20/08/25

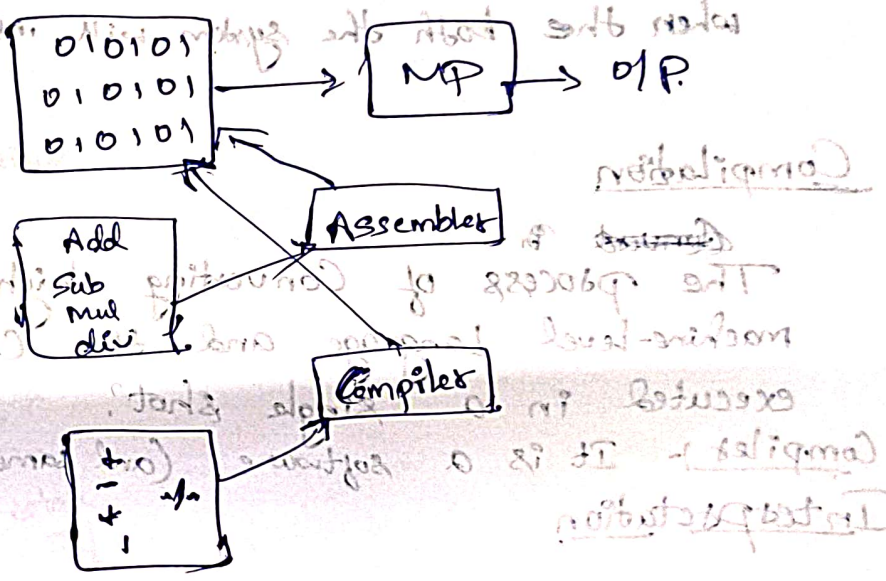
Python

Python is a **Interpedent** and high-level programming language.
↳ line by line execution.

Features of Python

1) * High level programming Language.
2) * Platform Independent
3) * Portable.
4) * Dynamic in nature.

①



② Platform:
Combination of operating system and Microprocessor.

Platform Independent

The python code written in one platform can be used in any other platform.

Portability

Case 1
Same windows and same Microprocessor → dependent
in other device.

Case 2:
Same windows and diff't MP → No dp

In C program is dependent

In Python.
OS = Windows
MP = IS

OS = Windows
MP = AMD

when the both the system will install PVM → Python virtual machine.

Compilation

~~Control~~ B
The process of Converting high-level language into machine-level Language and the Complete Code will be executed in a single shot.
Compiler :- It is a software (and same as compilation)

Interpretation

The Proces of Converting high-level language into machine-level Language by line-by-line.
interpretor :- It is a software @ tool (some)

Python support both Compilation and interpretation.

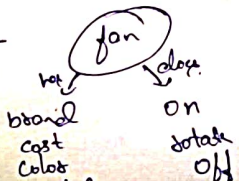
Python is a 3 dimenadion.

* OOPs
* Features
* Script.

→ oops :- Object orientation
Object → instance of class
→ Point of view
→ Perspective.
→ alignment

3 steps
1) Identify.
2) define
3) classify.
has → objects.
does → methods

Programs



Camel Conversion method. (starting letter will small).

class Fan:
brand
Cost
Color
numofBlades
on()
rotate()
off()

⇒ method will declare as def

class Fan:
brand: "usha"
Cost: "3000"
Color: "white"
numofBlades: 3
def one()
def rotate()
def off()

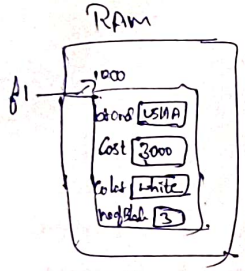
Note :- To initilize the Python Program. in oops we have to declare the Constructor.

def __init__(self): → called as Constructor
↳ variable. → Special method.
→ init method.

Indentation : Used to start and end the instruction.
↳ also called as 4 spaces (or) 1 tab

Creation of an object

Note :- we have to create a object outside the class.



Program of Fan :

```
class Fan:
    def __init__(self):
        self.brand = "usha"
        self.cost = 3000
        self.color = "white"
        self.noofBlades = 3
    def on(self):
        Print("The fan is ON")
    def rotate(self):
        Print("The fan is rotate")
    def off(self):
        Print("The fan is off")
f1 = Fan()
Print(f1.brand)
Print(f1.cost)
Print(f1.color)
Print(f1.numofBlades)
f1.on()
f1.rotate()
f1.off()
```

Output

```
usha
3000
white
3
The fan is on
The fan is rotate
The fan is off
```

Note :- ① Instead of (self) we can use any other word, but (self) is the standard way.

Steps to Creation of Object

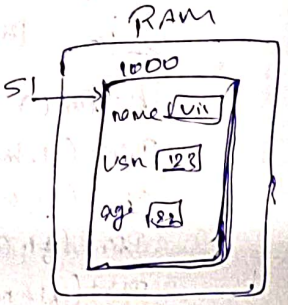
① PVM create a separate block of memory in the RAM with an address.

② PVM Its searchs for a special method called init method once it found the init method the address of the Object is stored in self keyword. Later the data will be stored in the separate block of memory.

③ The address of the object will be stored in one reference variable and the reference variable is Pointing out to the same memory Location.

24/3/25

Student




```
class Student:                          (1000)
    def __init__(self):
        self.name = "vishuxx"
        self.usn = 123
        self.age = 22.

    def study(self):
        Print("vishuxx is not running")

S1 = Student()                          (1000)
Print(S1.name)
Print(S1.usn)
Print(S1.age)
S1.study()
```

Organization of RAM

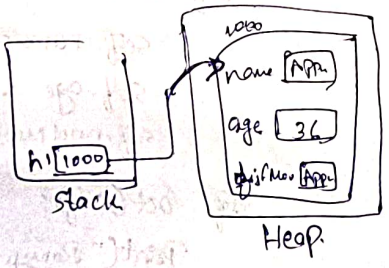
1) Code Segment → Stored the entire program.
2) Static Segment → Stored the static variable
3) Stack Segment → Local variables.
4) Heap Segments → Object

Actual Creation of an Object

Program Hero

```
Class Hero
    def __init__(self):
        self.name = "Appu"
        Self. age = 50
        Self. firstMovie = "Appu"

    def act(self):
        Print("Appu is a good Dancer")

h1 = Hero()
Print(h1.age)
Print(h1.name)
h1.act()
```

(Ctrl)



26/3/25

Steps :-

1) PVM creates a separate block of memory in a heap segment with an address
2) It search for a special method called __init__ method once it found the init method the address of the object is stored in self keyword.
   Later the data will be stored in the separate block of memory.
3) The address of the object will be stored in one reference variable and that variable will be stored in stack segment and that is pointing out to the same memory location.

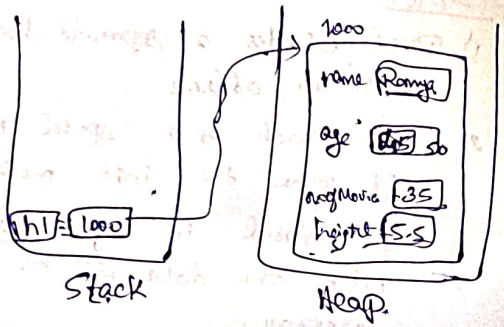
Note :- ① If we want to declare or accesses anything inside the class we have to use self keyword.

② If we want to declare or access anything outside the class we have to use reference variable.

Program Heroine :-

```
class Heroine:
    def __init__(self):
        self.name = "Ramya"
        self.age = 45
        self.noofMovie = 38
    def act(self):
        Print("Ramya is a versatile actress")

h1 = Heroine()
Print(h1.name)
Print(h1.age)
Print(h1.noofMovie)
h1.act()
h1.height = 5.5
Print(h1.height)
del h1.height
h1.age = 49
Print(h1.age)
```

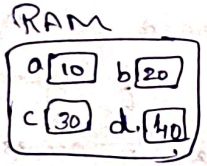
Instance variable



del → it is a keyword used to delete an variable

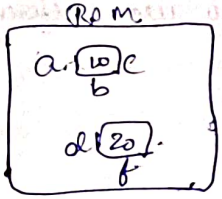
Data Types and Variables

a=10
b=20
c=30
d=40



variable is a container that stores a value.

a=10
b=10
c=10
d=20
f=20



data types

1) Numeric data types → int, float
2) String d.t → str
3) Boolean d.t → bool
4) Mapping d.t → dict
5) Sequence d.t → list, tuples, range
6) bin.array d.t → array, arraybytes

Note:- In Python the data size is not fixed

Rules for naming the variable

1) The variable name should starts with either underscore (_) or character.
   Ex :- _name (or) name.
2) variable names can contain A to Z, a to z, then 0 to 9
   Ex: Name, name, name1, name 8.
3) The variable name should not contain only digits
   Ex: 1234 it is not a variable name.
4) Any keywords are not allowed.
5) Any special characters are not allowed. accept (_) underscore
6) space is not allowed in between the variable name.
   Ex: ~~name~~ na me ×
7) Variable are case sensitive.

Ram will store the byte value

22/8/25

Example:- of Types

```
a = 10
Print(a)
Print(type(a))
```
output: <class 'int'>

```
b = 11.5
Print(b)
Print(type(b))
```
output: <class 'float'>

```
c = "thishwas"
Print(c)
Print(type(c))
```
output: <class 'str'>

Note:- while checking the type of particular variables it will give that data types along with class. because in Python every variables is Consider as an object.

In Python we can take the input from the user by the keyword input(). in PVM will accept the input as the string.

Ex: Program addition.py.

```
a= int (input("Enter the value"))
b= int (input ("Enter the value"))
c= a+b
Print(c)
```

Conditional Statement:-

In Python only 3 Conditional statement

1) if → used to check True or false.
2) elif → used when multiple statement
3) else → all the statement are false. the else is used.

Program greater.py

```
a = int (input(" Enter the value "))
b= int ( input (" Enter the value"))
if (a>b):
    Print (" A is greater ")
else(b>a):
    Print ("B is greater")
else:
    Print (" both are equal")
```

Looping Statement:-

In Python we have only 2 looping statement

1) for
2) while

→ For loop

Syntax:- for variable_name in list_of_elements:

Examples:

①
```
for i in 1,2,3,4:
    Print (i)
```
output:
1
2
3
4

②
```
for i in 10, 11.5, "Rama":
    Print (i)
```
output:-
10
11.5
Rama

③
```
for i in "Rama":
    Print (i)
```
output:
R
a
m
a

④
```
for i in "Rama":
    Print ( i, end=" ")
```
output: Rama

end = " " it is used print the output on same line

Range Function range()

Range is not a looping statement but it can be used in for loop

① range (stop) → last value, excluded.
② range (start, stop) dv→0
③ range (start, stop, step) → dv→4.

Example: ① for i in range (100):
Print (i)

o/p: 0
1
2
...
99

② for i in range (1,100):
Print (i)

o/p: 1
2
3
...
99

③ for i in range (1,100,2):
Print (i)

o/p: 1
3
5
...
99

④ for i in range (2,100,2):
Print (i)

o/p: 2
4
6
8
...
98

→ While loop

It can be used when we donot known the endpoint while loop can be end when it's condition become false.

① i=0
while (i<2)
Print("Hello")
i=i+1;

output: Hello
Hello
Hello

High Cohension
Low Cohension

High Cohension :- If properties and behaviors and methods are related to the object is called as high cohension
High Cohensive programs.

Low Cohension : If properties and behaviors and methods are not related to the object is called as low cohension
Low Cohensive programs.

High

class Fan:
brand
color
cost
def on
def off
} High

Low

class demo:
a=10
b=20
def function():
...
} not related.
Low Cohension.

Various ways to declare instance variable

Program Mobile

class Mobile:
def __init__(self):
self.brand = "Nokia"
self.cost = 1999
} 1st type of declare of Instance variable Standared way.
def work(self):
self.charge = 90
Print(self.charge)
} Second type.

m1 = Mobile()
Print(m1.brand)
m1.work()
m1.Color = "Black" } 3rd Type
Print(m1.Color)

1) Inside the class, inside the Constructor
2) Inside the class inside the method
3) Outside the class

To Create the realtime data and time.

```
from datetime import datetime
dt = datetime.now()
date = dt.strftime("%d-%m-%y")
```

10-04-25

%y → create full → 2025

%y → 25

```
time = dt.strftime("%I : %M %p")
```

10: 50 AM.

M → ~~month~~ Minute

m → month.

To print 24 hours time frame.

```
time = dt.strftime("%I : %M")
```

10 : 50

Note:- append → add the element at last of tuble or list

Data Size

* In Python ~~there~~ Data size is not fixed.

1 bit → 0 (or) 1

2 bits → 00, 01, 10, 11 — 0 to 3 — 0 to 2^2-1

3 bits → 000, 001, ..., 111 — 0 to 7 — 0 to 2^3-1

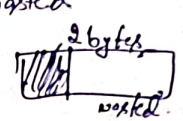
4 bits → 0000, ..., 1111 — 0 to 15 — 0 to 2^4-1

8 bits → ... — 0 to 255 — 0 to 2^8-1

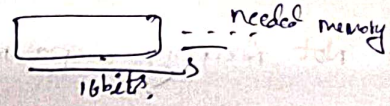
16-bits → 0 to 65535 — 0 to $2^{16}-1$

The disadvantage to fixed data types.

1) If a = 5, the 2 bits of memory is allocated, the other memory space will wasted



2) If we can extand the value. the memory will not consider the more value other than fixed bytes.



:- Methods :-

Syntax:- def method_name (self):

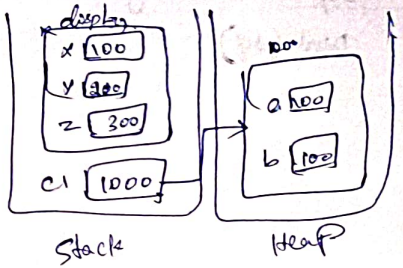
4 types of Methods — Instance Method, Static Method, Class Method

In Instance Type: 4 types

1) Accept Parameter, Returning a value
2) Accept Parameter, No Return value
3) No parameter, Returning a value
4) No parameter, No Return value.

→ 1) No parameter. No Return value.

```
class Calculator:
    def __init__(self):
        self.a = 100
        self.b = 200
    def display(self)
        x = 100
        y = 200
        z = x + y .
```

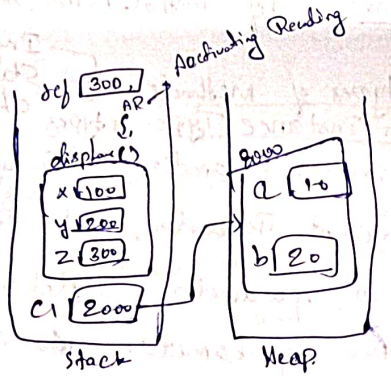


```
Print(z)

c1 = Calculator()
Print(c1.a)
Print(c1.b)
c1.display()
```

After executing of display method that will erase from the Stack segment

Type 2:
The method is a Not Accepting a Parameter but it returning a value.

```
class Calculator:
    def __init__(self):
        self.a = 10
        self.b = 20
Instance method { def display(self):
        x = 100
        y = 200
        z = x+y
        return z
```




```
300 c1 = Calculator()
Print(c1.a)
Print(c1.b)
def = c1.display()
Print(def)
```

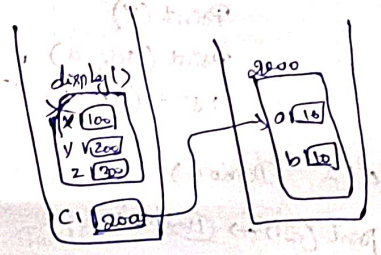
Type 3: The method is Accepting a Parameter but not Returning a value

Instance method
Static method
Class method

```
class Calculator:
    def __init__(self):
        self.a = 10
        self.b = 20
    def display(self, x, y):
        z = x+y
        Print(z)

c1 = Calculator()
Print(c1.a)
Print(c1.b)
c1.display()
x = 100
y = 200
c1.display(x, y)
```

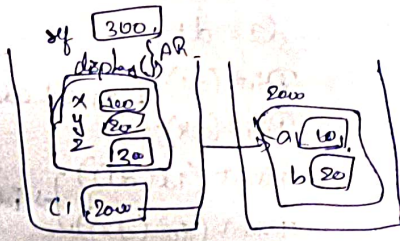
} Instance-variable.

} local variable



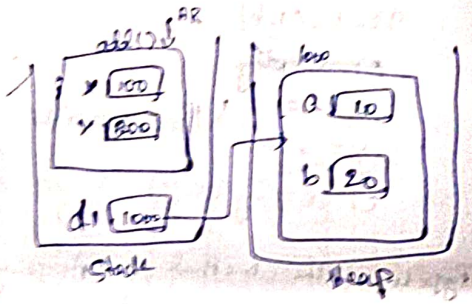
Type 4: The method is Accepting a Parameter and Returning a value

```
class Calculator:
    def __init__(self):
        self.a = 10
        self.b = 20
    def display(self, x, y):
        z = x+y
        return(z)

c1 = Calculator()
Print(c1.a)
Print(c1.b)
n1 = 100
n2 = 200
def = c1.display(n1, n2)
Print(def)
```



Program Demo:

```
class Demo:
    def __init__(self):
        self.a = 10          } Instance variable.
        self.b = 20
        Print(self) -----> output (1000)
    def add(self):
        x = 100              } local variable.
        y = 200
        Print(x)
        Print(y)
        Print(self.a)        } -> These can be used, when the Instance
        Print(self.b)            variable can used in the method.
```




```
d1 = Demo()
Print(d1) -> (1000) output
Print(d1.a)
Print(d1.b)
d1.add()
ns = ss   } local var
Print(ns)
d2 = d1
Print(d1)
Print(d2)   -> Print addr of an object
Print(id(d1))
Print(id(d2))   -> id
Print(d1 is d2)   output = True
```

why the id will change once run the code?

Because the PVM will considers the class name @ Code as a new code that's way, id will change when run's the code

To check the id is assign the same id.

```
a = 10
b = 10
c = 10
Print(a)
Print(b)   -> outputs 10
Print(c)
```

a, b → 10 ; c → 20

To compare we can use 'is' operator

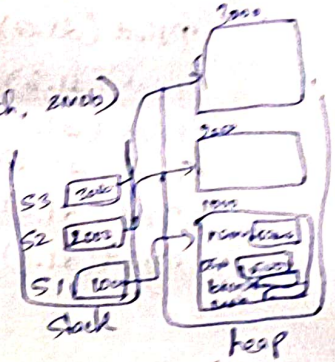
```
Print id(a)
Print(id(b))   -> 12345
Print(id(c))
Print(a is b) -> True
Print(b is c) True.
```

```
d = 30
e = 30
f = 20
Print(d)
Print(e)   -> 30
Print(f)
id(d)
id(e)   -> same addr 0x7320...
id(f)
Print(d is a) -> False
Print(f is d) -> False
Print(e is c) -> False
```

12/12

Program: To print n number of student data.

```
class Student:
    def __init__(self, sname, susn, sbranch, smob):
        self.sname = sname
        self.susn = susn
        self.branch = sbranch
        self.mob = smob
```

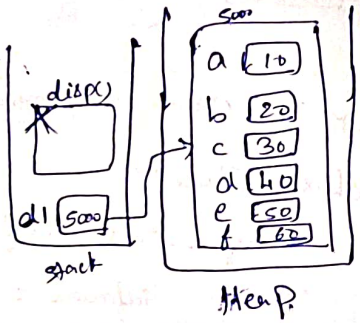



```
S1 = Student("sonna", "1VTU01", "CSE", 1234)
S2 = Student("Siba", "1VTU02", "CSE", 98887)
S3 = Student("vishwa", "1VTU03", "CSE", 97873)
```

Various ways to declare an instance variable

Program:

```
class Demo:
    def __init__(self):
        self.a = 10
        self.b = 20          # I method within the class within the Constructor
    def disp(self):
        self.c = 30          # II Method within the class within the Method
        self.d = 40
        Print(self.a)
        Print(self.b)

dl = Demo()
dl.disp()
dl.e = 50                    # III method outside the class
dl.f = 60
Print(dl.a)
Print(dl.b)
Print(dl.c)
Print(dl.d)
Print(dl.e)
Print(dl.f)
```

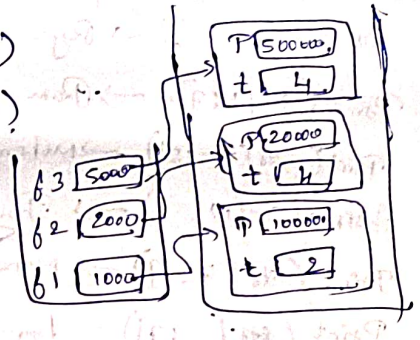


Static variable

* To static variable can be initilize with in the class outside the all the methods,
* To Improve the system performance we can use static variable.
* We can use the static variable by prefix -> use classname

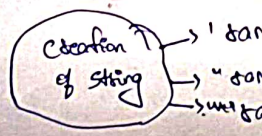
Program Farmer to find Simple Interest

```
class Farmer:
    r = 2.5
    def __init__(self, p1, t1):
        self.p = p1
        self.t = t1
    def disp(self):
        SI = (self.p * self.t * Farmer.r) / 100
        Print(SI)

f1 = Farmer(100000, 2)
f2 = Farmer(200000, 4)
f3 = Farmer(500000, 4)
f1.disp()
f2.disp()
f3.disp()
```



18/04/25

String in Python :- Collection of character with enclose with Codes (" ")



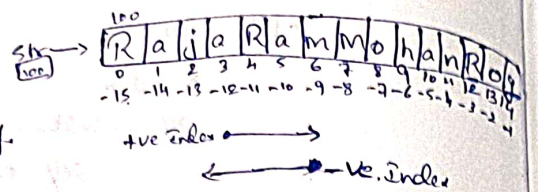

```
Str = "rama"
Print(Str) ~> rama
Print(len(Str)) ~> 4
Print(Str[0]) ~> r
Print(Str[2]) ~> m
```

Str 1000 → 1000 | r | a | m | a | (0 1 2 3)

Indexing and Slicing: (Creating the string into parts)

Program String

```
str = "RajaRamMohanRoy"
```

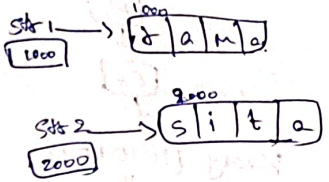



```
Print (str)  ⇝ RajaRamMohanRoy.
Print (len (str))  ⇝ 15
Print (str [0])  ⇝ R
Print (str [7])  ⇝ M.
Print (str [14])  ⇝ y
Print (str [-1])  ⇝ y
Print (str [-6])  ⇝ h.
Print (str [0:4])  ⇝ Raja
Print (str [4:7])  ⇝ Ram
Print (str [7:12])  ⇝ Mohan
Print (str [12:])  ⇝ Roy.
Print (str [7:])  ⇝ MohanRoy
Print (str [:7])  ⇝ RajaRam
Print (str [:])
Print (str [::-1])  [Step value given as -1] [Reverse string]   (output)
Print (str [9:3])  ⇝ No o/p
Print (str [3:9:-1])  ⇝ No o/p
Print (str [9:3:-1])  ⇝ hoMmaR
Print (str [::10:-1])  ⇝ yoRn
Print (str [-5::-1])  ⇝ ahoMmaRajaR
Print (str [-8:-3])  ⇝ Mohan
Print (str [-4:-9:-1]  ⇝ nahom.
Print (str [::3])  ⇝ RamhR
Print (str [5:12:0])  ⇝ Error
```

19/04

* Arithmetic Operations on Strings: (+, -, *, /)

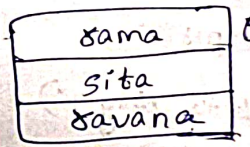
```
str1 = "rama"
str2 = "sita"
str3 = str1 + str2.
Print (str3)  ⟶ ramasita
str4 = str1 - str2.
Print (str4)  ⇝ Error
str5 = str1 * str2
Print (str5)  ⟶ Error
str6 = str1 / str2.
Print (str6)  ⟶ Error
str7 = str1 * 2.
Print (str7)  ⟶ ramarama
```

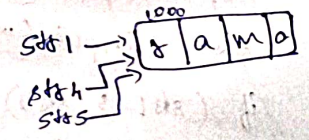


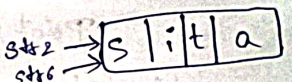
id → Runtime address.
address → Permanent address.

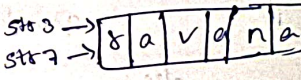
* String Interning:

```
str1 = "rama"
str2 = "sita"
str3 = "ravana"
str4 = "rama"
str5 = "rama"
str6 = "sita"
str7 = "ravana"
```







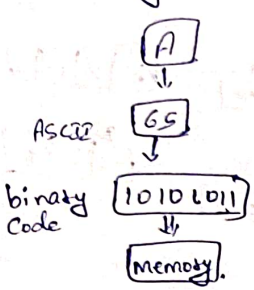



```
Print (id (str1))
Print (id (str4))
Print (id (str5))
Print (id (str2))
Print (id (str6))
Print (id (str7))
Print (id (str3))
```

Print (hex (id (str1))) → address of in hexadecimal form.

x.10x1

Extracting ASCII value of a character.



A → 65
B → 66
C → 67
⋮
Z → 90

a → 97
b → 98
⋮
z → 122.

```
Print ("Enter a character")
str = input()
str1 = ord (str)
Print ("The ASCII value is", str1).
```

[ord() is a function / keyword that extract the ASCII value of char]
↳ character to ASCII value

String Comparision

```
Print ("Enter string 1")
str1 = input()
str2 = input ("Enter a string 2")
if (str1 > str2):
    Print ("string 1 is greater")
elif (str2 > str1):
    Print ("string 2 is greater")
else:
    Print ("string1 and string2 both are equal")
```

C1 : str1 → rama 114 97 109 117
str2 → rama 114 97 109 97
0 0 0 +20

C2 : str1 → rama 114 97 109 97
str2 → ramu 114 97 109 117
0 0 0 -20

C3 = str1 → rama 114 97 109 97
str2 → rama 114 97 109 97
0 0 0 0

Write a Program to Convert uppercase to lowercase without using any inbuilt function.

* Pseudo Code

```
Accept the input
i=0
newstr = " "
as long as (string has characters):
    Extract the character at ith index
    Extract the ASCII value of a character
    if (ASCII value in range of 65 to 90):
        add +32 to ASCII value.
        Convert the ascii value to character
        add Convchar to newstr
    else:
        add the extracted character to newstr.
    i=i+1
Print the newstr variable.
```

chr() ↓ Convert ascii to character

i = 0 1 2 3 4

str: R a M a
data: R a M a
ascii: 82 97 77 97
newascii: 114 109
Convchar: r m
newstr: r a M a

Program

```
Print ("Enter a string")
str = input()
i=0
newstr = " "
while ( i <= len(str)-1):
    data = str[i]
    ascii = ord (data)
    if (ascii >= 65 and ascii <= 90):
        newascii = ascii + 32
        Convchar = chr (newascii)
```

```
        newstr = newstr + Convchar.
    else:
        newstr = newstr + data
    i = i+1
Print(newstr)
```

29/11

→ Write a program to Convert lowercase to Uppercase:

```
Print("Enter a String")
str = input()
i = 0
newstr = "    "
while (i<= len(str)-1):
    data = str[i]
    ascii = ord(data)
    if (ascii >=97 and ascii<= 122):
        newascii = ascii-32.
        Convchar = chr(newascii)
        newstr = newstr + Convchar
    else:
        newstr = newstr + data.
    i = i+1
Print(newstr)
```

i = 0

str | R | a | M | a

data | R

ascii | 82

newascii

Convchar

newstr | R | A | M | A

input: RaMa
output: RAMA

→ Write a program to Convert Lower → Upper and Upper → Lower [Swap Case]

```
str = input("Enter a String")
i=0
newstr = "  "
while (i<= len(str)-1):
    data = str[i]
    ascii = ord(data)
    if (ascii >=65 and ascii<=90):
        newascii = ascii + 32.
        Convchar = chr(newascii)
        newstr = newstr + Convchar.
    else:
        newascii = ascii - 32
        Convchar = chr(newascii)
        newstr = newstr + Convchar
    i = i+1
    Print(newstr)
```

i = 0 to 2 & 4.

str | R | a | M | a

data | R | a | M | a

ascii | 82 | 97 | 77 | 97

newascii | 114 | 65 | 109 | 65

Convchar | &AmA

newstr | & | A | m | A

By Inbuild function we can Perform Lower, Upper, Lower and Swap Case:

str → | R | a | M | a

Lower

```
str = input("Enter the string")
str1 = str.lower()  ⇝ rama
str2 = str.upper()  ⇝ RAMA
str3 = str.swapcase()  ⇝ &AmA
Print(str1)
Print(str2)
Print(str3)
```

jeevan@pentagonspace.in

* Finding Substring from a string:

```
str = "If you think you can or you can't, you are right"
Print(str)
Print(str.find("you"))  → 3
Print(str.index("you"))  → 3
```

str1 = "you"

Print(str1 in st) → True.

Print(str.count("you")) → 4

Print("str.find("you")) → 35

Print("str.index("you")) → 35

Print("str.find("rama")) → -1

Print("str.index("rama")) → Error.

25/04/25

Functions:

The function is a block of reuseable code which Performs specific tasks whenever it is called.

Syntax:-

```
def function_name(Parameters):
    ||||
function_name()
```

Ex1

```
def add():
    a=10
    b=20
    c=a+b
    Print(c)
add()
```

Difference b/w methods and functions.

```
def fun1():
    Print("Python")
fun1()
class Demo:
    def __init__(self):
        self.a=10
    def call(self):
        Print(self.a)
d=Demo()
d.call()
```

Output

```
Python
10
```

Types of Function

1) The function donot accept any Parameter and does not have any return value.
2) The function which donot accept any Parameter and as the return value.
3) The function which accept the Parameter and does not have any return value.
4) The function which accept the parameter and as the return value.

⟹

```
def function1():
    a=50
    b=60
    c=a+b
    Print(c)
```

```
def fun2():
    a = 50
    b = 60
    c = a+b
    return c

def fun3(a,b):
    c = a+b
    Print(c)

def fun4(a,b):
    c = a+b
    return(c)

function1():
    ret1 = fun2()
    Print(ret1)
    x = 50
    y = 60
    fun3(x,y)
    ret2 = fun4(x,y)
    Print(ret2)
```

→ function call

output
110
110
110
100

Default and Keyword Arguments

```
def fun1(a=50, b=60, c=70):
    Print(a)
    Print(b)
    Print(c)

x = 11
y = 22
z = 33

fun1(x,y,z)  ⟶ 11, 22, 33
fun1(x,y)    ⟶ 11, 22, 70
fun1(x)      ⟶ 11, 60, 70
fun1(z)      ⟶ 33, 60, 70
fun1(c=z)    ⟶ 50, 60, 33
fun1(b=y, c=z) ⟶ 50, 22, 33
```

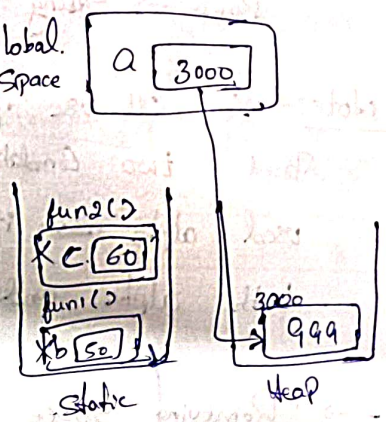
Default arguments

Keyword Arguments

Local and Global Variable.

```
a = 999          ⟶ Global Variable.
def fun1():
    Print(a)
    b = 50       ⟶ local variable. Global.
    Print(b)
def fun2():
    Print(a)
    c = 60
    Print(c)

Print(a)
fun1()
fun2()
```

output
999
999
50
999
60



23/04

```
print("Enter a String")
str = input()
if (str.isalpha()):
    print("String Contains only alphabets")
elif (str.isdigit()):
    print("String Contains only numbers")
elif (str.isalnum()):
    print("String Contains both char and num")
else:
    print("Enter valid string")
```

Results

Rama => String Contains only Alphabets

Note:- You will use if Condition every time it will ~~print~~ Print two Conditions ~~Every time~~ : like you will used alpha → it will shows alpha Condition and both alpha and numbers

→ Removing Space at Beginning and Ending of a String

```
str = "    Rama is Dancing    "
print(str)
str1 = str1.lstrip() ⟶ Left strip
str2 = str.rstrip() → Right strip
str3 = str.strip() → Both side
print(str1)
print(str2)
print(str3)
```

* Removing the Spaces on the string

```
print("Enter the string")
str = input()
i=0
newstr = ""
while (i<=len(str)-1)
    data = str[i]
    if (data == " "):
        i = i+1
    else:
        newstr = newstr + data
        i = i+1
print(newstr)
```

i=0 1 2 3 4 5 6 7

str 1000 → 1000 | R | a | M | a |

0 1 2 3 4 5 6

Result:

R a M a

RaMa

data R_a_M_a

newstr RaMa

Another Method

```
print("Enter the string")
str2 = input()
str1 = "".join(str.split()) / str1 = str.replace(" ", "")
print(str1)
```

Result:→ R a M a → Input

↳ RaMa → output

* String Formatting

```
Print ("Enter a string")
str  = input ()
Print ("Hi {}".format (str))
Print ("Name ={}".format (str))
str = "Rama"
Print (str)
Print (str[0]) ---> R
Print (str[4]) ---> n
Print (str[5]) ---> a
str [4] = "m" ---> Error
str1 = str.replace ("a", "o")
Print (str1)
```

Formal and Actual Parameters

```
class Demo:                    ↦ Formal Parameters
    def disp(self,a, b)
        temp = a
        a = b
        b = temp

d1 = Demo ()
x = 100
y = 200
Print ("Before Swap")
Print (x)
Print (y)
d1. disp (x, y)  → Actual Parameters (Arguments)
Print ("After swap")
Print (x)
```

Default and Keyword Argument

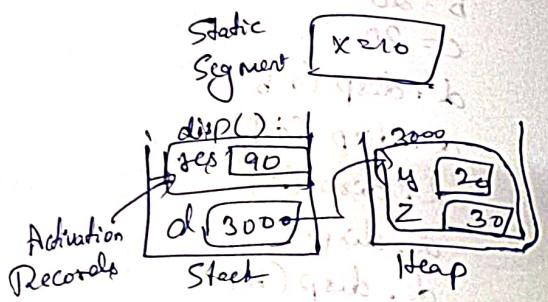
```
class Demo:
    def disp (self, x=11, y=22, z=33)   → default Argument
        Print (x)
        Print (y)
        Print (z)

d = Demo ()
a = 10
b = 20
c = 30
d.disp (a, b, c)
d..disp (a, b)
d. disp (a)
d. disp ()
d. disp (b)
d. disp (b, c)
d. disp (z=c)
d.disp (y=b, z=c)        Keyword Argument
d.disp (x=a, y=b, z=c)
```

String Replace

```
str = "Rama is drinking"
Print (str)
str1 = str.replace ("is", "was")
Print (str1)
Print (str.startswith ("Rama")) ---> True
Print (str.endswith ("drinking")) ---> True
Print (str.startswith ("Rakesh")) ---> False
```

Static Variable

```
class Demo:
    x = 10
    def __init__(self):
        self.y = 10
        self.z = 20
    def disp(self):
        Print(self.y)
        Print(self.z)
        ses = 40+50
        Print(ses)

d = Demo()
d.disp()
Print(Demo.x)
```



output:

```
y = 20
z = 30
ses = 90
x = 10
```

Static and class Method

```
class Demo:
    x = 10
    def __init__(self):
        self.y = 20
        self.z = 30
    def disp(self):
        Print(self.y)
        Print(self.z)
```

@ staticmethod

```
def static_disp():
    Print(Demo.x)
    Demo.x = 11
    Print(Demo.x)
```

@ Class Method

```
def class_disp(cls):
    Print("Python")

d = Demo()
d.disp()
Demo.static_disp()
Demo.class_disp()
```

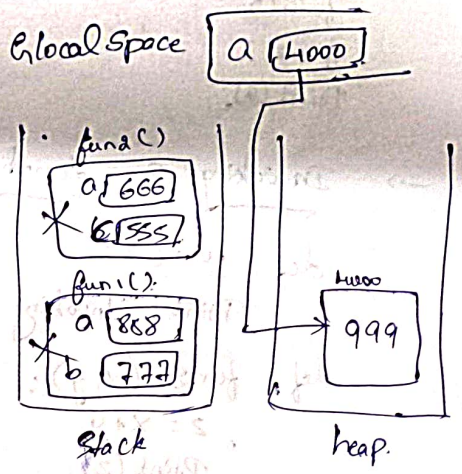
Result

```
20
30
10
11
Python
```

28/04

Global Variable

```
a = 999
def fun():
    a = 888
    b = 777
    Print(a)
    Print(b)
def fun2():
    a = 666
    c = 555
    Print(a)
    Print(c)
```



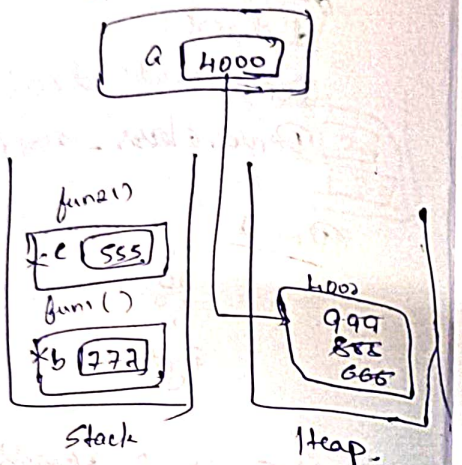
```
fun1()
fun2()
Print(a)
```

output : 999
888
777
666
555
999

→ Access and Modify Global Variable.

```
a = 999
def fun1():
    global a
    a = 888
    b = 777
    Print(a)
    Print(b)
def fun2():
    global a
    a = 666
    c = 555
    Print(a)
    Print(c)
```

o/p
999
888
777
666
555
666




```
Print(a)
fun1()
fun2()
Print(a)
```

→ Invoking a function through variable Assignment.

```
def fun1():
    Print("Invoking fun1")
def fun2(x,y):
    z = x+y
    Print(z)
```

```
fun1()
a = 10
b = 20
fun2(a,b)
```

o/p :
Invoking fun1
Sum is 30

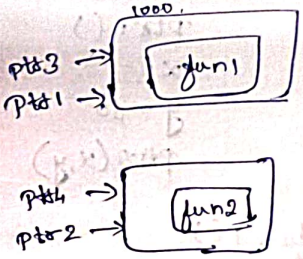
```
ptr1 = fun1
ptr2 = fun2
Print(ptr1) → address
Print(fun1) → same address
ptr1()
ptr2(40,50)
```

Invoking fun1
Sum is 90.

29/04

Passing function as arguments

```
def fun1():
    Print("Inside fun1")
def fun2(a,b):
    c = a+b
    Print("Sum is", c)
def display(ptr1, ptr2):
    ptr1()
    x = 50
    y = 60
    ptr2(x,y)
fun1()
fun2(10,20)
ptr3 = fun1
ptr4 = fun2
ptr3()
a = 30
b = 40
ptr4(a,b)
display(ptr3, ptr4)
```



Output.
Inside fun1
Sum is 30
Inside fun1
Sum is 70
Inside fun1
Sum is 110

→ Higher Order and First class Function

Higher order

The function which accept other function as a Parameter as refer to as higher order function.

* All the other functions are reffered to as fidst class function.

```
def fun1():
    Print("Inside fun1")
def fun2(a,b):
    c = a+b
    Print("Sum is", c)
```

=> First class Function

```
def display(ptr1, ptr2):
    ptr1()
    x = 50
    y = 60
    ptr2(x,y)
```

=> High-order Function.

```
fun1()
x = 40
y = 50
fun2(x,y)
ref1 = fun1
ref2 = fun2
display(ref1, ref2)
```

Output:

```
Inside fun1
Sum is 90
Inside fun1
Sum is 110
```

Lambda: * It is a argument function.
* Single line function.

Syntax:-

lambda arguments : expression

Ex:-

```
def fun1():
    return num * num.
```

lambda. num : num * num.

Ex1:-

```
s = lambda x : x * x
res = s(5)
Print(res)
```

Ex2:-

```
ptr1 = lambda x,y : x+y
res1 = ptr1(10,20)
Print(res1)
```

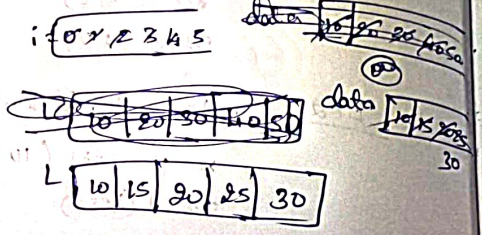
List:- Write a program to collect 5 integer value from the keyboard and store the integers in the list:

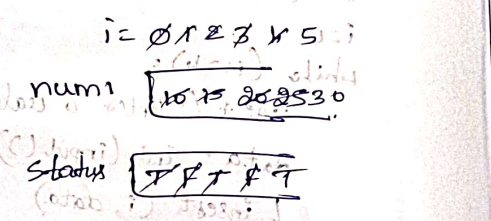
```
L = []
i = 0
while (i<=4):
    Print("Enter a value")
    data = int(input())
    L.insert(i, data)
    i = i+1
Print(L)
i = 0
while (i<=4):
    Print(L[i])
    i = i+1
for j in L:
    Print(j)
```

Print vertically.

30/04 Write a program to print only even numbers from a list.

```
def even(num):
    if (num % 2==0):
        return True.
    else:
        return False

L=[ ]
i=0
while (i<=4):
    Print ("Enter a value")
    data.=int(input())
    L.insert (i, data)
    i=i+1
Print(L)
i=0
while (i<=4):
    num1 = L[i]
    Status = even(num1)
    if (Status==True):
        Print (L[i])
    i=i+1
```





Output

```
10
20
30
```

→ Using filter function

The filter function is a process of extracting specific values from the list of elements based on specific conditions declared in the function.

Syntax:

```
filter (function_name, list_name)
```

Program

```
def even(num):
    if (num % 2==0):
        return True.
    else:
        return False.
L=[ ]
i=0
while (i<=4):
    Print ("Enter a value")
    data = int (input ())
    L. insert (i, data)
    i=i+1
Print (L)
k= list (filter (even, L))
Print(k)
```

k= filter(even, L) → given the address of filter

→ Using Lambda function

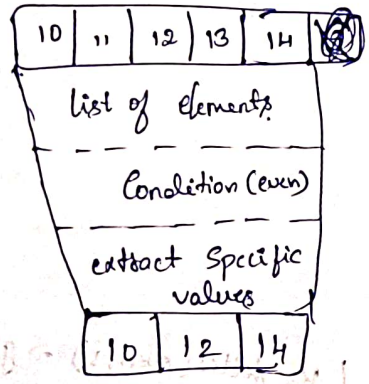
By using lambda inside the filter function. without having the even(num) function.

Program:

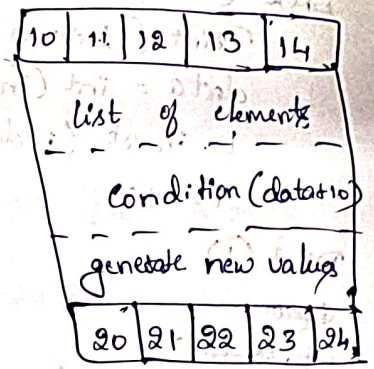
```
L = [ ]
i=0
while (i<= 4)
    data = int (input(" Enter a value "))
    L. insert ( i, data)
    i = i+1
Print (L)
k = list ( filter ( lambda num : (num % 2 == 0)), L))
Print (k)
```

01/05/25

Difference between filter() and map():



Filter()



~~Using m~~

Using map() function

map()

The map() function refers to applying a specific condition on each elements of list and convert the current values to new values using function.

Syntax:- map (function_name, list_name)

Program

```
def operation (data):
    return data+10
L=[ ]
i=0
while (i<=4):
    Print (" Enter a value")
    num = int (input ())
    L. insert (i, num)
    i = i+1
Print (L)
k = list (map (operation, L))
Print (k)
```

Using lambda function inside the map without having operation (data) function.

```
L [ ]
:
Print (L)
k= list (map( lambda data : (data +10), L))
Print (k)
```

Nested Function

The process of declaring a function inside another function is ~~reffed~~ reffred to as nested function.

Syntax:

```
def function_name():
    ========
    def function1_name():
        ========
        function1_name()
function_name()
```

Program

```
def out_fun():
    Print("Inside the outer fun")
    def in_fun():
        Print("Inside the inner fun")
    in_fun()
out_fun()
```

(outer 4000) (inner 3000)

08/05/25

Example:

```
def fun1():
    a=10
    Print("From fun1 a", a)
    def fun2():
        b=20
        Print("From fun2 b", b)
        Print("From fun2 a", a)
    fun2()
    Print("From fun1 a", a)
fun1().
```

O/P

```
From fun1 a 10
From fun2 b 20
From fun3 a 10
From fun1 a 10
```

②

```
def fun1():
    a=10
    Print("From fun1 a", a)
    def fun2():
        a=100
        b=20
        Print("From fun2 b", b)
        Print("From fun2 a", a)
    fun2()
    Print("From fun1 a", a)
    Print("From fun1 b", b)
fun1()
```

→ Inner function

→ outer function

→ Error

O/P

```
From fun1 a 10
From fun2 b 20
From fun2 a 100
From fun1 a 10
Error
```

Accessing non local variable and modifying the values:

```
def fun1():
    a=10
    Print("From fun1 a", a)
    def fun2():
        nonlocal a
        a=100
        b=20
        Print("From fun2 a", a)
        Print("From fun2 b", b)
    fun2()
    Print("From fun1 a", a)
fun1()
```

o/p

```
From fun1 a 10
From fun2 a 100
From fun2 b 20
From fun1 a 100
```

Types of Variables:

1) Global Variable
2) Static Variable
3) Instance Variable
4) local Variable
5) Nonlocal variable.

Standard way Declaring all the variables in a Single Program.

Program

```
a=10
def fun1():
    global a
    a=100
    b=20
    Print(a)
    Print(b)
def fun2():
    global a
    nonlocal b
    a=1000
    b=200
    c=30
    Print(a)
    Print(b)
    Print(c)
fun2()
Print(a)
fun1()
class Demo:
    x=11
    def __init__(self):
        self.y=22
        self.z=33
    def display(self):
        Print(self.y)
        Print(self.z)
        a=50
        b=40
        c=a+b
        Print(c)
d1=Demo()
Print(Demo.x)
d1.display()
Print(a)
```

o/p

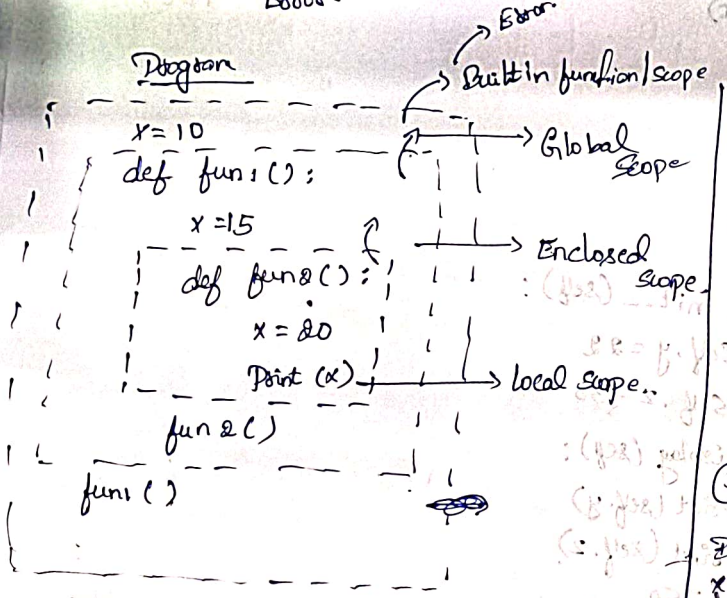
```
100
20
1000
200
30
1000
11
22
33
90
1000
```

3/5/25

LEGB Rule:-

First it search the variable in local scope. if variable is not there next search in Enclosed scope if variable is not found then search in the Global scope if variable is not found again it search in the built in scope. the variable is not in built in scope it shows an error.

Local scope
↓
Enclosed Scope
↓
Global Scope
↓
Built in Scope
↓
Error.



(1) o/p: 20

If Comment the x=20

(2) o/p = 15

If Comment the x=20 and x=15

(3) o/p = 10

If Comment the x=20 x=15 x=10

(4) o/p = Error.

Modules in Python :-

Program1.py

```
def fun1():
    Print("Inside fun1")
def fun2():
    Print("Inside fun2")
```

I

new.py

```
from program1 import fun1,fun2
fun1()
fun2()
```

By Replacing the , to import all the functions name like. fun1, fun2, fun3 - - - - - funnn just put the (*)

II

```
from program1 import *
```

III

Program1.py

```
def fun1():
    Print("Inside fun1")
def fun2():
    Print("Inside fun2")
```

new.py

```
import program1
Program1.fun1()
Program1.fun2()
```

Importing Multiple modules.

Program1.py

```
def fun1():
    Print("Inside fun1 in P1")
def fun2():
    Print(Inside fun2 in P1")
```

Program2.py

```
def fun1():
    Print("Inside fun1 in P2"):
def fun2():
    Print("Inside fun2 in P2)
```

new.py

```
from program1 import *
from program2 import *
fun1()
fun2()
fun1()
fun2()
```

o/p :-

```
Inside fun1 in P2
Inside fun2 in P2
Inside fun1 in P2
Inside fun2 in P2
```

The fun accept the recently importing file.

To overcome this output as import the file below that call the function.

new.py

```
from program1 import *
fun1()
fun2()
from program2 import *
fun1()
fun2()
```

new.py

```
import program1
import program2
Program1.fun1()
Program1.fun2()
Program2.fun1()
Program2.fun2()
```

o/p

```
Inside fun1 in P1
Inside fun2 in P1
Inside fun1 in P2
Inside fun2 in P2
```

5/5/25

```
def fun1():
    Print("Inside fun1")
def fun2():
    Print("Inside fun2")
def fun3(ptr1, ptr2):
    Print(ptr1)
    Print(ptr2)
    ptr1()
    ptr2()
Print(fun1)
Print(fun2)
ref1 = fun1
ref2 = fun2
Print(ref1)
Print(ref2)
fun3(ref1, ref2)
```

Closures in Python

The process of calling the inner function outside the scope of its definition is refered to as closures

```
def fun1():
    Print("I am outer")
    def fun2():
        Print("I am Inner")
        Print(fun2)
    return fun2
```

```
def1 = fun1()
Print(def1)
def1()
```

Output:-

I am Outer
Address
I am Inner
address

→ Decorator in Python

① The outer function which accept another function as a parameter and return inner function outside the scope of its definition.

② without changing the actual code, changing the behaviour of function is reflected to the decorator.

```
def outer(function):
    Print("I am outer")
    def inner():
        Print("I am inner")
    return inner.
def1 = outer(1)
Print(def1)
def1().
```

In Decorator's Python

```
def outer():
    Print("I am outer")
    def inner():
        Print("I am inner")
    return inner.
def1 = outer
Print(def1)
def1()
```

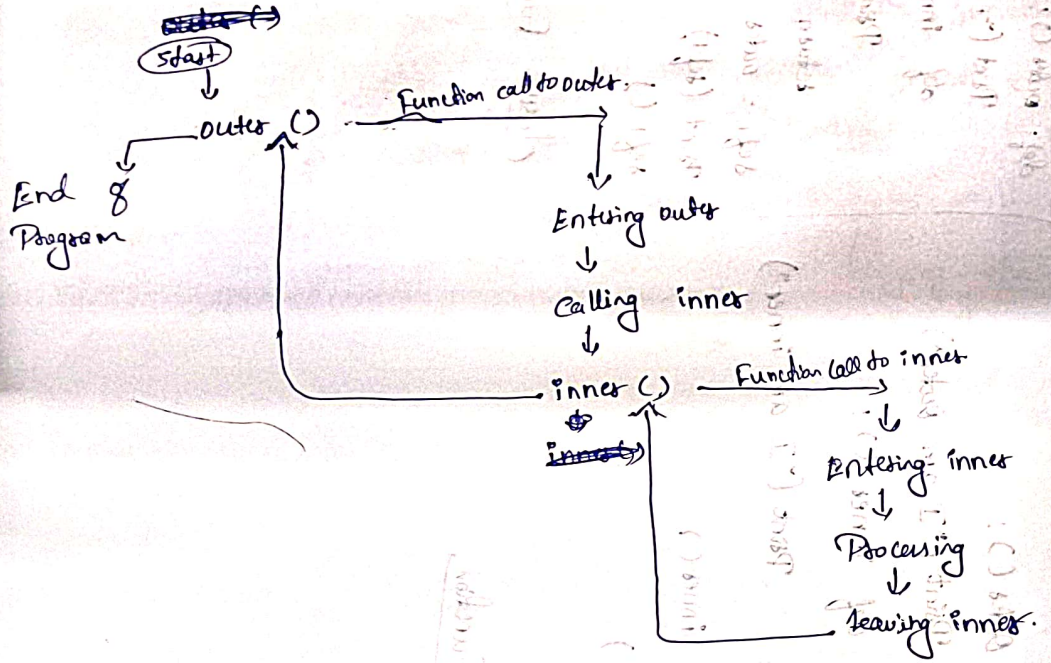
Closure Function.

```
def outer():
    Print("I am outer")
    def inner():
        Print("I am inner")
    inner()
outer()
```

Nested Function

Control flow of Nested Function:

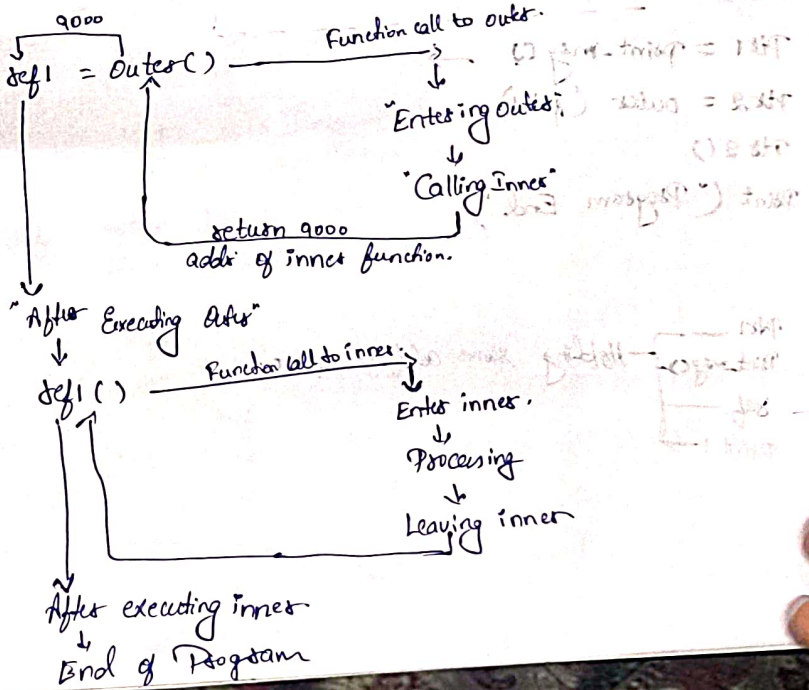
```
def outer():
    Print("Entering outer")
    def inner():
        Print("Entering inner")
        Print("Processing")
        Print("Leaving Inner")
    Print("Calling inner")
    inner()
outer()
```



6/5/- Control flow of closures:

```
def outer():
    Print("Entering outer")
    def inner():
        Print("Entering inner")
        Print("Processing")
        Print("Leaving inner")
    Print("Calling inner")
    return inner
def1 = outer()
Print("After Executing Outer")
def1()
Print("After Executing inner")
```

9000 — returning the addr of inner to outer ⇒ def1



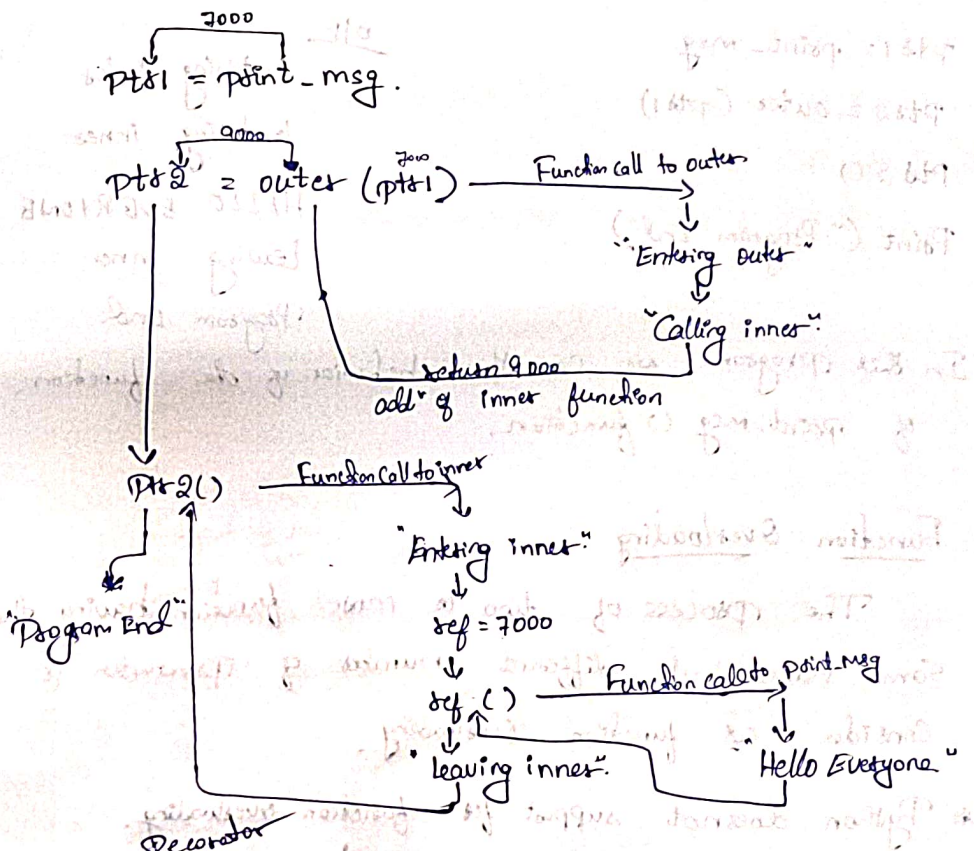
Control Flow of Decorators:-

① 7000
```
def Print_msg():
    Print("Hello Everyone")
```

8000
```
def outer(print1):
    Print("Entering Outer")
    9000
    def inner():
        Print("Entering Inner")
        def1 = Print1
        def()
        Print("Leaving inner")
    Print("Calling Inner")
    return inner
```

```
Ptr1 = Print_msg()
ptr2 = outer(ptr1)
ptr2()
Print("Program End")
```

Ptr1, Print_msg(), def, Print1 — Holding same address. 7000



gk

② Definition 2 — Decorator

```
def Print_msg():
    msg = "Hello Everyone"
    return msg

def outer(print1):
    Print("Entering Outer")
    def inner():
        Print("Entering inner")
        def = print1
        res = def()
        res1 = res.upper()
        Print(res1)
        Print("Leaving inner")
    return inner
```

```
ptr1 = print_msg.
ptr2 = outer(ptr1)
ptr2()
Print("Program End")
```

O/P

```
Entering outer
Entering inner
HELLO EVERYONE
Leaving inner
Program End
```

In this program we can the behavior of the function of print_msg() function

Function Overloading

The process of two or more function having the same name but different number of parameter is consider as function overloading.

* Python doesnot suppot for function overloading
* If the two or more function having the same name same number of parameter will give the output In this condition it will invoke the recent function that function is executed.

①
```
def fun1()
    Print("First Function")
def fun1(a)
    Print("Second Function")
def fun1(a,b)
    Print("Third function")
fun1()
```

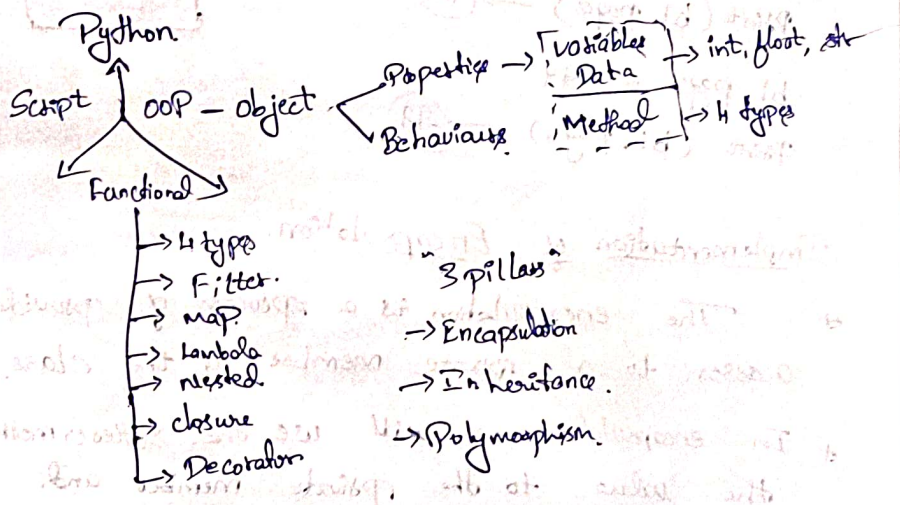
O/P

Error

②
```
def fun1():
    Print("First function")
def fun1():
    Print("Second function")
def fun1()
    Print("Third function")
fun1()
```

O/P

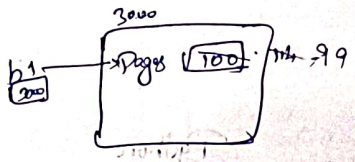
Third Function.



~~How to make~~
Encapsulation

how to make variable as private.

```
class Book :
    def __init__(self, value):
        self.pages = value

b1 = Book(100)
print(b1.pages) ---> 100
b1.pages = 114
print(b1.pages) ---> 114
b1.pages = -99
print(b1.pages) ---> -99
```



Implementation of Encapsulation.

* The encapsulation is a process of providing a control accesses to a private member of the class.
* In encapsulation we will use the setter() method to set the value to the private member and we will use the getter() method to access the value from the private member of the class.
* Encapsulation is data binding and it is not hiding the data.

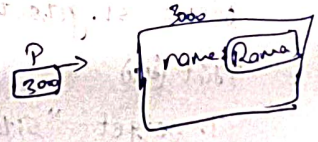
1)

```
class Book :
    def __init__(self, value):
        self.__pages = value
    def setter(self, value):
        if (value > 0):
            self.__pages = value
    def getter(self):
        return self.__pages

b1 = Book(100)
b1.setter(114)
res = b1.getter()
print(res) ---> 114
b1.setter(-99)
res1 = b1.getter()
print(res1) ---> 114
```

o/p
114
114

2)

```
class Person:
    def __init__(self):
        self.__name = " "
    def getter(self):
        return self.__name
    def setter(self, value):
        self.__name = value

P = Person()
P.setter("Rama")
res = P.getter()
print(res) ---> 'Rama'
```



09/05

Property Function in Encapsulation

```
class Student:
    def __init__(self):
        self.__name1 = "  "
        self.__name2 = "  "

    def getdata1(self):
        return self.__name1

    def setdata1(self, value):
        self.__name1 = value

    getset = property(getdata1, setdata1)

    def getdata2(self):
        return self.__name2

    def setdata2(self, value):
        self.__name2 = value

    setget = property(getdata2, setdata2)

s1 = Student()
s1.getset = "Rama"
res1 = s1.getset
print(res1)
s1.setget = "Sita"
res2 = s1.setget
print(res2)
```

- getdata1 / setdata1 } Put any function name
- no need to put only get/set

O/P

Rama
Sita

Property Decorator in Encapsulation

We can be property decorator when getter() and setter() function name is same.

Syntax:

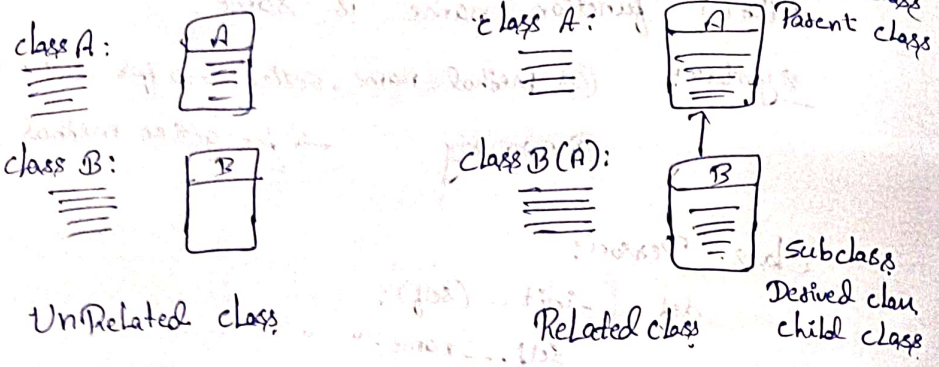
@ Method_name.setter → for setter method.

@ property → for getter method.

```
class Person:
    def __init__(self):
        self.__name = "  "

    @property
    def dataAccess(self):
        return self.__name

    @dataAccess.setter
    def dataAccess(self, value):
        self.__name = value

P = Person()
P.dataAccess = "Rama"
res = P.dataAccess
print(res)
```

→ Rama

Inheritance in Python



creating child class (or) Derived class

```
class Parent:
    def __init__(self):
        self.a = 10

class child(Parent):
    def __init__(self):
        Parent.__init__(self)
        self.b = 20

c1 = child()
Print(c1.b) ——— 20
Print(c2.b) ——— 10
```

Syntax:-

```
Parent = class_name.__init__(self)
```

```
class A:
    def __init__(self):
        self.a = 10

class B(A):
    def __init__(self):
        A.__init__(self)
        self.b = 20

class C(B):
    def __init__(self):
        B.__init__(self)
        self.c = 30

c1 = C()
Print(c1.c) ——→ 30
Print(c1.b) ——→ 20
Print(c1.a) ——→ 10
```

Return than using the parent class. now Constructor one can directly go it by.

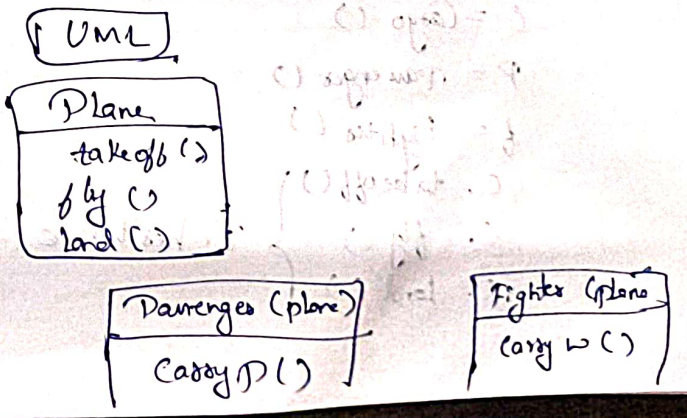
```
super().__init__()
```

Another method

```
class A:
    def __init__(self):
        self.a = 10

class B(A):
    def __init__(self):
        super().__init__()
        self.b = 20

class C(B):
    def __init__(self):
        super().__init__()
        self.c = 30

c1 = C()
Print(c1.c)
Print(c1.b)
P(c1.a)
```

Before Inheriting

```
class Cargoplane:
    def takeoff(self):
        Print("The plane is takeoff")
    def fly(self):
        Print("fly")
    def land(self):
        Print("Land")
    def carry(self):
        Print("Cargo")

class Passengerplane:
    def takeoff(self):
        Print("takeoff")
    def fly(self):
        Print("fly")
    def land(self):
        Print("Land")
    def carry(self):
        Print("Passenger")

class fighterplane:
    def takeoff(self):
        Print("Takeoff")
    def fly(self):
        Print("fly")
    def land(self):
        Print("Land")
    def carry(self):
        Print("weapons")

c = Cargoplane()
P = Passengerplane()
f = fighterplane()
c.takeoff()
c.fly()
c.land()
c.carry()
P.takeoff()
P.fly()
P.land()
P.carry()
f.takeoff()
f.fly()
f.land()
f.carry()
```



```
class Plane:
    def takeoff(self):
        Print("Plane is takeoff")
    def fly(self):
        Print("Plane is flying")
    def Land(self):
        Print("Plane is Landing")

class Cargo(plane):
    def carryC(self):
        Print("Plane is carry Cargo")

class Passenger(plane):
    def carryP(self):
        Print("Plane is carry passenger")

class Fighter(plane):
    def carryW(self):
        Print("Plane is carry weapon")

C = Cargo()
P = Passenger()
F = Fighter()
C.takeoff()
C.fly()
C.Land()
```

Inheritance

C.carryC() → Specialized method.

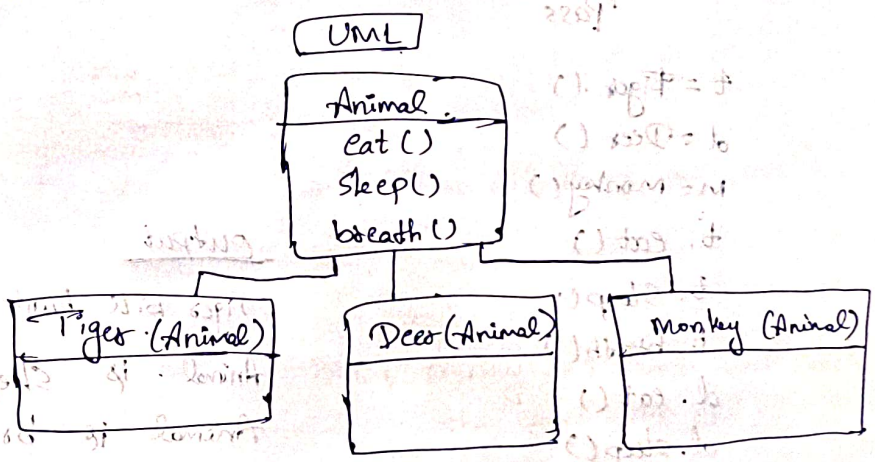
P.takeoff()
P.fly()
P.Land()
} Inheritance

P.carryP() → Specialized method.

F.takeoff()
F.fly()
F.Land()
} Inheritance

F.carryW() → Specialized method.

Animals:




```
class Animal:
    def eat(self):
        Print("Animal is eating")
    def sleep(self):
        Print("Animal is sleeping")
```

```
    def breath (self):
        Print ("Animal is Breathing")

class Tiger (Animal):
    def eat (self):
        Print ("Tiger will hunt and eat")

class Deer (Animal):
    Pass
```

lower case

```
class Monkey (Animal):
    Pass

t = Tiger ()
d = Deer ()
m = Monkey ()
t. eat ()
t. sleep ()
t. breath ()
d. eat ()
d. sleep ()
d. breath ()
m. eat ()
m. sleep ()
m. breath ()
```

Output

```
Tiger will hunt & eat
Animal is sleeping
Animal is breathing
Animal is eating
Animal is sleep
Animal is Breathing
Animal is sleeping
Animal is breathing
```

Overridden Methods

```
class Animal:
    def eat (self):
        Print ("Animal is eating")
    def sleep (self):
        Print ("Animal is sleeping")
    def breath (self):
        Print ("Animal is breathing")

class Tiger (Animal):
    def eat (self):
        Print ("Tiger will hunt and eat")

class Deer (Animal):
    def eat (self):
        Print ("Deer will graze and eat")

class Monkey (Animal):
    def eat (self):
        Print ("Monkey will Jump and Catch
                the food and eat")

t = Tiger ()
d = Deer ()
m = Monkey ()
t. eat ()
t. sleep ()
t. breath ()
```

```
d. eat ()
d. sleep ()
d. breath ()
m. eat ()
m. sleep ()
m. breath ()
```

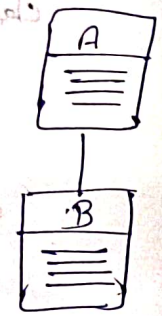
14/05

Types of Inheritance

1) Single level or Single inheritance.
2) Multi Level Inheritance
3) Hierarchical Inheritance
4) Multiple Inheritance

→ Single Inheritance

```
class A:
    def dispA(self):
        Print("Inside dispA")

class B(A):
    def dispB(self):
        Print("Inside dispB")

b = B()
b.dispB()
b.dispA()
```



o/p

```
Inside dispB
Inside dispA
```

→ Multi-Level Inheritance

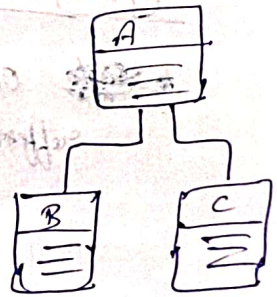
```
class A:
    def dispA(self):
        Print("Inside dispA")
class B(A):
    def dispB(self):
        Print("Inside dispB")
class C(B):
    def dispC(self):
        Print("Inside dispC")

c = C()
c.dispC()
c.dispB()
c.dispA()
```

o/p

```
Inside dispC
Inside dispB
Inside dispA
```

3) Hierarchical Inheritance : Two or more child class

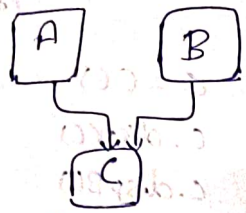
```
class A:
    def dispA(self):
        Print("Inside dispA")

class B(A):
    def dispB(self):
        Print("Inside dispB")

class C(A):
    def disp(self):
        Print("Inside dispC")

b = B()
c = C()
c.disp()
c.dispA()
b.dispB()
b.dispA()
```



o/p

```
Inside dispC
Inside dispA
Inside dispB
Inside dispA
```

4) Multiple Inheritance




```
class A:
    def dispA (self):
        Print (" Inside dispA")

class B:
    def dispB (self):
        Print (" Inside dispB")

class C(A,B):
    def dispC (self):
        Print ("Inside dispC")
```

```
c = C()
c.dispC ()
c.dispB ()
c.dispA ()
```

O/P

```
Inside dispC
Inside dispB
Inside dispA
```

Polymorphism

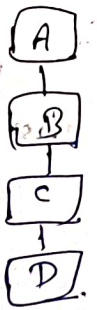
One object performing Multiple Task is refferd to as polymorphism.

```
class Plane:
    def takeoff (self):
        Print ("Plane is takeoff")
    def fly (self):
        Print (" Plane is fly")
    def land (self):
        Print ("Plane is Landing")

class Cargo (Plane):
    def takeoff (self):
        Print ("Cargo is takeoff")
    def fly (self):
        Print (" Cargo is flying")
    def land (self):
        Print ("Cargo is Landing")

class Passenger (Plane):
    def takeoff (self):
        Print (" Passenger is takeoff")
    def fly (self):
        Print (" Passenger is flying")
    def land (self):
        Print (" Passenger is Landing")

class Fighter (Plane):
    def takeoff (self):
        Print (" Fighter is takeoff")
```

```
    def fly(self):
        Print("Fighter is flying")
    def land(self):
        Print("Fighter is Landing")

c = Cargo()
p = Passenger()
f = Fighter()

def allowplanes(obj):
    obj.takeoff()
    obj.fly()
    obj.land()

allowplanes(c)
allowplanes(p)
allowplanes(f)
```

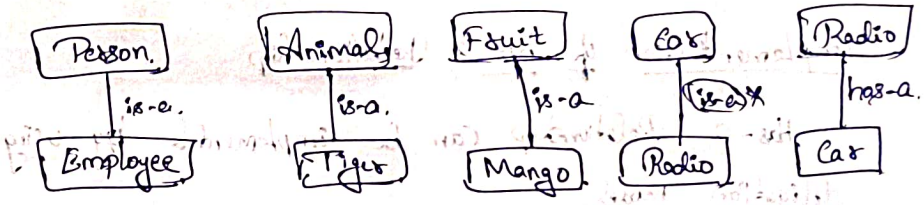
Invoking parentclass method during overriding

```
class A:
    def display(self):
        Print("Inside dispA")
class B(A):
    def display(self):
        Print("Inside dispB")
class C(B):
    def display(self):
        Print("Inside dispC")
class D(C):
    def dispD(self):
        A.display(self)
        B.display(self)
        C.display(self)

d = D()
d.dispD()
```



O/P:
Inside dispD
Inside dispA
Inside dispB
Inside dispC

Has-a Relationship



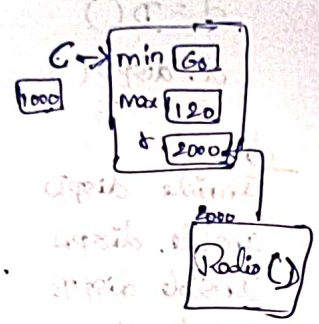
"delegation"

The process of creating object of one class in another class is referred to as delegation.

Eg: for delegation

```
class Radio:
    def turnOn(self, x):
        if (x==1):
            Print("Radio is on")
        else:
            Print("Radio is off")
```

```
class Car:
    def __init__(self, min, max):
        self.min = min
        self.max = max
        self.r = Radio()
```




```
c = Car(60, 120)
Print(c.min)
Print(c.max)
c.r.turnOn(1)
c.r.turnOn(2)
```

Op

60

120

Radio is now ON

Radio is OFF

Implementation of has-a Relationship

Has-a Relationship can be implemented by using deligation model.

It Consist of two things.

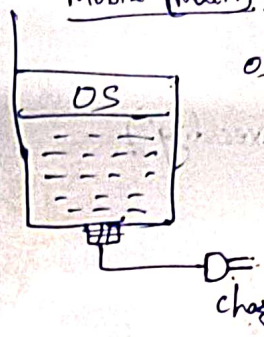
1) Composition [Composed object]
2) Agregation [Aggregate object]

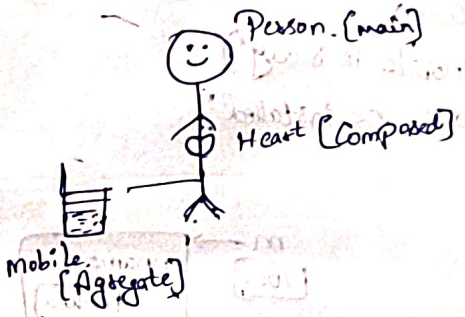
1) Composed object

All those objects which can't exist without the main object are reffered to as Composed object.

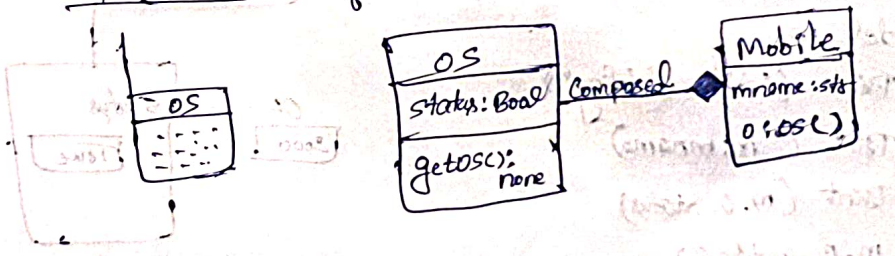
Aggregation

All those object which can exist without the main object are refered to as aggregated object.





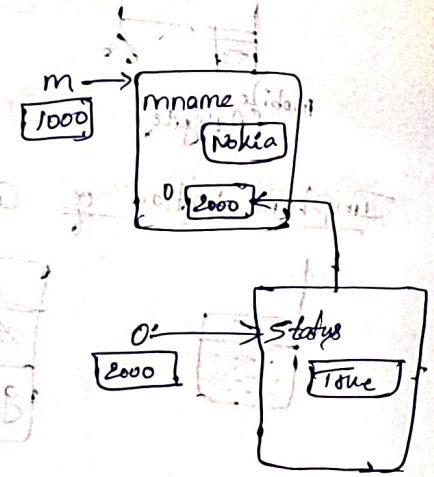
Implementation of Composed object



Composed object

Aggregate object.

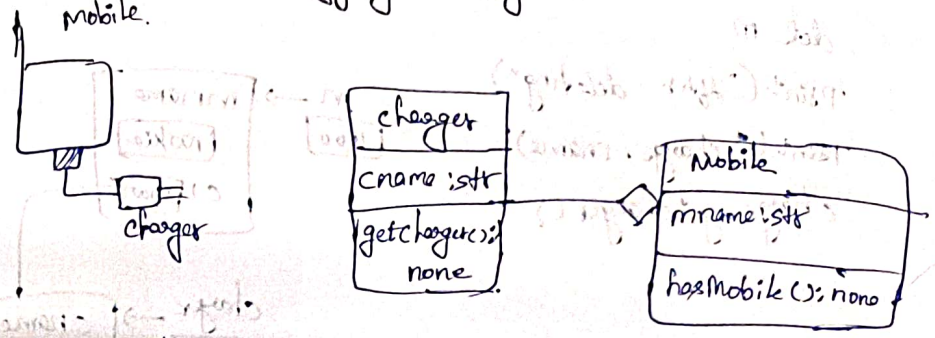
```
class OS:
    def __init__(self):
        self.status = "True"
        Print("OS is ready")

    def getOS(self):
        Print("OS is still Executing")

class Mobile:
    def __init__(self, name):
        self.mname = name
        self.o = OS()
        Print("Mobile is ready")
        Print("with OS installed")

m = Mobile("Nokia")
Print(m.mname)
Print(m.o.status)
m.o.getOS()

del m
Print("After deleting")
Print(m.mname)
Print(m.o.status)
m.o.getOs()
```



After deleting m (del m)
It shows an error -

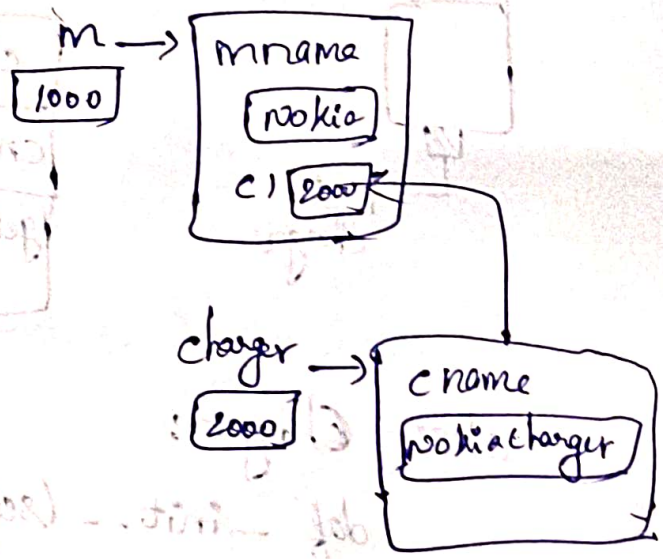
o/p
OS is ready
Mobile is ready
with OS installed
Nokia
True
OS is still running

Implementation of Aggiegate object:




```
class Charger:
    def __init__(self, name):
        self.cname = name
        Print("Charger is ready")
    def getCharger(self):
        Print("Charger is used for charging")

class Mobile:
    def __init__(self, name):
        self.mname = name
        self.c1 = ""
        Print("Mobile is Ready")
    def hasMobile(self, c):
        self.c1 = c
        Print("Both Mobile and charger")

m = Mobile("Nokia")        # m ← 1000
charger = Charger("Nokia charger")   # charger ← 2000
Print(m.mname)
m.hasMobile(charge)
Print(m.c1.cname)
m.c1.getCharger()
```

```
del m
Print("After deleting")
Print(m.charge.cname)
charge.getCharger()
```



o/p

Mobile is ready
Charger is ready
Nokia
Both Mobile and charger
Nokia charger
charger is used for charging.
After deleting
Nokia charger
charger is used for charging

Note:- Aggregate object will be passed as Parameter to the main object. method.

Programming

* Do not ~~cant~~ be scared to commit mistake, Learn revise and moderate.

Rules of Programming

1) keep your code dynamic
2) First write a code in Brute force approach [Get the output any how]
3) then optimize your code.

Operators :- operation

It is a task or an action perform on the values based on the operator used.

operand :- It is value used for operator.

operators :-

It is a special symbol (or) keyword that includes inbuild functionality. based on that the operation is perform on operands,

Ex :- 5+3

operation : Addition

operator : +

operand : 5, 3.

operators :- +, −, ||, /, *, and, or, not. ...

These are Totaly 7 set number of operator in Python.

1) Arthemetic operator.
2) Comparation or Relational operator
3) logical operator
4) Bit wise.

5) Assignment

6) Identity

7) Membership.