ctype
        self.c_fuel = fuel
        self.c_capacity = capacity
    def display(self):
        car.display(self)
        print("car type:", self.c_ctype)
        print("car fuel:", self.c_fuel)
        print("car capacity:", self.c_capacity)

obj2 = car1('bmw2', 75999999, 'black',
            'SUV', 'petrol', '4')
```

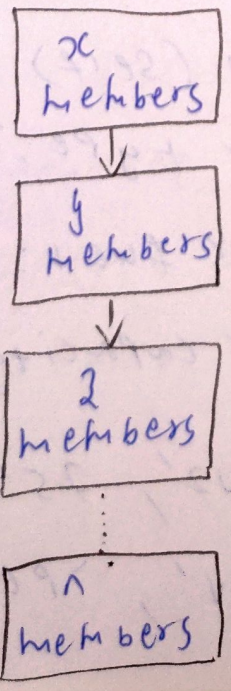
18-4-23.

Multi level inheritance :- Deriving the properties parent class to child class and Child class to Sub-Child class.

-> In Sub-Child class it have parent class members + child class members + Sub-Child class members.

-> In multi-level inheritance, the features of the baseclass and derivedclass members inherited into the new derived class, this mechanism is similar to the relationship grandfather and Child.

-> It is used for creating new versions update the new functionality/ new features to the existing application.



Syntax :-

Parent, base, super

```
class Parent:
    pass
```

x

Child, derived, sub

```
class Child(Parent):
    pass
```

y => x+y

Child child, subderived, sub sub class

```
class SubChild(child):
    pass
```

z => x+y+z

example :-

```
class Parent:
    x = 10
    def __init__(self):
        self.m = 'super'

class Child(Parent):
    y = 20
    def __init__(self):
        self.n = 'derived'

class SubChild(child):
    z = 30
    def __init__(self):
        self.o = "sub derived"
```

Parent

>>> <class '__main__.Parent'>

Child

<class '__main__.Child'>

subChild

<class '__main__.subChild'>

parent.x

10

Child.x

10

subChild.x

10

>>> Child.y

AttributeError

Child.y

2228

subChild.y

20

subChild.z

30

>>> Parent.z

AttributeError

-> subChild class attributes don't derive to Child class and Parent class.

obj = Parent()

obj.x

10

>>> obj1.x

10

obj.n

'derived'

>>> obj1.h

attribute error.

obj2 = subChild()

obj2.x

10

obj2.o

'sub derived'

>>> obj2.h

attribute error

obj.h

super

obj.y

20

>>> obj2.y

20

>>> obj2.n

attribute error

-> Parent Class object attributes don't derive to Child Class object and sub Child Class objects and vice versa.

Parent Class Modification.

```
>>> Parent.x = 180
```

Parent.x	Child.x	Sub Child.x
180	180	180
obj.x	obj1.x	obj2.x
180	180	180

Parent Class attribute initialize.

```
>>> Parent.a=150
```

>>> Parent.a	Child.a	Subchild.a
150	150	150
obj.a	obj1.a	obj2.a
150	150	180

Parent Class attribute delete.

```
>>> del Parent.a
>>> Parent.a
  attribute error

>>> Child.a
   attribute error

>>> subChild.a
    attribut error
```

Child Class attribute Modification.

Child.y = 250

Child.y → 250

obj1.y → 250

SubChild.y → 250

Parent.y → attribute error.

obj.y → attribute error.

Chid Class attribute initialize

Child.b = 60

>>> Child.b
60

>>> obj1.b
60

>>> SubChild.b
60

>>> obj2.b
60

>>> Parent.b
attribute error.

obj.b
attribute error.

Child class attribute delete.

>>> del child.b

child.b
attribute error

subChild.b
attribute error.

SubChild class attribute initialize.

>>> subChild.C = 190

SubChild.C

190

>>> obj2.C

190

>>> Child.C

attribute error:

>>> Parent.C

attribute error:

>>> obj.C

attribute error

>>> obj1.C

attribute error

SubChild class <- attribute delete

>>> del subChild.C

>>> subChid.C

attribute error

>>> Parent.C

attribute error

>>> Child.C

attribute error.

>>> obj2.C

attribute error

>>> obj.C

attribute error

Parent Class object initialize

>>> obj.i = 20

attribute error.

Parent.i

obj. i

attributeerror attribute error.

Parent Class object delete

del obj.i

>>>obj. i >>>parent. i

attribute error attribute error.

Parent Class object read

>>> obj.x del obj.x

>>> obj.x -> (180 is derived from class to object).
180

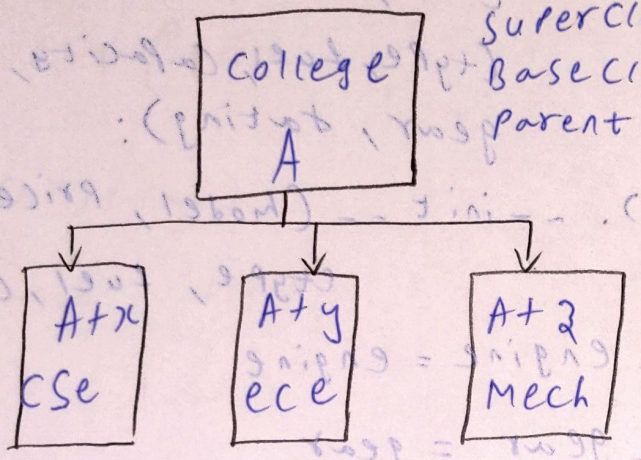
19-4-23.

```
Class (Car):
    Company = 'BMW'
    def __init__(self, model, price, Color):
        self.C_model = model
        self.c_price = price
        self.C_color = color
    print ("Car__init__method")
```

```
    def display(self):
        print("car model number:"
              self.c_model)
        print("car price:", self.c_price)
        print("car color:", self.c_color)
obj1 = car('BMW01', 19999999, 'white')

class car1(car):
    branch = "HYD"
    def __init__(self, model, price, color
                 ctype, fuel, capacity):
        super().__init__(self, model, price,
                         color)
        self.c_ctype = ctype
        self.c_fuel = fuel
        self.c_capacity = capacity
        print("car1__init__method")

    def display(self):
        car.display(self)
        print("car type:", self.c_ctype)
        print("car fuel:", self.c_fuel)
        print("car capacity:", self.c_capacity)
```

```
obj2 = car1('bmw2', 759999, 'black', 'suv',
                'petrol', '4')

class car2(car1):
    tollno = 18001000 5
    def __init__(self, model, price, color,
                 type, fuel, capacity, engine,
                 gear, rating):
        super().__init__(model, price, color,
                         type, fuel, capacity)
        self.c_engine = engine
        self.c_gear = gear
        self.c_rating = rating
        print("car2 __init__ method")
    def display(self):
        super().display()
        print("car engine:", self.c_engine)
        print("car gear:", self.c_gear)
        print("car rating:", self.c_rating)

obj3 = car2('bmw3', 1592090, 'red', 'suv',
            'petrol', 4, 'auto', 'auto', 5)
```

Hierarchical inheritance :- Deriving the properties from parent class to more than one child class is called hierarchical inheritance.



-> parent class of 'A' members.

-> first child class have 'A' members with 'x' members.

-> second child class have 'A' members with 'y' members.
and so on.

-> If we manipulate anything in parent class, it will impact on parent class, associate parent class objects and inherit child classes and child class objects.

-> If we manipulate anything inside Child_Class, it will impact on Child Class with associate Child class object but it will not impact on parent class and parent class objects.

Syntax

```
class Parent:
    pass

Class Child1(Parent):
    pass

Class Child2(Parent):
    pass
```

example:-

```
Class google:
    account = "GOOGLE"
    def __init__(self, first, email, phone, dob):
        self.firstname = first
        self.email = email
        self.phone = phone
        self.dob = dob
```

```
obj = google('murali', 'murali@gmail.com',
                9988556622, '06-09-1999')
class facebook(google):
    app = 'META'
    account = 'Facebook'
    def __init__(self, first, last, email,
                 phone, dob, gender, status):
        super().__init__(first, email, phone,
                         dob)
        self.lastname = last
        self.gender = gender
        self.status = status
obj1 = facebook('rahul', 'singh',
        'rahul@gmail.com', 9977884455,
        '06-08-1998', 'male', 'un-married')

class youtube(google):
    app = 'YOUTUBE'
    def __init__(self, first, last, email,
                 phone, dob, gender, status,
                 acc_type):
        super().__init__(first, email,
                         phone, dob)
```

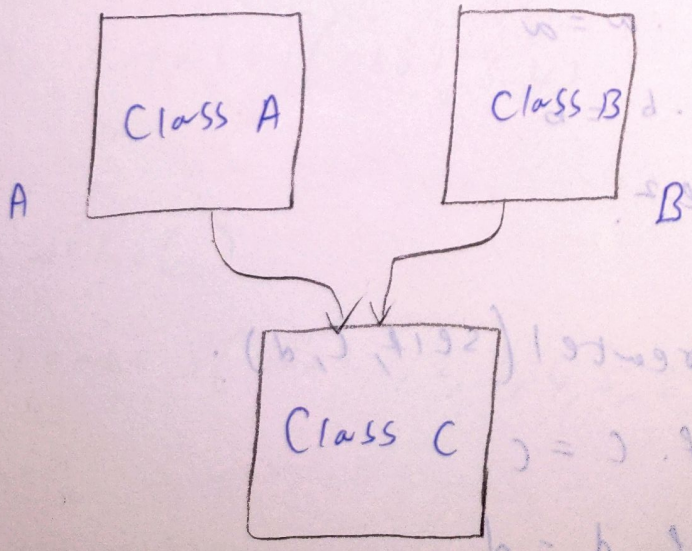
```
        self.lastname = last
        self.gender = gender
        self.status = status
        self.acc_type = acc_type
obj2 = youtube('rani', 'kumari', 'rani@gmail.
                                    com',
                9988776633, '09-09-1992',
                'female', 'single', 'free')
```

20-4-23.

Multiple inheritance :- Deriving the properties more than one parent class to child class.

-> Child class contains all the parent class properties.

-> All the parent class properties are derived to child class.



A+B+C members.

Syntax :-

```
Class Parent 1 :
    Pass
Class Parent 2 :
    Pass
Class Child (Parent 1, Parent 2):
    Pass
```

Multiple inheritance without overriding method

```
Class Sample1 :
    x = 10
    def create (self, a, b):
        self.a = a
        self.b = b
Class Sample2:
    y = 20
    def create1 (self, c, d):
        self.c = c
        self.d = d
```

```
class demo(sample1, sample2):
    }=30
    def create3(self, a, b, c, d, e, f):
        sample1.create(self, a, b)
        sample2.create1(self, c, d)
        self.e = e
        self.f = f
obj = sample1()
# sample1.create(obj, 1, 2)  # Creating non static states using class address
# obj.create(1,2)  # Creating non static states using object address.

obj1 = sample2()
sample2.create1(obj1, 3, 4)

obj2 = demo()
demo.create3(obj2, 1, 2, 3, 4, 5, 6)
```

userdefine constructor

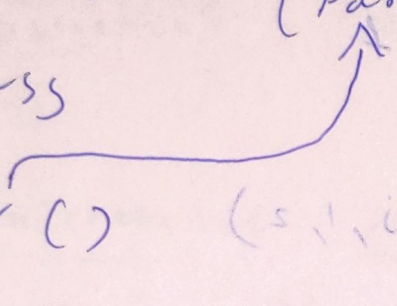
userdefine constructor Chaining

-> the super() this function is calling only for one parent { Class address.

-> In multiple inheritance if we use super() method it is pointing to the first parent class.

Class classname (Parent 1, Parent 2):
    pass

super() (Higher priority)



program :-

case 1 :-

```
Class Parent1:
    apple = 10
    banana = 20
    def __init__(self, Custardapple, drangonfruit):
        self.Custardapple = Custardapple
        self.drangonfruit = drangonfruit

Class Parent2:
    orange=50
    pineapple =60
    def __init__(self, Watershelon, grapes):
        self.Watershelon = Watershelon
        self.grapes = grapes
```

```
class Child(Parent2, Parent1):
    mango = 100
    def __init__(self, custardapple,
                 dragonfruit, waterhelon, grapes,
                 papaya, kiwi):
        Parent1.__init__(self, custardapple
                         dragonfruit)
        Parent2.__init__(self, waterhelon, grapes)
        self.papaya = papaya
        self.kiwi = kiwi
obj = Child(100, 200, 300, 400, 500, 600)
```

O/P :-

```
obj.__dict__
{'custardapple': 100, 'dragonfruit': 200,
 'waterhelon': 300, 'grapes': 400,
 'papaya': 500, 'kiwi': 600}
```

Case 2 :-

```
class Child(parent1, parent2):
    mango = 100
obj = Child(100, 200)
```

O/P :-

```
obj.__dict__
{'custardapple': 100, 'dragonfruit': 200}
```

Case 3 :-

```
Class child (Parent 2, Parent 1):
    mango = 100
obj = Child (100, 200)
```

o/p :-

```
obj.__dict__
```

{'watermelon': 100, 'grapes': 200}.

Case 4 :-

```
Class Child (Parent 2, Parent1)
    mango = 100
    def __init__(self, a, b):
        Parent 1.__init__(self, a, b)
obj = Child (100, 200)
```

here parent classes priority doesn't matter in init method. Because, we are calling init method with the help of Parent class names.

Program :-

```
Class Parent1:
    def display(self):
        print("parent Class 1")

Class Parent2:
    def display(self):
        print("Parent Class 2")

Class Child(Parent2, Parent1):
    def display(self):
        print("Child Class")

obj = Child().
```

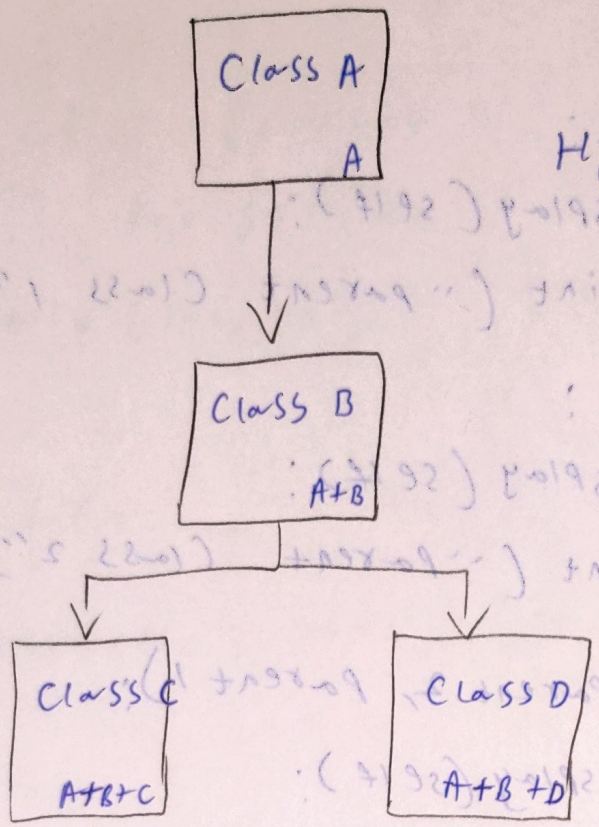
O/P :-

```
obj.display()
>>> child Class.
```

21-4-23.

Hybrid inheritance :- The Combination of more than one inheritance is called hybrid inheritance.

Hybrid.



Syntax :- (example)

```
Class A:
    pass

Class B(A):
    pass

Class C(B):
    pass

Class D(B):
    pass
```