Object Oriented Programming (OOPs)

class:

class is a structure which will tell the user how to store the data and how to utilize the stored data

(or)

class is a blue print which consist of properties and functionalities (or) states and behaviours of a real time Entite or real time object.

Object:

Object is an instance of class (or) It is a variable which is created for a Particular class.

class syntax:

```
class cname :
    properties/states
    functionalities/behaviours
```

(1 tab)

Object syntax:

```
objname = cname(args)
```

where passing the arguments is not mandatory

steps to create class and Object inside the memory:

class creation:

1. As soon as control see a keyword called class it will create a dictionary inside a memory, which consist of key and value address will be given and that will stored into class name.

2. All the properties and functionalities class will get stored into class dictionary, reference address will be given.

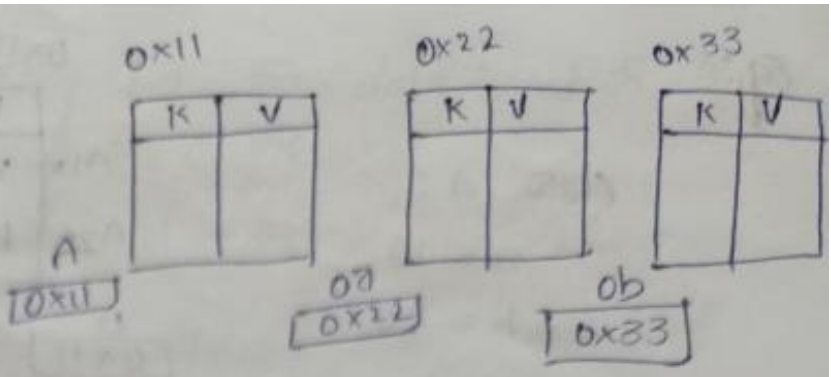
Object creation:

1. As soon as control see an Object creation process it is going to create object dictionary, which consist of key value pair, address will be given and will get stored into Object name

2. All the members of class will get stored into object dictionary then control will store its own members or Objects numbers into object dictionary (This process will happen only if the Object is having '__init__' method)

Eg:

```
class A :
    Pass
oa = A()
Print (type (oa))
ob = A()
Print (type (ob))
```



O/P: <class '__main__.A>
<class '__main__.A>

→ If the object can't contains the __init__ method in the object class then it will ignore the instruction in the object class

→ Once the object is created with class name it will store the members of instructions in the class if the instructions is not their then it will goes the own object class area

The syntax to access the members of class and object

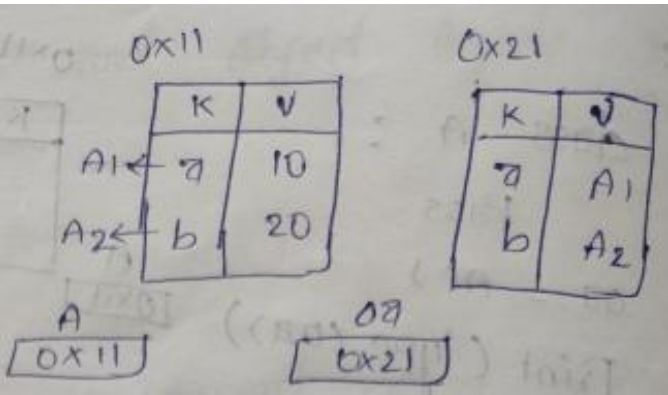
For class : cname . mname
For obj : objname . mname

whenever we want to modify the value in the class/ obj we will make use of syntax is

For class : cname . mname = new_value
For obj : objname . mname = new_value

Eg:

```
class A:
    a = 10
    b = 20
oa = A()
Print (A.a, A.b)
Print (oa.a, oa.b)
```



O/P: 10 20
10 20

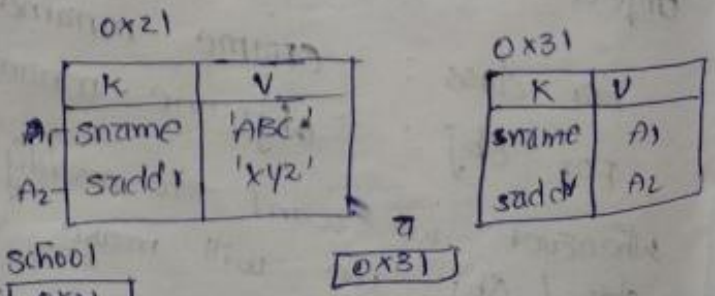
A1, A2 are the reference address given for the variable and that will stored as value for the object dictionary

Eg:

```
class A
    a = 10
    b = 20
oa = A()
Print (A.a, A.b)
Print (A.a, A.b = 300)
Print (oa.a, oa.b)
```

O/P: 10 20
10 300
10 300

```
class school:
    sname = 'ABC'
    saddr = 'xyz'
a = school()
Print (school.sname)
Print (school.saddr)
```



O/P: ABC
xyz

states:

→ The states are nothing but the data which is stored inside class (or) object

→ There are two types of states:

1. Generic state/ class members / static members
2. specific state / object members.

1. Generic state:

These are the members which will be common for Each and Every object that is created for a particular class

Eg:

If we consider bank as a class (details of bank is common for the objects) then the bank name, address of the back etc---
will be common for all the objects so that we can call them as generic members

2. specific states:

These are the states which will be different for Each and Every object which is created for a particular class

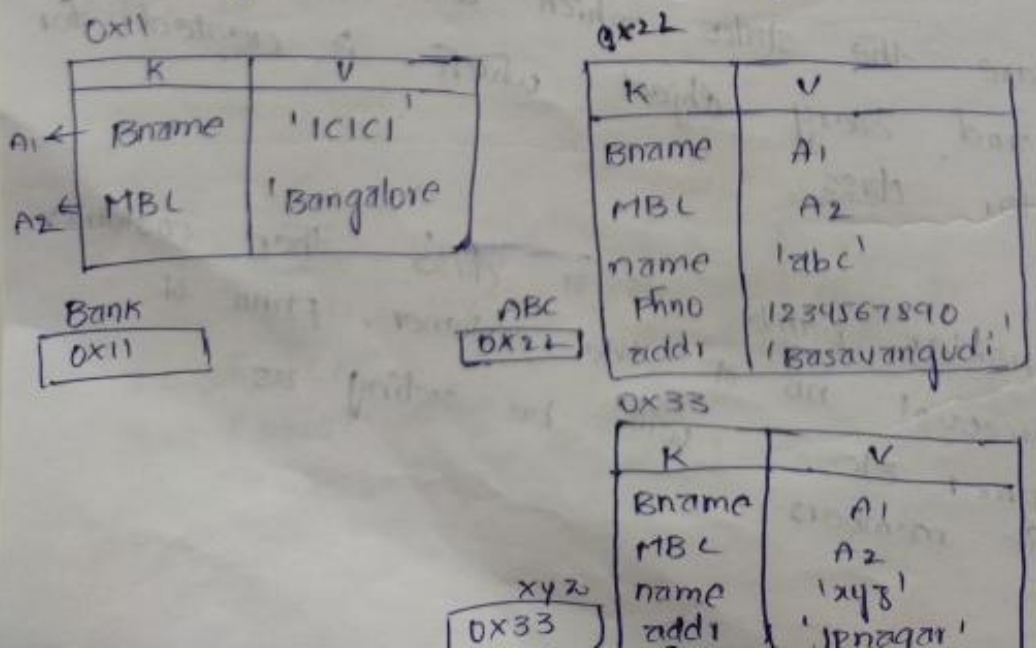
Eg:

If we consider bank as a class then costumer name, account no of the costumer, phno of the costumer etc.. will be acting as specific members.

```
class Bank:
    Bname = 'ICICI'
    MBL = 'Bangalore'        } - class members which are same for all the obj members.

ABC = Bank()
xyz = Bank()                 } - Object members.

ABC.name = 'abc'
ABC.phno = '1234567890'
ABC.addr = 'Basavangudi'

xyz.name = 'xyz'
xyz.phno = 9876543210
xyz.addr = 'jpnagar'

Print (Bank.Bname, Bank.MBL)
Print (ABC.Bname, ABC.MBL)
Print (ABC.name, ABC.phno, ABC.addr)
Print (xyz.name, xyz.phno, xyz.addr)
```

O/P:

```
ICICI   Bangalore
ICICI   Bangalore
abc     1234567890   Basavangudi
xyz     9876543210   jpnagar
```



constructor or Intialization method :

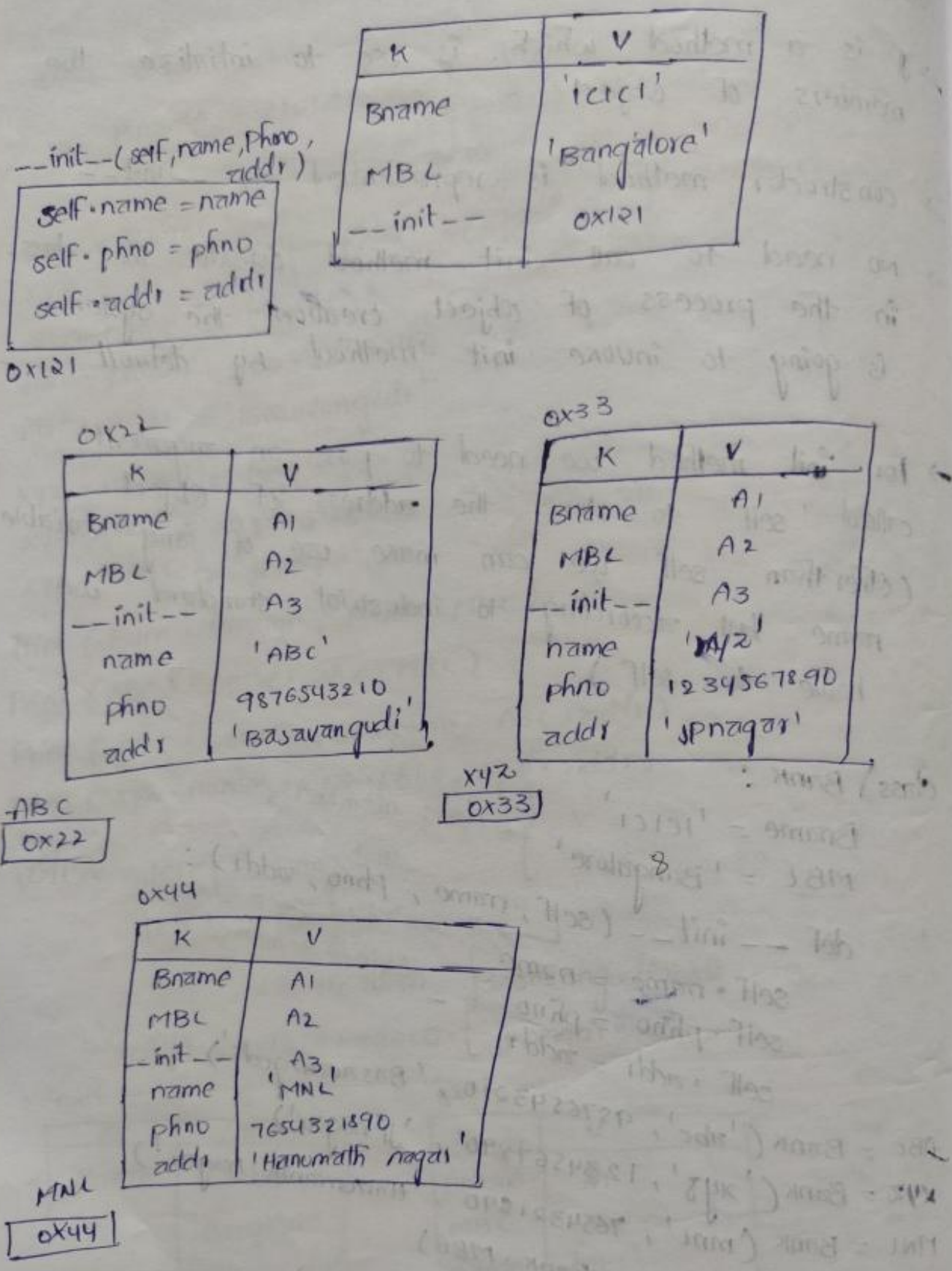
→ It is a method which is use to intialize the members of Object

→ constructor method is represented as '__init__'

→ No need to call init method outside the class, in the process of object creation the system is going to invoke init method by default.

→ For init method we need to pass an argument called 'self' to store the address of object (other than self we can make use of any variable name but according to industrial standard we have to self)

```
class Bank :
    Bname = 'ICICI'
    MBL = 'Bangalore'          } - class members
    def __init__(self, name, phno, addr):
                    ↓ it will hold the address of the obj
        self.name = name
        self.phno = phno       } - Object members
        self.addr = addr

ABC = Bank('abc', 9876543210, 'Basavangudi')
XYZ = Bank('xyz', 1234567890, 'JPnagar')
MNL = Bank('mnl', 7654321890, 'Hanumanth nagar')
Print(Bank.Bname, Bank.MBL)
Print(ABC.name, ABC.phno, ABC.addr)
Print(XYZ.name, XYZ.phno, XYZ.addr, XYZ.Bname)
Print(MNL.name, MNL.phno, MNL.addr, MNL.MBL)
```


__init__(self, name, phno, addr)
self.name = name
self.phno = phno
self.addr = addr
0x121
K
V
Bname
'ICICI'
MBL
'Bangalore'
__init__
0x121
0x22
K
V
Bname
A1
MBL
A2
__init__
A3
name
'ABC'
phno
9876543210
addr
'Basavangudi'
ABC
0x22
0x33
K
V
Bname
A1
MBL
A2
__init__
A3
name
'XYZ'
phno
1234567890
addr
'JPnagar'
XYZ
0x33
0x44
K
V
Bname
A1
MBL
A2
__init__
A3
name
'MNL'
phno
7654321890
addr
'Hanumath nagar'
MNL
0x44

Write a Prgm to create a class called school and create atleast two class members and five obj members.

```
class school:
    sname = 'LAHS'
    saddr = 'xyz'                    - class members
    def __init__(self, name, Rollno, class, phno, Email)
                                     it will hold the addr of each & every obj which we are created
        self.name = name
        self.Rollno = Rollno
        self.class = class           - Object members
        self.phno = phno
        self.Email = Email

sree = school('sree', 514, 10, 1234567890, 'sree@gmail.com')
Raju = school('raju', 513, 9, 9876543210, 'raju@gmail.com')
Print(sree.name, sree.Rollno, sree.Email)
Print(Raju.name, Raju.phno, Raju.sname)
Print(school.sname, school.saddr)
```

Write a Pgm to create class called Employee and create atleast three generic members and five specific member

```
class Employee :
    cname : 'MICROSOFT'
    cloc : 'Bangalore'
    CEO : 'satya nandh'
    def __init__(self, name, EmpID, Emp_Phno, Email, Eaddr):
        self.name = name
        self.EmpID = EmpID
        self.Emp_Phno = Emp_Phno
        self.Email = Email
        self.Eaddr = Eaddr
Rupa = Employee('rupa', '201', 1234567890, 'rupa@gmail.com', 'JP1')
Sree = Employee('sree', 202, 9948232420, 'S@gmail.com', 'JP1nagar')
Vinnu = Employee('vinnu', 203, 9876543210, 'V@gmail.com', 'Hnagar')
Print(Employee.cname, Employee.cloc, Employee.CEO)
Print(Rupa.name, Rupa.EmpID, Rupa.Email)
Print(Rupa.cname, Rupa.cloc, Rupa.CEO)
Print(Vinnu.name, Vinnu.EmpID, Vinnu.Eddr)
Print(Sree.Emp_Phno, Sree.EmpID, Sree.Email)
```

Behaviour of Functionalites :

These are the methods which will tell the user how to utilize the stored data

Methods inside the class get classified into.

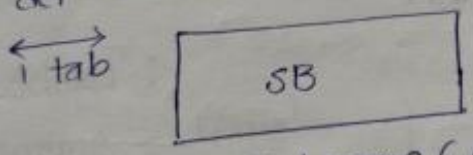
1. Object method
2. class method
3. static method

1. Object Method :

These are the method which are used to access and modify the members of object

for all the Object methods we need to pass self as an argument to store the address of an Object

syntax : def methodname (self, args):



cname . methodname (obj, args)
obj . methodname (args)

Eg :

```
class EMP:
    cname = 'pyspiders'
    caddr = 'Basavangudi'
    CEO = 'Hr. Girish'
    def __init__(self, name, phno, Eid, sal, desig)
        self.name = name
        self.phno = phno
        self.Eid = Eid
        self.sal = sal
        self.desig = desig
```

class members

Obj members

use to access the data when we write like this it will access the data of object members

```
def display(self):
    Print(self.name, self.phno, self.Eid,
          self.sal, self.desig)

def ch-phno(self, new):
    self.phno = new
def ch-desig(self, new):
    self.desig = new
def ch-sal(self, new):
    self.sal = new
```

obj method is use to modify & access the data

obj creation

```
Jeevitha = EMP('Jeevitha', 9121212121, 'pyspl23', 10000, 'ASE')
Anto = EMP('Anto', 9876543210, 'pyspl23', 10000, 'ASE')
Jeevitha.display()
EMP.display(Anto)
Jeevitha.ch_desig('ASE')
Jeevitha.ch_sal(20000)
Anto.ch_phno(8907654521)
Jeevitha.display()
Anto.display()
```

→ here we are accessing particular object by passing obj name

Memory diagram :

	K	V
A1	cname	'pyspiders'
A2	caddr	'Basavangudi'
A3	CEO	'Mr. Girish'
A4	--init--	0x11
A5	display	0x22
A6	ch-phno	0x33
A7	ch-desig	0x44
A8	ch-sal	0x55

EMP 0x21

0x21

K	V
cname	A1
caddr	A2
CEO	A3
--init--	A4
display	A5
ch-phno	A6
ch-desig	A7
ch-sal	A8
name	'Jeevitha'
phno	9121212121
Eid	'pyspl23'
sal	10000
desig	'ASE'

Jeevitha 0x441

0x441

WAP
three
four
class

0x51

K	V
cname	A1
caddr	A2
CEO	A3
__init__	A4
display	A5
ch-phno	A6
ch-desig	A7
ch-sal	A8
name	'Anto'
phno	1987654321
Eid	'pysp123'
sal	10000
desig	ASE

Anto
0x51

WAP to create a class called school and create three generic and 6 specific members and atleast four object methods.

```
class school :
    sname = 'Cathis'
    sHead = 'srinivas'
    saddr = 'JPT'                    } class members
    def __init__(self, name, aRollno, phno, Email,
                 class, fee)
        self.name = name
        self.aRollno = aRollno
        self.phno = phno
        self.Email = Email
        self.class = class
        self.fee = fee               } Obj members
```

```
def display (self):
    Print( self.name, self.Rollno, self.phno,
           self.Email, self.class, self.fee)

def ch_phno (self, new):
    self.phno = new
def ch_class (self, new)
    self.class = new
def ch_Email (self, new)
    self.Email = new
def ch_fee (self, new)
    self.fee = fee
```

Object method

14

obj creation

```
Rupa = school ('Rupa', 514, 1234567890, R@gmail.com, 10,
                                                14000)

Sree = school ('sree', 518, 9876543210, S@gmail.com, 7,
                                                10000)
```

```
Rupa.display()
school.display('Rupa')
Rupa.ch_phno(1234567812)
Rupa.ch_Email('r@gmail.com')
Sree.ch_fee("13000").
Sree.ch_Rollno(517)
Rupa.display()
Sree.display()
```

3. Class Methods:

It is a method which is use to access and modify the members of class.

→ For all the class method we have to pass 'cls' as a argument to store the address of class

→ To make the control to identify the class method we have to decorate it with '@ class method'

→ The syntax to defined class method is:

Syntax:

```
@ class method
def fname (cls, args) :
↔ 1 tab    SB
```

```
class Q4y132n34:
    Iname = 'Pyspiders'
    Trainer = 'Jeevitha'
    CRN = '3rd floor ROOM NO 1'
    Mentor = 'Rajkumar'
```

(class members)

```
    def __init__ (self, name, phno, Email, stream, YOP, Percentage):
```

(it will holds object address)

```
        self.name = name
        self.phno = phno
        self.Email = Email
        self.stream = stream
        self.YOP = YOP
        self.percentage = percentage
```

(Object members)

```
    def display(self):
        Print(self.name, self.phno, self.Email, self.stream,
              self.YOP, self.percentage)
    def ch_Email(self, new):
```

```
def ch_phno (self, new):
    self.phno = new
def ch_percentage (self, new):
    self.percentage = new
```

- Object methods
these are used to change the values from the object (original)

@classmethod → it access the data
def disp(cls): Every time we have to write class method
it is use to store the details of class members
    print(cls.Iname, cls.Trainer, cls.CNR, cls.Mentor)

```
@classmethod
def disp(cls, new):
@classmethod
def ch_CNR (cls, new):
    cls.CNR = new
@classmethod
def ch_Mentor (cls, new):
    cls.Mentor = new
@classmethod
def ch_Trainer (cls, new):
    cls.Trainer = new
```

- class methods
these are used to change the values

objects

```
Rupa = Q4YM32N34('Rupa', 1212---, r@gmail.com, CSE, 2021, 68)
Sree = Q4YM32N34('Sree', 1313---, S@gmail.com, ECE, 2021, 79)
Rupa.display()
Sree.display()
Q4YM32N34.disp()
Rupa.ch_Email('r@gmail.com')
Sree.ch_phno('9948232420')
Q4YM32N34.ch_Trainer('Anto')
Q4YM32N34.ch_Mentor('Krishna')
Q4YM32N34.ch_CNR('4 floor Room NO 2')
```

0x51

K	V
A1 Iname	'Pyspiders'
A2 Trainer	'Jeevitha'
A3 CNR	3[illegible]RNO1'
A4 Mentor	'Rajkumar'
A5 __init__	0x11
A6 display	0x12
A7 ch-Email	0x13
A8 ch-phno	0x14
A9 ch-percentage	0x15
A10 disp	0x16
A11 ch-CNR	0x17
A12 ch-Trainer	0x18
A13 ch-Mentor	0x19

Q4YA132n34

0x51

Rupa

0x61

K	V
Iname	A1
Trainer	A2
CNR	A3
Mentor	A4
__init__	A5
display	A6
ch-Email	A7
ch-phno	A8
ch-pertange	A9
disp	A10
ch-CNR	A11
ch-Trainer	A12
ch-mentor	A13
name	Rupa
phno	1212--
Email	r@gmail.com
stream	CSE
YOP	2021
Percentage	68

Rupa

0x61

0x71

K	V
Iname	A1
Trainer	A2
CNR	A3
Mentor	A4
__init__	A5
display	A6
ch-Email	A7
ch-phno	A8
ch-percentage	A9
disp	A10
ch-CNR	A11
ch-Trainer	A12
ch-Mentor	A13
name	sree
phno	1818--
Email	s@gmail.com
stream	ECE
YOP	2021
Percentage	79

sree

0x71

WAP to create a class called company create obj and class methods for it.

```
class cmpy :
    cName = 'Wipro'
    CEO = 'ABC'
    caddr = 'Basavangudi'
    MGR = 'Rupa'
    TLeader = 'Sree'
    def __init__(self, name, EmpID, Ephno, Email, Branch)
        self.name = name
        self.EmpID = EmpID
        self.Ephno = Ephno
        self.Email = Email
        self.Branch = Branch
    def display(self)
        Print(self.name, self.EmpID, self.Emphno,
              self.Email, self.Branch)
    def ch_Ephno(self, new)
        self.Ephno = new
    def ch_Email(self, new)
        self.Email = new
    @classmethod
    def disp(cls):
        Print(cls.cname, cls.CEO, cls.caddr, cls.MGR,
              cls.TLeader)
    @classmethod
    def ch_MGR(cls, new):
        cls.MGR = new
    @classmethod
    def ch_TLeader(cls, new):
        cls.TLeader = new
```

class members

Obj members

Obj methods

class methods

```
Gopi = CMPY('Gopi', 'g@mail.com', 9123144321, 301, 'Development')
Ravan = CMPY('Ravan', 302, 'r@mail.com', 9948232420, 'Testing')

Gopi.display()
Ravan.display()
CMPY.display()
Gopi.ch_Ephno('9948710251')
Gopi.ch_Email('G@gmail.com')
Ravan.ch_Ephno(9952136450)
Ravan.ch_Email('v@gmail.com')
CMPY.ch_MGR('MNC')
CMPY.ch_Tleader('xyz')
```

- if we want to change values after the display function these are optional based on the user

3. Static Method:

It is a method which is neither belongs to class nor belongs to object but it will act as supportive method for both class as well as object

→ For static method no need to pass an argument called 'self' and 'cls' to store the address of class and object

→ To create static method we have to use a decorator called '@staticmethod'

Syntax:

```
@staticmethod
def fname(args):
    SB
```

↔ 1 tab

```
Class Bank :
    Bname = 'SBI'
    MBL = 'Mysore'          - class member
    Bmname = 'xyz'

    def __init__(self, name, phno, addr, email, bal) :
        self.name = name
        self.phno = phno
        self.addr = addr
        self.Email = Email
        self.bal = bal          - Obj member

    def disp(self) :
        Print(self.name, self.phno, self.addr, self.Email,
                                                self.bal)

    def ch_phno(self, new) :
        self.phno = new
    def ch_addr(self, new) :
        self.addr = new
    def ch_email(self, new) :
        self.Email = new

    def deposite(self, amt) :
        self.bal = self.add(self.bal, amt)

    def withdraw(self, amt) :
        if amt <= self.bal :
            self.bal = self.sub(self.bal, amt)
            self.msg()
        Else :
            Print('insufficient bal')
```

- Obj method.

By using static method we can create

class method

```
@classmethod
def display(cls):
    Print(cls.Bname, cls.MBL, cls.Bmname)
@classmethod
def ch_MBL(cls, new):
    cls.MBL = new
@classmethod
def ch_Mname(cls, new):
    cls.Bmname = new
```

To modify or to access in the class method we have to write @classmethod while defining the class member

```
@staticmethod
def add(a, b):
    return a+b
@staticmethod
def sub(a, b):
    return a-b
@staticmethod
def msg():
    Print('modification is done')
```

Static Method

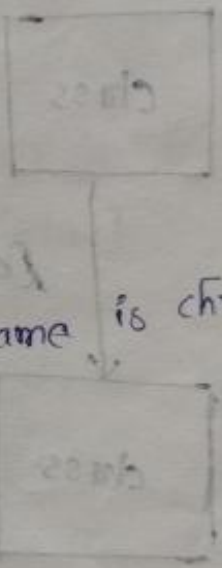
```
A = Bank('A', 912121212, 'xyz', 'a@gmail.com', 5000)
B = Bank('B', 9876543210, 'ABC', 'b@gmail.com', 10000)
Bank.display()
Bank.ch_Mname('AAAAA')
Bank.display()
A.disp()
A.deposite(15000)
A.withdraw(20000)
A.disp()
```

WAP to create a class called library and create at least three object method, three class method, two static method.

```
class Lib:
    cname : 'xyz'
    phno : 91212345
    email : 'x@gmail.com'
    def __init__(self, Bname, Author, PgNo)
        self.Bname = Bname
        self.Author = Author
        self.PgNo = PgNO
    def disp(self):
        Print(self.Bname, self.Author, self.pgNo)
    def ch_Bname(self, new):
        self.Bname = new
    def ch.Pgno(self, new):
        self.pgno = new
    def ch.Author(self, new):
        self.Author = new
    def Account(self, acct):
        self.account = self
    def new(self, acct):
        if acct == self.account:
            self.account = self.Bookname
            self.msg()
        Else:
            Print('create new account')
```

```
@classmethod
def display(cls):
    Print(cls.cname, cls.phno, cls.Email)
@classmethod
def ch_cname(cls, new):
    cls.cname = new
@classmethod
def ch_phno(cls, new):
    cls.phno = new
@classmethod
def ch_Email(cls, new):
    cls.Email = new
@staticmethod
def Bookname(
    return
@staticmethod
def msg()
    Print('Modification of Book name is changed')

A = lib('ABC', 'MNC', 230)
B = lib('HMC', 'Rupa', 580)
lib.display()
lib.ch_cname('Sree')
lib.display()
A.disp()
A.account
```

23

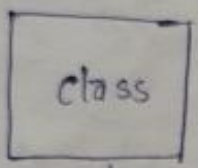


Inheritence :

It is a phenomenon of dividing the properties of one class to another class

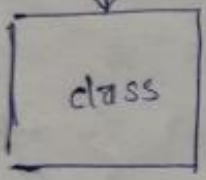
In this process the class from which the inherit the properties is called as parent / super / Base class.

The class to which we inheril the property is called as child class / sub class.



→ Parent / super / Base class

Deriving the properties.



→ sub / chiad class / Derive class

In Python inheritance got classified into the types

1. Single level
2. Multi level
3. Multiple level
4. Herachical
5. Hybrial.

1. Single level Inheritance :

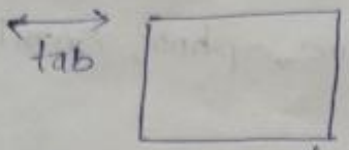
It is a phenomenon of deriving the properties of single parent class to single child class

Single inheritance will happen in one level it is called as single level inheritance.

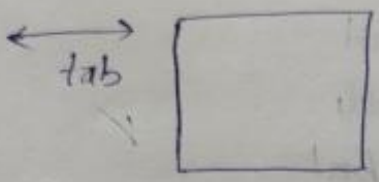
It consists of single parent class and single child class.

Syntax:

```
class parent_class :
    tab
```



```
class child_class ( parent_class ) :
    tab
```



```
class A
    a = 10
    b = 20
oa = A()
class B(A) :
    c = 10
ob = B()
Print (B.a)  → 10
```

class A

0x11

K	V
a	10
b	20

A
0x11

Obj of class A

0x12

K	V
a	A1
b	A2

oa
0x12

class B

0x22

	K	V
B1	a	A1
B2	b	A2
B3	c	10

B
0x22

Obj of class B

K	V
a	B1
b	B2
c	B3

ob
0x23

```
class Bank1 :
    Bname = 'SBI'
    MBL = 'Banglore'
    URoI = 12
    def __init__(self, name, phno, Email, addr, bbal):
        self.name = name
        self.phno = phno
        self.Email = Email
        self.addr = addr
        self.bbal = bbal
    def disp(self):
        Print(self.name, self.phno, self.Email, self.addr,
              self.bbal)
    def ch_phno(self, new):
        self.phno = new
    def ch_addr(self, new):
        self.addr = new
    def ch_Email(self, new):
        self.Email = new
    def deposite(self, new):
        self.bbal += amt
    def withdraw(self, amt):
        if self.bbal >= amt:
            self.bbal -= amt
        else:
            Print('insufficent fund')
```

```
Rupa = Bank1('Rupa', 9876768321, R@gmail.com, Bus...
                                                10000)

Rupa.disp()
class Bank2(Bank1):
    Bman = 'xyz'
    def __init__(self, name, phno, Email, addr, bbal, aadha
                                                    pan)

        self.name = name
        self.phno = phno
        self.Email = Email
        self.addr = addr
        self.aadhar = aadhar
        self.pan = pan
@classmethod
def display(cls)
    Print(cls.Name, cls.phno, cls.EmBL, cls.Bman)

Vivek.Bank2('vivek', 123323459, V@gmail.com, 'Qspiders',
```

Key	value
A1 Bname	'SBI'
A2 MBL	'Bangalore'
A3 LRO1	12
A4 def__init__	0x11
A5 def disp	0x12
A6 def ch_email	0x13
A7 def ch_phno	0x14
A8 def ch_addr	0x15
A9 def deposite	0x16
A10 def withdraw	0x17
A11 def display()	0x18
A12 def ch_LRO1	0x19

Bank1
0x11

28

0x22

Key	value
Bname	A1
MBL	A2
LRO1	A3
def__init__	0x11
def disp	0x112
def ch_email	A6
def ch_phno	A7
def ch_addr	A8
def deposite	A9
def withdraw	A10
def display()	0x113
def ch_LRO1	A12
Bman	'xyz'
def__init__	0x111
def disp	0x112
def display	0x113

Bank 2
0x22

constructor chaining:

It is a phenominan of invoking the constructor of the constructor of parent class by sitting in the constructor of child class

→ With the help of this method we can reduce the code redudancy problem

→ The syntax use to perform constructor chaining is

Syntax:

```
super().__init__(args)
super(childname, self).__init__(args)
Parentname.__init__(self, args)
```

In three syntax we can use any one to invoke the parent class to child class.

constructor chaining is use to reduce of code redudancy problem/ code increasing.

Method chaining:

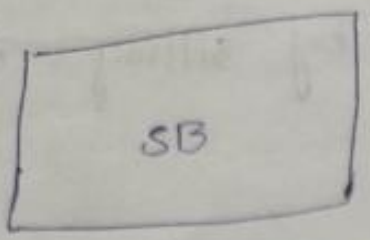
It is a phenominan of invoking the method of parent class by sitting in the method of child class

→ The syntax to perform the method chaining is

Syntax:

```
super().mname(args)
super(childname, self/cls).mname(args)
Parentname.mname(self/cls, args)
```

class Bank1 :

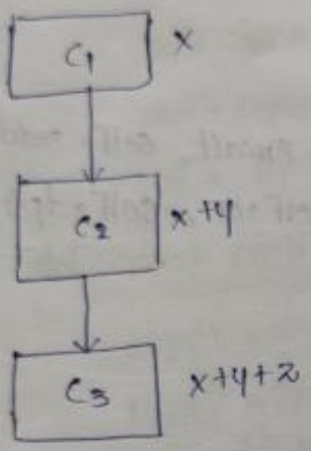



```
class Bank2(Bank1):
    Bname = 'xyz'
    def __init__(self, name, phno, Email, addr, bbal, aadh
        super().__init__(name, phno, Email, addr, bbal)
        super(childname, self/cls).mname(args)
        Parentname.mname(self/cls, args)
        self.aadhar = aadhar
        self.pan = pan
    def disp(self):
        super().disp()
        Print(self.aadhar, self.pan)
    @classmethod
    def display(cls):
        super().display()
        Print(cls.Bname)
Vivek = Bank2('Vivek', 9121212121212, 'v@gmail.com', 'Banglore',
                                    125000, 12345678, 'PAN12345')
vivek.disp()
Bank2.display()
```

2. Multi level Inheritance :

It is a phenomenon of deriving the properties of one class to another class by considering more than one level.

→ In this process the last derived class will be having all the properties of its parent classes



Syntax :

```
class C1 :
    1 tab
class C2(C1) :
    1 tab
    .
    .
    .
class Cn(Cn-1) :
    1 tab
```

Eg:

```
class C1 :
    a = 10
    b = 20
    def __init__(self, c) :
        self.c = c
class C2 :
    d = 45
    P = 90
class C3 :
    m = 1000
```

0x11

	K	V
A1	a	10
A2	b	20
A3	__init__	0x14

C1 [0x11]

0x13

	K	V
B1	a	A1
B2	b	A2
B3	__init__	A3
B4	d	45
B5	P	90

C2

0x14

	K	V
d1	a	B1
d2	b	B2
d3	__init__	B3
d4	d	B4
d5	P	B5
d6	m	1000

C3

```
class Resume:
    def __init__(self, name, phno, Email, addr, typ, sn, tp):
        self.name = name
        self.phno = phno
        self.Email = Email
        self.addr = addr
        self.typ = typ
        self.sn = sn
        self.tp = tp
    def disp(self):
        Print(self.name, self.phno, self.Email, self.addr,
              self.typ, self.sn, self.tp)
    def ch_phno(self, new):
        self.phno = new
    def ch_Email(self, new):
        self.Email = new
    def ch_addr=(self, new):
        self.addr = addr
vivek = Resume('vivek', 9121121212, 'vc@gmail.com', 'ABC', 2020,
               'xyz', 75)
vivek.disp()

class Resume1(Resume):
    def __init__(self, name, phno, Email, addr, typ, sn, tp, pcn, PP,
                 pyop):
        super.__init__(name, phno, Email, addr, typ, sn, tp)
        self.pcn = pcn
        self.PP = PP
        self.pyop = pyop
    def disp(self):
        super.desume().disp()
        print(self.pcn, self.PP, self.pyop)
```

obj members

Object methods

constructor chaining

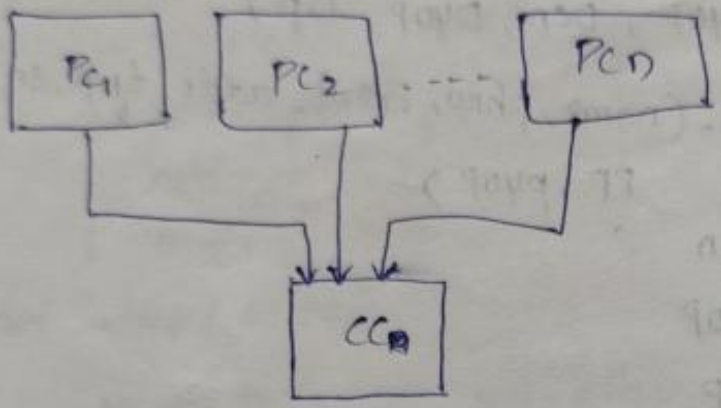
Method chaining

```
vivek = Resume1('Vivek', 9121212121, 'v@gmail.com', 'ABC', 2021,
kavin                'xyz', 75, 'PN Nagar', 2022, 85)
vivek.disp()

class Resume2(Resume1):
    def __init__(self, name, phno, email, addr, tyop, sn, tp, pcn,
                 pp, pyop, dcn, dyop, dp)
    sup super().__init__(name, phno, Email, addr, tyop, sn, tp, pcn
                         pp, pyop) → constructor chaining
        self.dcn = dcn
        self.dyop = dyop
        self.dp = dp
    def disp(self):
        super.disp() → Method chaining
        Print(self.dcn, self.dyop, self.dp)

Vivek = Resume2('vivek', 9121212121, 'v@gmail.com', 'ABC', 2021,
                'xyz', 75, 'PN Nagar', 2022, 85, 'MNL', 2026, 90)
Vivek.disp()
```

3. Multiple Inheritance:

It is a phenomenon of deriving the properties of multiple parent classes to a single child class

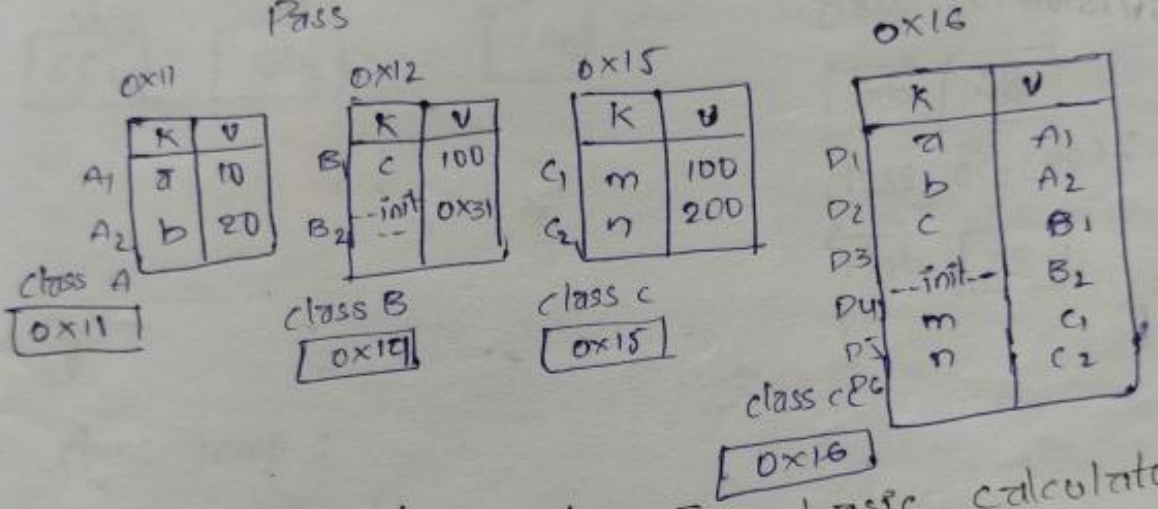
→ Flow chart



Syntax:

```
class PC1:
    SB
class PC2:
    SB
    :
class PCn:
    SB
class CC(PC1, PC2, PC3 --- PCn):
    SB
```

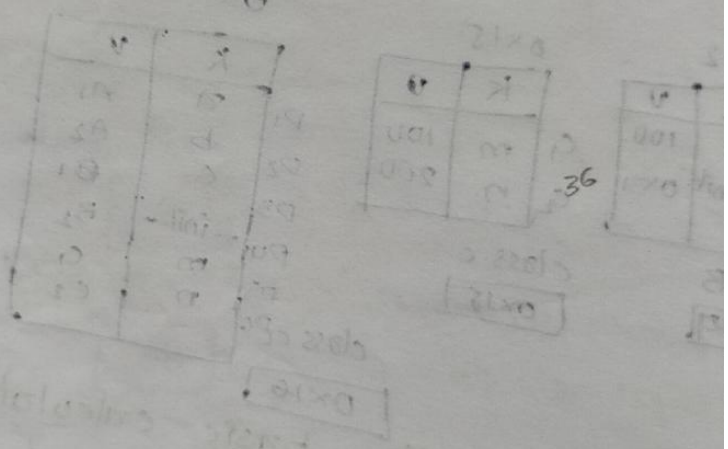
Eg:
```
class A :
    a = 10
    b = 20
class B :
    c = 100
    def __init__(self, d) :
        self.d = d
class C :
    m = 100
    n = 200
class CC(A, B, C):
    Pass
```



Write a Pgm to create the basic calculator

```
class Add :
    @staticmethod
    def add(a,b):
        return a+b
class sub :
    @staticmethod
    def sub(a,b):
        return a-b
class Mul :
    @staticmethod
    def mul(a,b):
        return a*b
```

```
class Div :
    @staticmethod
    def div(a,b) :
        return a// b
class cal(add, Sub, Mul, Div) :
    Pass
Print('sum is :', cal.add(10,20))
Print('diff is :', cal.sub(40,10))
Print('mul is :', cal.mul(4,3))
Print('division is :', cal.div(45,5))
```

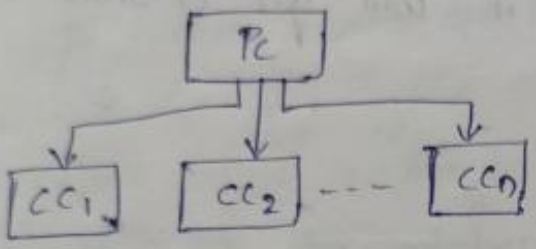


4. Hirachical inheritance:

It is a phenominan of deriving the properties of single parent class to multiple child classes

→ While creating template for any of the projects we will make use of hirarechical inheritance.

Flow Diagram:



Syntax:

```
class PC:
    ↔ itab  SB
class CC1(PC):
    ↔ itab  SB
class CC2(PC):
    ↔ itab  SB
    !
    !
```

Eg:

```
class Temp:
    Iname = ''
    CEO = 'Mr. Girish'
    Iloc = 'Basavangudi'
    def __init__(self, name, phno, Email, stream, yop, tp):
        self.name = name
        self.phno = phno
        self.Email = Email
        self.stream = stream
        self.yop = yop
        self.tp = tp
class J(Temp):
    Iname = 'Jspiders'
class Q(Temp):
    Iname = 'Qspiders'
class Py(Temp):
    Iname = 'Pyspiders'
```

5. Hybrid Inheritance :

Hybrid inheritance it is a phenamenon of deriving the properties of one class to another class where it includes more than one type of inheritance

→ For this type we don't have the proper syntax and the Proper flow diagram. because based on the user requirment it will get created.

Eg :

```
class A :
    a = 10
class B(A):
    b = 20
    def __init__(self, c, d):
        self.c = c
        self.d = d
class C :
    P = 100
    Q = 200
class D(B, C):
    Pass
```

upto here single level

Multiple level inheritance

A_1	K	V
	a	10

	K	V
B_1	a	A_1
B_2	b	20
B_3	init	0x11

	K	V
C_1	P	100
C_2	Q	200

K	V
a	B_1
b	B_2
init	B_3
P	C_1
Q	C_2

Encapsulation / Access specifiers:

Access specifiers are the members which will specify whether the user can access them outside the class or not.

There are three types of access specifiers in Python:

1. Public access specifier
2. Protected access specifier
3. Private access specifier

1. Public access specifier:

These are the members which will allow the user to access them outside the class

→ Normally whatever the members we are creating inside the class will be considered as public members and we can access them outside the class without getting any Error.

Eg:

```
class Akhila:
    a = 10
    b = 20
    def __init__(self, c):
        self.c = c
    def disp(self):
        Print(self.c)
    @classmethod
    def display(cls):
        Print(cls.a, cls.b)
    @staticmethod
    def msg():
        Print('hello Everyone')
```

```
Print (Akhila.a, Akhila.b)
ob = Akhila(30)
ob.disp()
Akhila.display()
Akhila.msg()
```

2. Protected access specifiers :

These are the members which should give protection to the data stored inside the class.

→ But, in python protected member will not give any protection to the data stored inside the class

→ Because of the above reason Protected members are not in used, in some websites they are given only two access specifiers.

→ Syntax to create Protected members is

Syntax :

```
_var = value
```

for method :

```
def _fname(args):
    SB
```

(↔ 1 tab)

Eg:

```
class Akhila:
    _a = 10
    _b = 20
    def __init__(self, c):
        self._c = c
    def _disp(self):
        Print(self._c)
```

```
    @classmethod
    def _display(cls):
        Print(cls._a, cls._b)
    @staticmethod
    def _msg()
        Print('Hello Everyone')
Print(Akhila._a, Akhila._b)
oa=Akhila(50)
oa._disp()
Akhila._display()
Akhila.msg()
```

O/P: 10 20
50
10 20
Hello Everyone

3. Private access specifier:

These are the members which will not allow the user to access them outside the class

→ (or) These are the members which will provide the protection to the data store inside the class

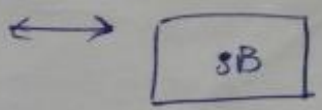
→ The syntax to create private members is

Syntax:

```
__var = value
```

For method:

```
def __fname(args):
```



Eg:

```
class Akhila:
    __a = 10
    __b = 20
    def __init__(self, c):
        self.__c = c
    def __disp(self):
        Print(self.__c)
    @classmethod
    def __display(cls):
        Print(cls.__a, cls.__b)
    @staticmethod
    def __msg():
Print(Akhila.__a, Akhila.__b) # Error by saying that no attribute called '__a'
oa = Akhila(50)
oa._disp()
Akhila.__display()
Akhila.__msg()
```

→ it is not possible to call outside the class, but we have Private members we can call outside of the class syntax to call

Syntax:

classname._classname__var / fonname(args)

Based on the three specifiers.

Eg:

```
class Company :
    cname = 'ABC'
    MBL = 'PQR'
    CEO = 'xyz'
    __Profit = '12%'
    def __init__(self, name, phno, Email, Eid, sal):
        self.name = name
        self.__phno = phno
        self.Email = Email
        self.__Eid = Eid
        self._sal = sal

    def __disp(self):
        Print(self.name, self.__phno, self.Email,
              self.__Eid, self._sal)

    @classmethod
    def display(cls):
        Print(cls.cname, cls.MBL, cls.CEO)

    def __ch_sal(self, new):
        self._sal = new
    def get_disp(self):
        return self.__disp

oa = company('n', 9121212121, 'a@gmail.com', 'abc123', 10000)
res = oa.get_disp()
res()
# Print(company._company__Profit)
```

#normally it is not possible to access the data which is Private member

#creating another function to storing address of the Object to access the data.

company._company__disp(oa)

WAP to find the sum of an individual digits present inside an integer number by using class method.

```
class sam:
    num=int(input('Enter the num:'))
    @classmethod
    def sam(cls):
        n=cls.num
        while n!=0:
            d=n%10
            sum+=d
            n=n//10
        Print(sum)
Sam.sam()
```

WAP to find the num of occurance of a specified character present inside the given string by using class method.

```
class string:
    st=input('Enter the string:')
    ch=input('Enter the char:')
    @classmethod
    def sam(cls):
        if ch in cls.st:
            for c in st:
                for i in cls.st:
                    if i==cls.ch:
                        c+=1
            Print(c)
        else:
            Print('char is not fount')
string.sam()
```

WAP to extract all the integer numbers present inside the given list by using static method.

```
class Ext:
    @staticmethod
    def sam(a):
        out=[]
        for i in a:
            if type(i)==int:
                out+=[i]
        Print(out)
Ext.sam([12,34,5.6,'hello',[1,2]])
```

WAP to create a dictionary where the key should be the character present inside the string and value should be the number of occurance of character by using object method.

```
class A:
    def __init__(self,st):
        self.st=st
    def sam(self):
        out={}
        for i in self.st:
            out[i]=self.st.count(i)
        Print(out)
oa=A(input('Enter the string:'))
oa.sam()
```

O/P:

Enter the string : collection

{'c':2, 'o':2, 'l':2, 'e':1, 'i':1, 'n':1}

WAP to print all the repeated characters from the given string by using all the three method.

```
class string :
    st = input('Enter the string :')
    @classmethod
    def som(cls) :
        for i in cls.st :
            n = cls.st.count(i)
            if n > 1
            print(n)
            else :
                print('retry the string')
string.som()
```

```
class string:
    @staticmethod
    def som(st):
        for i in st:
            n = st.count(i)
            if n > 1
                print(n)
string(som('connection'))
```

```
class string :
    def __init__(self, st):
        self.st
    def som(self):
        for i in self.st :
os = string('connection')
```

Abstraction:

1. Abstract method
2. Abstract class
3. concrete class

def sam(a) → fun declaration

Print a → fun definition

1. Abstract method

It is a method which will be having function declaration but not function definition

Eg: def sam(a)
    pass

3. Abstract class:

If the class consist of atleast one abstract method in it then we can call that class as abstract class.

→ for abstract class creating object is not possible, if we try to create control will through an error

3 concrete class:

It is a class which does not consist of abstract method in it

Abstract:

Abstraction is a phenominon of hiding the functionalities from the user.

Why abstraction not used:

Abstraction is not used in python because whenever we creat abstrad class when we want to convert into concrete class we have mention every class that is the rection

Syntax to create abstract class:

Syntax:

```
from abc import ABC, abstractmethod

class cname(ABC):
    @abstractmethod
    def funame(args):
        pass
```

Eg:

```
from abc import ABC, abstractmethod
class SAM(ABC):
    @abstractmethod
    def msg():
        pass
    @abstractmethod
    def msg1():
        pass
oa = SAM()
```

We can convert abstract class inthe form of concrect class by implementing or writing the functionalites for all the abstract methods present inside an abstract class.

Eg: To convert the abstract class to concrect class:

```
from abc import ABC, abstractmethod
class SAM(ABC):
    @abstractmethod
    def msg():
    @abstractmethod
    def msg1():
        Pass
class Demo(SAM):
    def msg():
        Print('something')
    def msg1():
        Print('nothing')
```

Polymorphism :

Polymorphism is a phenaminon of making the operator or method to work on different operation

→ Polymorphism can be done in two ways

1. Method Overloading
2. Operator Overloading.

1. Method Overloading :

It is a pheraminan of making the same method to work on two or more functions or operation.

→ In python method overloading is not there

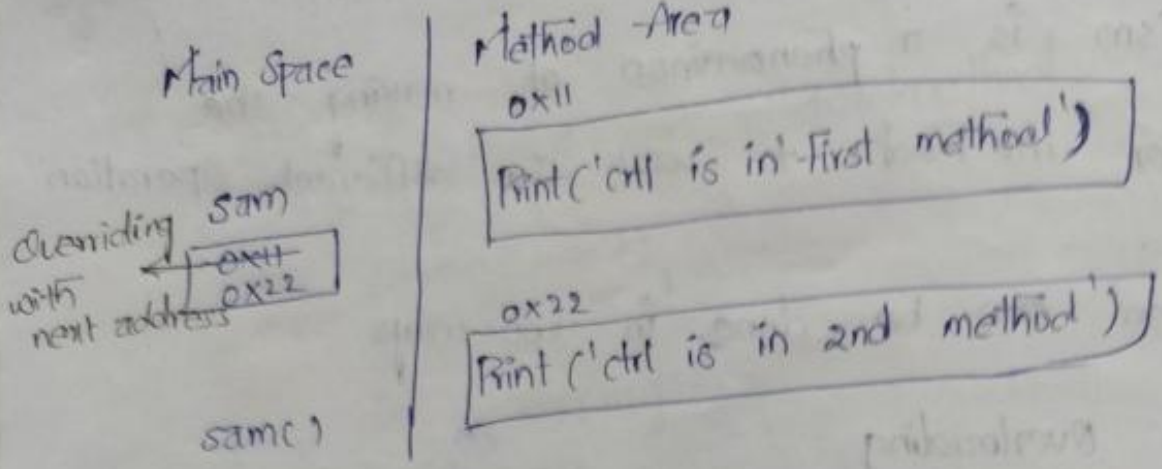
→ If we try to do method overloading in python by default it will perform method overriding.

Eg:

Method Overriding :

If we create two different methods with the same name in the single file then the address of previous function will get overrided by the address of next function.

```
def sam():
    Print ('ctrl is in first method')

def sam():
    Print ('ctrl is in 2nd method')

sam()
```



2. Operator Overloading:

It is a phenomenon of [Operator Overloading] making the operator to work on object of user defined classes by invoking the respective magic method.

Eg:

If we want to perform addition between two values control will go and check for a method called

__add__

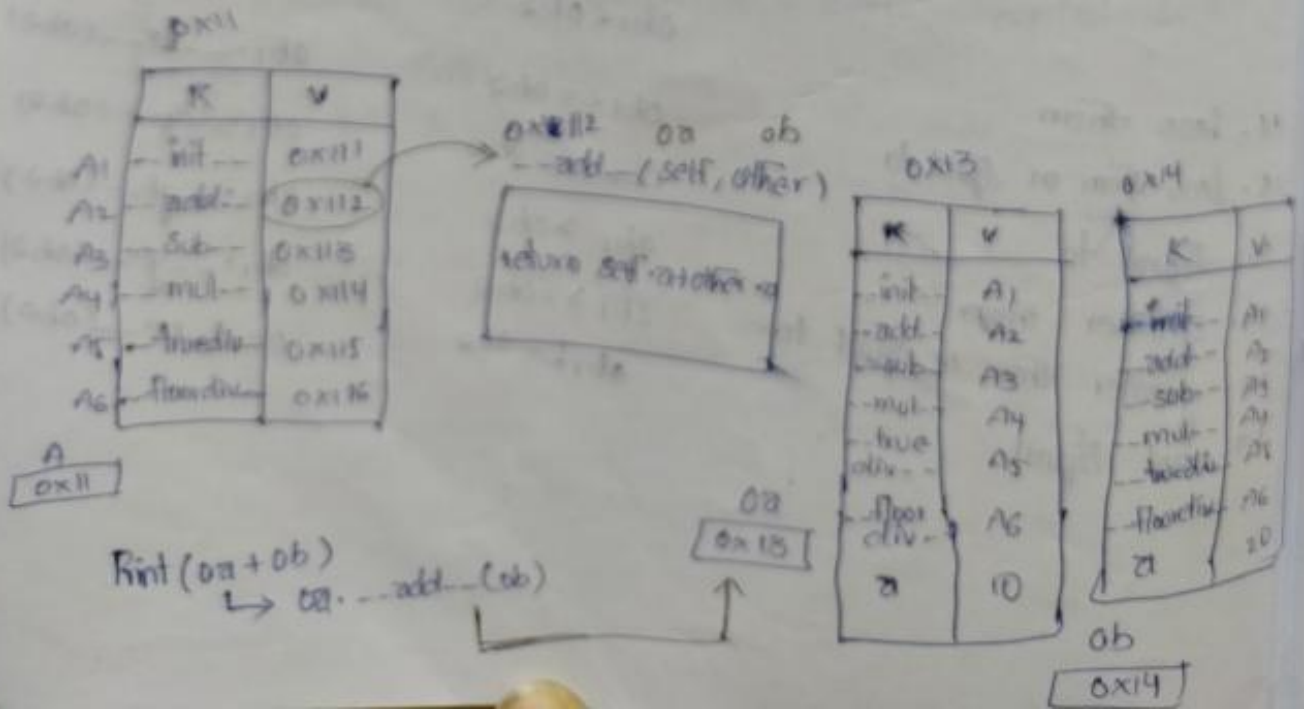
→ If it is present inside the class then we will get addition of two values else, control will through an Error

Operation	Operator	internal Operation
1. addition	ob1 + ob2	ob1.__add__(ob2)
2. substraction	ob1 - ob2	ob1.__sub__(ob2)
3. Multiplication	ob1 * ob2	ob1.__mul__(ob2)
4. power	ob1 ** ob2	ob1.__pow__(ob2)
5. division	ob1 / ob2	ob1.__truediv__(ob2)
6. floor div	ob1 // ob2	ob1.__floordiv(ob2)
7. Modulos	ob1 % ob2	ob1.__mod__(ob2)
8. Bitwise Left shift	ob1 << ob2	ob1.__lshift__(ob2)
9. Bitwise Right shift	ob1 >> ob2	ob1.__rshift__(ob2)
10. Bitwise AND	ob1 & ob2	ob1.__and__(ob2)
11. Bitwise OR	ob1 | ob2	ob1.__or__(ob2)
12. Bitwise XOR	ob1 ^ ob2	ob1.__xor__(ob2)
13. Bitwise NOT	~ob1	ob1.__invert()
14. less than	ob1 < ob2	ob1.__lt__(ob2)
15. less than or Equal to	ob1 <= ob2	ob1.__le__(ob2)
16. Equal to	ob1 == ob2	ob1.__eq__(ob2)
17. greater than	ob1 > ob2	ob1.__gt__(ob2)
18. greater than or Equal to	ob1 >= ob2	ob1.__ge__(ob2)
19. not Equal	ob1 != ob2	ob1.__ne__(ob2)

Eg: On Arthematic Operator

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other.a
    def __sub__(self, other):
        return self.a - other.a
    def __mul__(self, other):
        return self.a * other.a
    def __truediv__(self, other):
        return self.a / other.a
    def __floordiv__(self, other):
        return self.a // other.a
```

```
oa = A(10)
ob = A(20)
Print(oa/ob)
Print(oa+ob)
Print(oa*ob)
```



Eg: On Bitwise Operator

```
class B:
    def __init__(self, b):
        self.b = b
    def __and__(self, other):
        return self.b & other.b
    def __or__(self, other):
        return self.b | other.b
    def __xor__(self, other):
        return self.b ^ other.b
    def __invert__(self):
        return ~self.b

oa = B(10)
ob = B(30)
Print(oa & ob)
Print(oa | ob)
Print(oa ^ ob)
Print(~oa)
```

Eg: On logical Operators

```
class C:
    def __init__(self, c):
        self.c = c
    def __lt__(self, other):
        return self.c < other.c
    def __le__(self, other):
        return self.c <= other.c
    def __eq__(self, other):
        return self.c == other.c
    def __gt__(self, other):
        return self.c > other.c
    def __ge__(self, other):
        return self.c >= other.c
    def __ne__(self, other):
        return self.c != other.c

oa = C(10)
ob = C(20)
Print(oa < ob)
Print(ob > oa)
Print(oa == ob)
```

0x14

K	V
__init__	A1
__add__	A2
__sub__	A3
__mul__	A4
__truediv__	A5
__floordiv__	A6
a	20

ob
0x14

Wap to check weather the given number is Even (or) odd by using function if number is Even then true else false

```
def sam(n):
    if n%.2==0:
        Print('Even')
    else:
        Print('odd')
```

```
def sam(n):
    if n%.2==0:
        return True
    return False
Print(sam(10))
```

sam(10)

Reduce code:

```
def sam(n):
    return n%.2==0:
Print(sam(10))
```

Lambda :

Lambda is a Keyword which is use to perform some small small operations in Python

→ Lambda is an anonimous function (it is not having any name)

→ Because of the above reason we will store the address of lambda function into one variable and that variable is going to act as function name

→ Syntax to create lambda function is

Syntax :

```
var = lambda arg1, arg2, --- argn : return expression
Print (var(arg1, arg2, --- argn))
```

Eg :

```
res = lambda n : n%2==0
Print (res(10))
```

O/P : True

Wap to check weather the given character is present inside the collection are not

```
out = lambda (n, st) : st in n
Print ( out [2, 3, 5.2, 10] , 3)
Print (out ('rai', 'i'))
```

WAP to find the sum of three number by using lambda

```
sum = lambda a,b,c : a+b+c
Print(sum(10, 5, 12))
```

WAP to find the product of minimum three numbers and max five numbers by using lambda

```
Pd = lambda a,b,c,d,e,: a*b*c
Print (Pd (10, 5, 3))
```

WAP to check weather the given number is integer or not

```
num = lambda n : type(n) == int
Print (num (5))
```

Wap to check weather the given character special symbol or not by using lambda.

```
Sp = lambda st : not ('a'<= st <='z' or
                      'A'<= st <='Z' or '0'<= st <='9')
Print (Sp('&'))
```

WAP to check weather the given value is of collection or not

```
coll = lambda var : type(var) not in [int, float, complex, bool]
Print (coll([2, 10, 'Rai', 2.3]))
```

(or)

```
coll = lambda var : type(var) in [str, list, tuple, set, dict]
Print (coll([2, 10, 'Rai', 2.3]))
```

Map function map() :

Map is a function which is use to perform some specific operation on each and every value stored inside collection

→ The syntax used is

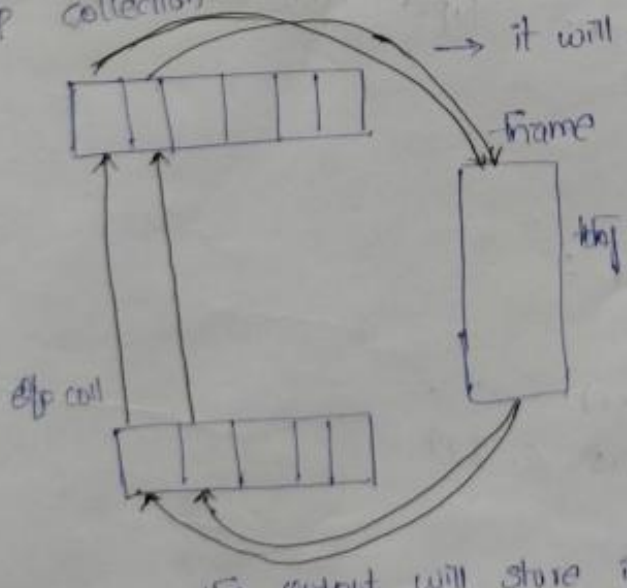
Syntax :

```
Var = map(fname, collection)
```

→ map() will create an output collection and the address of output collection will get stored into an output variable

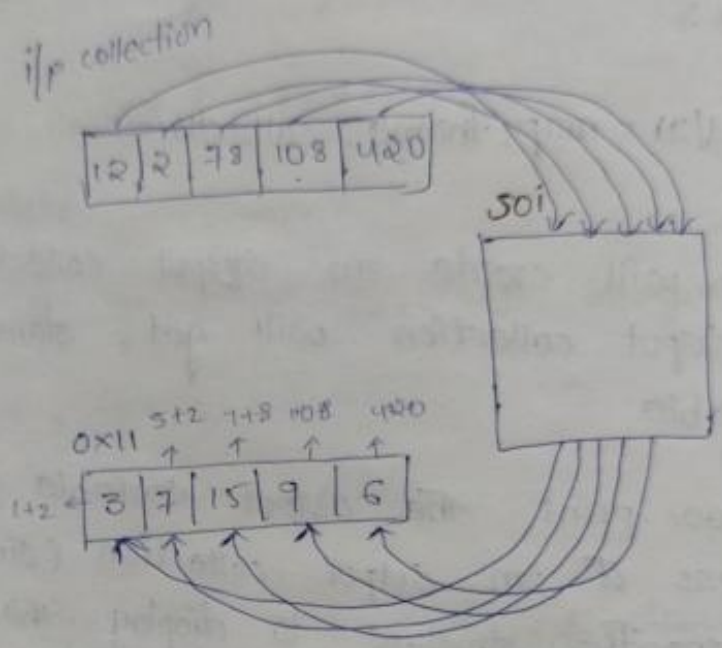
→ If we print the output variable we will get the address of an output collection. (since it is not having a specific structure to display the values)

→ To get the output values on the screen type casting is required.



WAP to find the sum of an individual digits of the integer number present inside the given homogeneous list collection.

```
def soi(n):
    sum = 0
    while n != 0:
        d = n % 10
        sum += d
        n = n // 10
    return sum
a = [12, 50, 78, 108, 420]
out = []
for i in a:
    out += [soi(i)]
Print (out)
```



by using map

```
res = map(soi, a)
Print (list(res))
```

WAP to find the factorial of a given number present inside a homogeneous tuple collection by using map function.

```
def fact(n):
    if n == 0 or n == 1:
        return 1
    return n * fact(n-1)
Out = map (fact, (12, 15, 6, 3))
Print (tuple (out))
```

Wap to find square of all the numbers present in the range 1 to 10 by using map.

```
def sq(n):
    return n*n
res = map(sq, range(1,11))
print(list(res))
```

(01)

```
Even = lambda n : n**2
Out = map(Even, range(1,11))
print(list(Out))
```

(01)

```
Even = lambda n :n**2
print(list(map(Even, range(1,11))))
```

(01)

```
print(list(map(lambda n :n**2, range(1,11))))
```

Wap to find add 10 to all the numbers present between the range 20 to 30 by using map.

```
add = lambda n : n+10
Out = map(add, range(20,31))
print(list(Out))
```

Wap to store the length of the string present inside homogeneous list collection using map.

```
def st(c):
    return len(c)
res = map(st, ['hai', 'hello', 'how'])
print(list(res))
```

```
st = lambda n : len(st)
res = map(st, ['hai', 'hello', 'how'])
print(list(res))
```

Filter function:

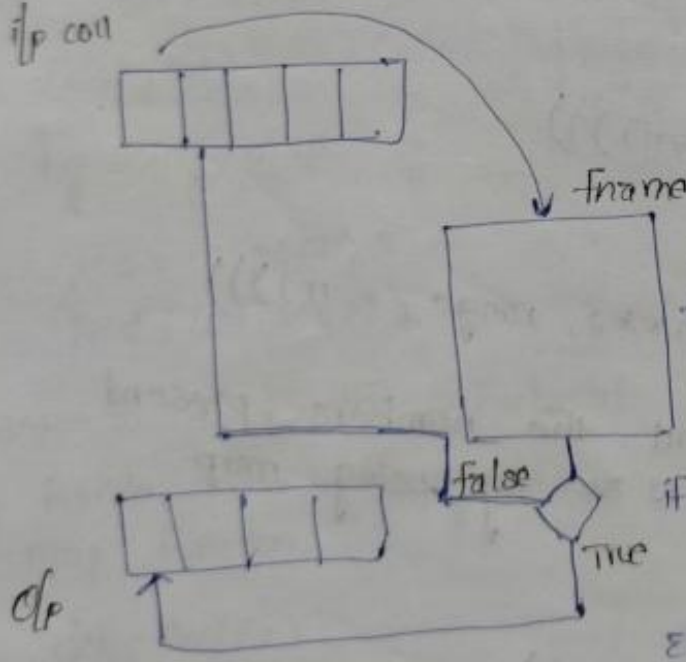
filter():

- filter is a function which is use to filterout the collection by getting only the required values
- the syntax used is

syntax:

var = filters(frame, coll)

→ where func should return either true or false
→ in this case also typecasting is required to get the values on the screen



WAP to Extract only the Even number within the range 1 to 100 using filter

```
a = lambda n : n%2==0
res = filter(a, range(1,101))
Print(list(res))
```

(or)

```
Print(list(filter(lambda n:n%2==0, range(1,101))))
```

WAP to find the cube of the Even numbers between the range 1 to 20.

```
~~a = lambda n:n%2==0~~
Out = filter(lambda n:n%2==0, range(1,21))
res = map(lambda n:n**3, Out)
print(list(res))
```

(or)

```
Print(list(map(lambda n:n**3(filter(lambda n:n%2==0
                range(1,21))))))
```

WAP to Extract all the substrings present inside an homogeneous list collection if it consist of one a in it and length of the collection/string is greater or equal to 5

```
Print(list(filter(lambda st:a in st and len(st)>=5,
        ['kaaii','hello','mango','bau']))))
```

File handling:

File:

File is a container which is use to store the data where the data can be either text, txt, mp3, mp4, img ... etc

→ Handling the file is nothing but reading the data from the file (or) writing the data into the file.

→ To perform the operations with respective file getting the accessability is very important.

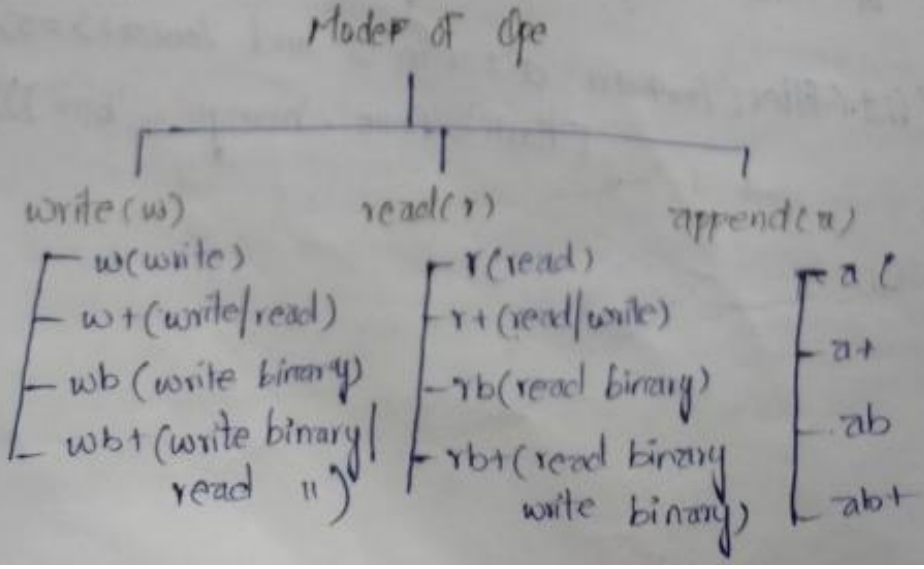
→ In python we are going to use a function called open() to get the accessability of any file

→ open() consist of two arguments

i) file with Extension / Location of the file

ii) mode of operation

→ mode of operation is again classified into three types.



The syntax used to get accessability is

Syntax:

```
Var = Open('file.txt/loc', 'mode')
```

Writing the data into the file:

There are two syntaxes to write the data into the file

Syntax:

```
Var.write(data)
Var.writelines(data)
Var.close()
```

After modifing the data we have to close the file by using close().

→ In write mode if the file is not Existing then it will create a file then it will write the data

→ If the file is already Existing it will override the previous data by the new data

→ write function use to write only the single kind of data in the form of string

→ writelines is a function which is use to write multiple data into the file in the form of list

→ Once after performing the operation we need to close the open file.

Eg:

```
write()
fo = open('Pro1.txt', 'w')
data1 = 'good afternoon'
fo = write(data1)
fo.close()
```

```
writelines()
fo = open('Pro1.txt', 'w')
data1 = 'good afternoon\n'
data2 = 'good morning'
data = [data1, data2]
fo = writelines(data)
fo.close()
```

Reading the data from the file:

Syntax:

```
res = var.read()
res = var.~~fine~~readline()
res = var.readlines()
```

Eg: read()

```
fo = open('Pro1.txt', 'r')
res = fo.read()
fo.close()
Print(res)
```

o/p: good afternoon
good morning.

readline()

```
fo = open('Pro1.txt', 'r')
res = fo.readline()
res1 = fo.readline()
fo.close()
Print(res)
Print(res1)
```

o/p:

good afternoon
good morning.

readlines()

```
fo = open('Pro1.txt', 'r')
res = fo.readlines()
fo.close()
Print(res)
```

o/p: ['good afternoon\n', 'good morning']

Appending the data:

appenda is Exactly Similar to write mode but in with case the mode of operation will consist of 'a'

→ Instead of overriding the previous data it will add a new data to the Existing data

→ The functions use to perform append operation is

```
var.write(data)
var.writelines(data)
```

Eg:

```
fo = open('fno2.txt', 'w')
data = 'In best Engneers'
fo.write(data)
fo.close()
```

location of the file.

```
fo = open(r'c:\users\Qsp\desktop\Pydata.txt', 'r')
res = fo.read()
fo.close()
Print(res)
```

Passing Techniques:

It is a phenomenon of providing security to the data while sending from source to destination or one file to another file

There are two types of Passing Techniques

1. json
2. pickle

1. Json passing Techniques:

It is a phenomenon of sending the data from source to destination by converting it in the form of string type.

→ In this case Original data will get converted in the form of Encrypted data.

Encryption:

It is a phenomenon of converting the data from one type to another the final encrypted data is in string type

→ We can't convert/ Encrypt the data directly for that we will make use of function called dumps()

→ syntax used is,

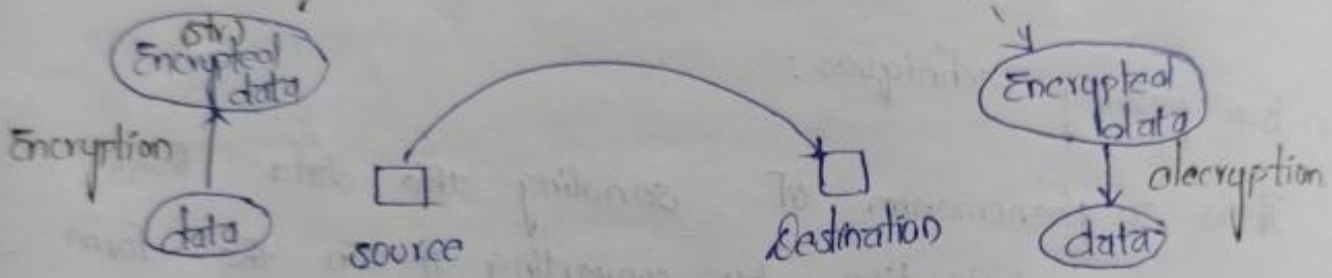
Syntax:

```
import json
var = json.dumps(data)
```

→ At the destination we need to convert Encrypted data in the form of original data.

Syntax:

```
import json
var = json.loads(Encry)
```



→ whenever we want to send the data source to destination we have to convert the data to Encrypted data

→ The Encrypted data is present in string format because string is high secure data.

→ The final data of Encrypted data is in string we can't access

→ for that we have to Encrypted data to data (by decryption) to access the data individual.

Eg:

Encryption:

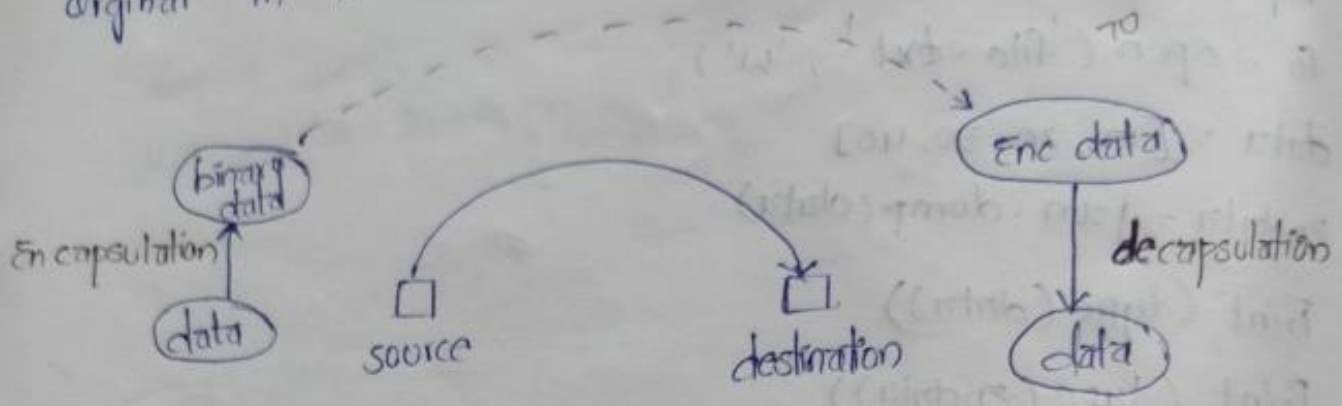
```
import json
fo = open ('file.txt', 'w')
data =,[10, 20, 30, 40]
Endata = json.dump(data)
Print (type (data))
Print (type (endata))
fo. write (Endata)
fo. close()
```

Decryption:

```
fo = open ('file1.txt', 'r')
res = fo.read()
Print(type(res))
Out = json.loads (res)
Print (type (out))
```

2. Pickle Parsing Technic :

It is a phenomenon of sending the data securly from source to destination by converting the original in the form of binary data.



→ In pickle parsing Thechnic whenever the data to be converteal source to destination the it is data should be Encapsulation the final data in in binary form

→ It is more secure than json and the binary lang can only understand by machine.

→ If the data in Encapsulation we want to access in original after reaching to destiontion we have to decapsulation

→ Means we are converting the binary format to Original data.

→ The pickle parsing Technic is mostly useal in the Online transaction like phone pay, google pay, zomato bank transactions etc--

Encapsulation:

It is a phenamenon of converting the any type of data in to binary formal / Lang.

```
import pickle
enc = pickle.dumps(data)
```

Decapsulation:

It is a phenamenon of converting the binary lang into to original data (getting back the original data)

```
var = pickle.loads(enc)
```

encapsulation:

```
import pickle
fo = open('file2.txt', 'wb')
data = [10, 20, 30]
enc = pickle.dumps(data)
fo.write(enc)
fo.close()
```

decapsulation:

```
fo = open('file2.txt', 'rb')
res = fop.read()
out = pickle.loads(res)
fo.close()
Print(out)
```

Exception Handling :

Exception :

It is an authorized Event which will arise while Executing the program and which will stop the flow of Execution

Apart from syntax Error all the Errors are considered as Exception.

Exception Handling :

It is a phenomenon of handling the Exception where we can run the code without getting any Exception.

In Python Exception Can be handled by :

1. try
2. Except keywords,

→ where try block consist of problem statement and Except consist of solution for the above problem

→ Exception handling got classified into three types

1. specific Exception handling
2. Generic exception handling
3. Default exception handling

1. Specific Exception Handling

It is a type of exception handling, we can use this only if we know the type of Error for a single try block we can have n number of Except block

Eg:

```
def div():
    try:
        a = int(input('Enter the val :'))
        b = int(input('Enter the val :'))
        res = a/b
        Print(res)
    Except ZeroDivisionError:
        Print('the val for b should not be zero')
    except ValueError:
        Print('vals should be int')
div()
```

Problem statement

solution

solution

Specific Syntax:

```
try:
    Problem statement
except <ExceptionName>:
    solution
```

Generic Exception Handling:

This is the type of Exception handling where it is capable of handling all the kind of Exception Except keyboard Intreput.

→ when we don't know the type of Error then we can use generic Exception.

Syntax:

```
try:
    Problem statement
Except Exception:
    solution
```

Eg:

```
def div():
    try:
        a = int(input('Enter the val:'))
        b = int(input('Enter the val:'))
        res = a/b
        Print(res)
    Except Exception:
        Print('Exception Handled successfully')
div()
```

Eg: O/P: 12/'hai'
0/2
Exception Handled successfully

Eg:

```
try:
    i=1
    while True:
        Print(i)
        i+=1
Except Exception:
    Print('Exception handled successfully')
```

Op:
1
2
3
4
5
:
:

infinity num until we stop / kill the Execution

Keyboard Intrreput : Error

3. Default Exception Handling:

It is an Exception handling process where it is capable of handling all the type of Exception.

Syntax:

```
try:
    Problem statement
Except:
    solution
```

Eg:

```
try:
    i=1
    while True:
        Print(i)
        i+=1
except:
    Print('Exception Handled successfully')
```

Eg:

By using three types of handling:

```
def div():
    try:
        a=int(input('Enter the val:'))
        b=int(input('Enter the val:'))
        res=a/b
        Print(res)
    except ZeroDivisionError:
        Print('b should not be 0')
    except Exception:
        Print('Exception Handled successfully')
    except:
        Print('Exception Handled successfully')
div()
```

Coustum Exception :

In Python users also can throw the Exceptions based on their requirments and that can be done with the help of a syntax.

Syntax :

```
raise <Exc Name>('msg')
```

Eg: where Exception name should be the standard Exception or inbuilt Exception and printing the msgs are not mandatory.

```
Eg: a=10
    b=20
    if a>b:
        Print('a is greater')
    Else:
        raise ZeroDivisionError('b is greater')
```

Print msg is not mandatory

Be Exception name that should be inbuilt in by the developer

Assert : (Keyword)

We can do coustom Exception with the help of assert keyword as well.

Syntax

```
assert <cond>, 'msg'
```

statements