Day-1 31/10/2023

**Object**: - It will have some attributes (id, type, value, refereeing count etc.)

For **INT** datatype

A = 1000----->Memory allocation in RAM by PVM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| stack | heap | | | |
|  | id | type | value | rc |
| A | 101 | int | 1000 | 1 |

For **String** datatype

A = “HYDH” ----->Memory allocation in RAM by PVM

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| stack | heap | | | | |
|  | index allocation | id | type | value | rc |
| A | 0 1 2 3 | 101 | str | <> | 1 |
|  | 0 | 102 | str | H | 2 |
|  | 1 | 103 | str | Y | 1 |
|  | 2 | 104 | str | D | 1 |

If any referring object is referring to another referring object like A=B the PVM will not allocate a new object, it will refer to what the object is referring to.

For **List** datatype

A = [1000, 2000, 3000] ----->Memory allocation in RAM by PVM

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| stack | heap | | | | |
|  | index\_allocation | id | type | value | rc |
| A | 0 1 2 3 4 | 101 | list | <> | 1 |
|  | 0 | 102 | int | 1000 | 1 |
|  | 1 | 103 | str | 2000 | 1 |
|  | 2 | 104 | str | 3000 | 1 |

While creating a list memory index allocation it will add more extra indexes. For **tuple**, it will not create extra indexes that wise it is immutable.

B = A [:] ----->**Hybrid**

For the B will create new index allocation but the values referring what indexes of A objects referring

**Python-Task-1**

Question: - Write a flow of memory allocation in RAM for following Python code

1. X = 1000
2. Y = [1000, 2000, 3000]
3. Z = “HYD”
4. A = [X, Y, 1000, 2000, 3000, Z]
5. K = A
6. L = [1000, 2000, 3000]
7. M = A [:]
8. N = deepcopy(a)
9. O = copy(a)
10. A.append(4000)
11. print(K)
12. print(L)
13. print(M)
14. print(N)
15. print(O)

Soultion: -

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **stack** | **heap** | | | | | **last\_id** |
| ref\_obj | objs\_indexs\_allcoation | id | value | type | rc |  |
| X |  | 1001 | 1000 | int | 5 |  |
| Y | 0 1 2 | 1002 | <> | list | 5 |  |
|  | 0 | 1003 | 1000 | int | 1 |  |
|  | 1 | 1004 | 2000 | int | 1 |  |
|  | 2 | 1005 | 3000 | int | 1 |  |
| Z | 0 1 2 | 1006 | <> | str | 5 |  |
|  | 0 | 1007 | H | str | 1 |  |
|  | 1 | 1008 | Y | str | 1 |  |
|  |  | 1009 | D | str | 1 |  |
| A | 0 1 2 3 4 5 6(4000) | 1010 | <> | list | 2 |  |
|  | 0 | 1001 | id referring | | |  |
|  | 1 | 1002 | id referring | | |  |
|  | 2 | 1011 | 1000 | int | 4 |  |
|  | 3 | 1012 | 2000 | int | 4 |  |
|  | 4 | 1013 | 3000 | int | 5 |  |
|  | 5 | 1006 | id referring | | |  |
|  | 6 | 1021 | 4000 | int | 1 | Stopped Here |
| K | 0 1 2 3 4 5 | 1010 | id referring | | |  |
| L | 0 1 2 | 1014 | <> | list | 1 |  |
|  | 0 | 1015 | 1000 | int | 1 |  |
|  | 1 | 1016 | 2000 | int | 1 |  |
|  | 2 | 1017 | 3000 | int | 1 |  |
| M | 0 1 2 3 4 5 | 1018 | <> | list | 1 |  |
|  | 0 | 1001 | id referring | | |  |
|  | 1 | 1002 | id referring | | |  |
|  | 2 | 1011 | id referring | | |  |
|  | 3 | 1012 | id referring | | |  |
|  | 4 | 1013 | id referring | | |  |
|  | 5 | 1006 | id referring | | |  |
| N | 0 1 2 3 4 5 | 1019 | <> | list | 1 |  |
|  | 0 | 1001 | id referring | | |  |
|  | 1 | 1002 | id referring | | |  |
|  | 2 | 1011 | id referring | | |  |
|  | 3 | 1012 | id referring | | |  |
|  | 4 | 1013 | id referring | | |  |
|  | 5 | 1006 | id referring | | |  |
| O | 0 1 2 3 4 5 | 1020 | <> | list | 1 |  |
|  | 0 | 1001 | id referring | | |  |
|  | 1 | 1002 | id referring | | |  |
|  | 2 | 1011 | id referring | | |  |
|  | 3 | 1012 | id referring | | |  |
|  | 4 | 1013 | id referring | | |  |
|  | 5 | 1006 | id referring | | |  |

Output: -

[1000, [1000, 2000, 3000], 1000, 2000, 3000, 'HYD', 4000]

[1000, 2000, 3000]

[1000, [1000, 2000, 3000], 1000, 2000, 3000, 'HYD']

[1000, [1000, 2000, 3000], 1000, 2000, 3000, 'HYD']

[1000, [1000, 2000, 3000], 1000, 2000, 3000, 'HYD']

Day-2 01/11/2023

**Monkey Patching: -**

[Monkey patching is a technique used in programming to dynamically update the behavior of a piece of code at run-time](https://en.wikipedia.org/wiki/Monkey_patch). [It allows developers to change or extend the behavior of existing code without directly modifying the source code](https://dev.to/himankbhalla/what-is-monkey-patching-4pf). [This can be particularly useful when the code is closed-source or the developer doesn’t have access to the original code.](https://dev.to/himankbhalla/what-is-monkey-patching-4pf)

1. class A:
2. def func(self):
3. print("func() is being called")
5. # We use the above class in below code and change behavior of func() at run-time by assigning different value.
6. def monkey\_f(self):
7. print("monkey\_f() is being called")
9. # replacing address of "func" with "monkey\_f"
10. A.func = monkey\_f
12. obj = A ()
13. # Now calling func() will output: "monkey\_f() is being called"
14. obj.func()

For **functions** type

Write a flow of memory allocation in RAM for following Python code

1. def fun (x,y):
2. print (x+y)
3. Res = fun

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| stack | heap | | | | |
|  | index\_allocation | id | type | value | rc |
| fun |  | 101 | <> | function | 2 |
| Res | referring | 101 |  |  |  |

1. def fun (x, y):
2. return x+y
3. Res = fun (1000, 2000)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| stack | | heap | | | |
| main | local | id | type | value | rc |
| fun |  | 101 | <> | function | 1 |
|  | x | 102 | int | 1000 | 1 |
|  | y | 103 | int | 2000 | 1 |
| Res |  | 104 | int | 3000 | 1 |

After execution of function the local stack or frame memory will be deleted

1. L = [1000, 2000, 3000]
2. def fun (x, y):
3. x.append(6000)
4. Res = fun (L, 2000)
5. Res1 = fun (L, 3000)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| stack | | | heap | | | |
| main | local | indexes | id | type | value | rc |
| L |  | 0 1 2 3 4 5 6 | 101 | list | <> | 2 |
|  |  | 0 | 102 | int | 1000 | 1 |
|  |  | 1 | 103 | int | 2000 | 1 |
|  |  | 2 | 104 | int | 3000 | 1 |
|  |  | 3 | 108 | int | 6000 | 1 |
|  |  | 4 | 109 | int | 6000 | 1 |
| fun |  |  | 105 | function | <> | 1 |
|  | x | referring | 101 |  |  |  |
|  | y |  | 107 | int | 2000 | 1 |
| Res |  | None |  |  |  |  |
| Res1 |  | None |  |  |  |  |

After execution of function the local stack or frame memory will be deleted

For finding the **referring count(rc)** in Python code

1. import sys
2. A = [1,2,34]
3. Rc = sys.getrefcount(A)
4. print(rc)---->2
5. #But it will give not us expected

**Python-Task-2**

Question: - Write a flow of memory allocation in RAM for following Python code

1. X = [2000, [6000, 7000]]
2. A = [[[3000, 5000, 7000], 4000, X], 1000, 2000]
3. B = A [:]
4. E = A
5. C = deepcopy(A)
6. A [1] = 6000
7. A [0] [0] = 7000
8. X.append(5000)
9. X [1]. append (8000)
10. print(X)
11. print(A)
12. print(B)
13. print(E)
14. print(C)

Solution: -

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| stack | heap | | | | |
|  | index\_allocation | id | type | value | rc |
| X | 0 1 2 3 | 101 | list | <> | 2 |
|  | 0 | 102 | int | 2000 | 1 |
|  | 1 (0 1 2 3) | 103 | list | <> | 1 |
|  | 0 | 104 | int | 6000 | 1 |
|  | 1 | 105 | int | 7000 | 1 |
|  | 2 |  |  |  |  |
|  | 3 |  |  |  |  |
|  | 2() | 120 | int | 5000 | 1 |
|  | 3() |  |  |  |  |
| A | 0 1 2 3 4 | 106 | list | <> | 1 |
|  | 0(0 1 2 3 4) | 107 | list | <> | 1 |
|  | 0(0 1 2 3 4) | 108 | list | <> | 1 |
|  | 0 | 109 | int | 3000 | 1 |
|  | 1 | 110 | int | 5000 | 1 |
|  | 2 | 111 | int | 7000 | 1 |
|  | 3 |  |  |  |  |
|  | 4 |  |  |  |  |
|  | 1 | 112 | int | 4000 | 1 |
|  | referring(2) | 101 |  |  |  |
|  | 3 |  |  |  |  |
|  | 4 |  |  |  |  |
|  | 1 | 113 | int | 1000 | 1 |
|  | 2 | 114 | int | 2000 | 1 |
|  | 3 |  |  |  |  |
|  | 4 |  |  |  |  |
| B=A[:] | 0 1 2 | 115 | list | <> | 1 |
|  | 0(referring)->A[0] | 107 |  |  |  |
|  | 1(referring)->A[1] | 113 |  |  |  |
|  | 2(referring)->A[2] | 114 |  |  |  |
| E=A | referring-->A | 106 |  |  |  |
| C | 0 1 2 | 116 | list | <> | 1 |
|  | 0(referring)->A[0] | 107 |  |  |  |
|  | 1(referring)->A[1] | 113 |  |  |  |
|  | 2(referring)->A[2] | 114 |  |  |  |
| A[1]=6000 | updated c/t value in A | 118 | int | 6000 | 1 |
| A[0][0]=7000 | updated c/t value in A | 119 | int | 7000 | 1 |
| X.append(5000) | updated c/t value in X->2 | 120 | int | 5000 | 1 |
| X[1].append(8000) | updated c/t value in X->1 | 121 | int | 8000 | 1 |

Output: -

[2000, [6000, 7000, 8000], 5000]

[[7000, 4000, [2000, [6000, 7000, 8000], 5000]], 6000, 2000]

[[7000, 4000, [2000, [6000, 7000, 8000], 5000]], 1000, 2000]

[[7000, 4000, [2000, [6000, 7000, 8000], 5000]], 6000, 2000]

[[[3000, 5000, 7000], 4000, [2000, [6000, 7000]]], 1000, 2000]

Day-3 02/11/2023

**Shallow copy or copy or A [:]: -**

If we add or update immutable objects to any sequence like list, set, dictionary, etc., to an original object it will affect only the original object not a copied item.

If we add or update mutable objects to any sequence like list, set, dictionary, etc., to an original object it will affect both original and copied item.

**Deep copy: -**

It will not change. It remains the same as the copied item if we update or add mutable or immutable objects to an original object.

**Class: -**

class is a keyword in python that is used to define user-defined data types. There are two types of objects for class

* Class object
* Data object

For **class** object: -

1. A = 1000
2. Isinstance(a, int) --->True

Write a flow of memory allocation in RAM for following Python code

1. def fun(\*args):
2. print(‘fun’)
3. def fun(\*args):
4. print(‘fun1’)
5. return fun1
6. Res = fun ()
7. print (Res)
8. R1 = Res (10,20,30)
9. print (‘r1: ‘ r1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| stack | | heap | | | | op |
| main | local | id | type | value | rc |  |
| fun |  | 101 | func | <> | 1 |  |
| res | fun1 | 102 | func | <> | 1 | fun |
|  | fun | 101 |  |  |  |  |
|  | 0 | 103 | int | 10 | 1 |  |
|  | 1 | 104 | int | 20 | 1 |  |
|  | 2 | 105 | int | 30 | 1 |  |
| r1 |  | 106 | func | None | 1 | fun, None |

After execution of function the local stack or frame memory will be deleted

For **class**type objects

Write a flow of memory allocation in RAM for following Python code

1. class Emp:
2. def \_\_init\_\_ (x, name, id):
3. x.name = name
4. x.id = id
5. def get (x):
6. return x.name, x.id
7. A = Emp ()
8. B = Emp (‘john’, 111)
9. Print (B.get())
10. print (type (Emp)) --> Type
11. Print(type(A)) --> class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| stack | | | heap | | | |
| main | namespace | local | id | type | value | rc |
| Emp |  |  | 101 | type | <> | 2 |
|  | x |  | 102 | <> | <> | 1 |
|  | \_\_init\_\_ |  | 103 | function | <> | 1 |
|  | get |  | 104 | function | <> | 1 |
| A |  |  | 101 |  |  |  |
|  |  | x | 105 | <> | <> | 1 |
|  |  | name | 106 | str | john | 1 |
|  |  | id | 107 | int | 111 | 1 |
| B | x | referring | 105 |  |  |  |
|  | name | referring | 106 |  |  |  |
|  | id | referring | 107 |  |  |  |

After execution the local stack will be deleted, and namespace remains same

Day-4 03/11/2023

Different types of arguments passing to functions

1. **def** fun (x,y,z**=**0, \*args, **\*\***kwargs):   
   print(f"x={x},y={y},z={z},args={args},kwargs={kwargs}")   
   fun(10,20,30,40,50,60, a**=**3000) ----->No Error #All values passed

1. **def** fun (x,y,z**=**0,**\***args, **\*\***kwargs):   
   print(f"x={x},y={y},z={z},args={args},kwargs={kwargs}")   
   fun(10,20,30,40,50,60,x**=**3000) -----> Error # Already x is declared again given

1. **def** fun (x,y,z**=**0,**\***args, **\*\***kwargs):   
   print(f"x={x},y={y},z={z},args={args},kwargs={kwargs}")   
   fun(y**=**1000, z**=**3000, a**=**1000) ------> Error # Not passed the x value

fun(10,"ok","approved",12.34)

  **output of above function call**

x=10,y=ok,z=approved,args=(12.34,),kwargs={}

1. **def** fun(x, **/**): # **If we use / as last parameter it allows only positional argumements**   
   print(x)   
   fun(10)   
   fun(x**=**10) ----> Error

1. **def** fun (**\***, x): # **If we use \* as first parameter it allows only keyword argumements**   
      
   print(x)   
   fun(x**=**10)   
   fun(10) ----> Error

1. **def** get\_values(x):*#x=10*   
   print("started")   
   **for** i **in** range(x):*#i=0,i=1*   
   print("i=",i)   
   **yield** i   
   res **=** get\_values(10)   
   print("result=",res) -----> It will generate **generator object** when the **yield** is used in function
2. r1**=**next(res) -----> It will start executing code for **first** **iteration** only and when called for next time it calls the where the code stopped the previous function until loop completes   
   print("r1=",r1)   
   print("ok")

1. **def** fun(x):   
   c**=**1   
   **while** c**<**x:   
   **yield** c**\***1000   
   c**=**c**+**1   
   res **=** fun(4)   
   print(res)   
   r1 **=** []   
   for i **in** res:   
   r1**.**append(i**+**2)   
   print(r1)
2. **Output: -** [1002, 2002, 3002]

1. **def** fun(x):   
   **yield** 30   
   **return** 10   
   res**=**fun (10)   
   print("res=", res) -----> It will generate generate generator object
2. r1**=**next(res)   
   print("r1=",r1) -----> op:- r1=30
3. r1**=**next(res)   
   print("r1=",r1) -----> op:- Error-stopIteration

1. **def** fun(x):   
   **return** 10   
   **yield** 30   
   res**=**fun(10)   
   print("res=",res) -----> It will generate generate generator object   
   r1**=**next(res) ------   
   print("r1=",r1) -----> op:- Error-stopIteration

1. **def** fun():   
   print(1)   
   **yield** 10   
   print(2)   
   **yield** 20   
   print(3)   
   **yield** 30   
   res**=**fun()   
   print("res=",res)   
   for k **in** res:   
   print("k=",k)   
   print("DONE")   
   print("iterating through generator object second time")   
   for k **in** res:*#next(res)*   
   print("k=",k)   
   print("Second done DONE")   
   print("calling again")   
   res**=**fun()   
   print("iterating through generator object ")   
   for k **in** res:*#next(res)*   
   print("k=",k)   
   print("DONE")

1. l**=**["user1","user3","user2","user4"]   
   k**=**l**.**sort()   
   print(k) ----> None

Print(l) -----> the above l

**Decorators: -**

[In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function](https://www.programiz.com/python-programming/decorator). [The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.](https://www.programiz.com/python-programming/decorator)

[Decorators are a powerful tool in Python as they allow programmers to modify the behavior of a function or class without permanently modifying it](https://www.geeksforgeeks.org/decorators-in-python/). They are used extensively in Python frameworks like Flask and Django.

[Remember, in Python, everything is an object, even functions](https://www.programiz.com/python-programming/decorator). [This means you can pass functions as arguments to other functions, return them from other functions, and even store them in data structures](https://www.geeksforgeeks.org/decorators-in-python/). This is a key concept that makes decorators possible in Python.

1. **def** order\_data(f):*#f=get\_user*   
   **def** wrapper():   
   res**=**f()   
   print(res)   
   res**.**sort()   
   **return** res   
   **return** wrapper   
      
   def get\_users():   
   **return** ["user1","user3","user2","user4"]   
   def get\_products():   
   **return** ["p1","p3","p2","p4"]   
   def get\_orders():   
   **return** ["o1","o3","o2","o4"]   
   res**=**order\_data(get\_users)   
   print("res=",res)   
   r1**=**res()   
   print("r1=",r1)

Comparison of **built in data structures**: -

For **Lists**: -

1. a = [1,2,3,3]
2. b = [3,4,5,4]
3. print(a>b) ---------> False # It compares the starting value

For **tuples**: -

1. x = (10, 20, 30)
2. y = (20, 4, 10)
3. print (x < y) -------> True # It compares the starting value

For **sets**: -

1. s1 = {1,2,4,7}
2. s2 = {1,2,4}
3. print (s1 > s2) ------> True # It compares based on superset and subset

For **dictionary**: -

1. dict1 = {'a':1, 'b':2}
2. dict2 = {'c':'3', 'd':4}
3. print (dict1 > dict2) ------> Type Error

The error because the dictionary items can’t be compared other dictionary items

1. dict1 = {'a':1, 'b':2}
2. dict2 = {'c':'3', 'd':4}
3. print (dict1 == dict2) ------> False #It compares based on key and value pair
4. If we used to **equality for list, tuple, string** it will check **all the items** in the sequence and return Boolean value

Arthamatic operators used for different **data types**: -

1. Arthamatic operators are +, -, \*, /, %, //, \*\* etc.
2. Data types are int, float, string, boolean

For **Integers**: -

1. a = 10
2. b = 20
3. print(a+b) -->30

1. a = 10
2. b = 20.34
3. print(a+b) -->30.34

1. a = 10
2. b = ‘str’
3. print(a+b) -->TypeError

1. a = 10
2. b = True
3. print(a+b) --> 11 #Checked the one operator worked other datatype and remain operators also works

For **Strings**: -

1. s1 = 'abc'
2. s2 = 'd'
3. print(s1+s2) --> abcd

1. s1 = 'abc'
2. s2 = 'd'
3. print(s1-s2) --> #If we used \*, //, / Type Error occurs

1. s1 = 'abc'
2. s2 = 2
3. print(s1\*s2) --> abcabc

Day-5 04/11/2023

If we used string methods to a string it treats every character behind and beside have one empty space

**Python built-in functions: -**

1. add **=** **lambda** x:x**\*\***2   
   print(type(add)) -----><class 'function'>
2. print(add(10)) ---->100

1. **def** check\_even(x):   
       **if** x**%2**==0:   
           **return** **True**   
       **else**:   
           **return** **False**   
   **print(check\_even(12)) --->True**

1. **def** check\_even(x):   
       **return** **True** **if** x**%2**==0 else False  #Short-hand nonation   
   print(check\_even(12)) ----> True

1. check\_even **=** **lambda** x:**True** **if** x**%2**==0 else False   
   print(check\_even(12)) #Using lambda simply written if we want single value as output

1. l**=**"abcdabdcababab"   
   # write a code to find a letter which is occured most number of times   
   res**=**{}   
   for i **in** l:   
       **if** i **not** **in** res:   
           res[i]**=**1   
       **else**:   
           res[i]**+=**1   
   print(res) ----> {'a': 5, 'b': 5, 'c': 2, 'd': 2}

1. l**=**"abcdabdcababab"   
   # write a code to find a letter which is occured most number of times   
   res**=**{}   
   for i **in** l:   
       **if** i **not** **in** res:   
           res[i]**=**1   
       **else**:   
           res[i]**+=**1   
   m**=**0   
   cahr**=**""   
   for i,j **in** res**.**items():   
       **if** j**>=**m:   
           m**=**j   
           char**=**i   
   print(m, char) ------> 5 b

1. l**=**[1,2,3,4]   
   print(max(l)) -------> 4 #prints based on maximum value

1. l**=**{'a': 5, 'b': 5, 'c': 2, 'd': 2}   
   print(max(l)) #prints the based on the key of Unicode value which is max

1. max("abcd") #prints the based on unicode of sequence of each item which is max

1. max([(1,2),(3,4),(5,6)]) #prints the based on first value which is maximum

1. **def** fun(value):   
       print(value)   
       **return** value   
   max([(1,2),(30,4),(5,6)], key**=**fun) #For maximum function can pass key as argument function and decides the return value --->(1, 2) (30, 4)(5, 6)(30, 4)

         Here decided based on **first value of tuple** for maximum of list

1. **def** fun(value):   
       print(value)   
       **return** value[1]   
   max([(1,2),(30,4),(5,6)], key**=**fun) --->(1, 2)(30, 4)(5, 6)(5, 6)

          # Here decided based on **second value of tuple** maximum of list

1. data**=**[{"name":"jay","age":8},{"name":"vedha","age":7}]   
   print(max(data)) --->Type Error

1. data**=**[{"name":"jay","age":8},{"name":"vedha","age":7}]   
   def fun(d):   
       **return** d**.**get("age")    
   print(max(data, key**=**fun)) ----> {'name': 'jay', 'age': 8}

           #Here passed the key function and decided maximum value of dic of list of  keys must be same for each item

1. data**=**[{"name":"jay","age":8},{"name":"vedha","age":7}]   
   print(max(data, key**=lambda** d:d**.**get("name"))) #The above one simplified using lambda

1. s**=**"abcdabcdaabcaabc"   
   d**=**{i:s**.**count(i) **for** i **in** s}   
   print(d) -----> {'a': 6, 'b': 4, 'c': 4, 'd': 2}

1. s**=**"abcdabcdaabcaabc"   
   d**=**{i:s**.**count(i) **for** i **in** set(s)}   
   max(d, key**=lambda** x:d**.**get(x)) ----> 'a'

1. print(set({"name":"jay","age":23})) ----> {'age', 'name'}

#Set don’t allow duplicates set takes dictionary keys as set elements

1. print(set("abcdabcdabcd",)) ----> {'c', 'a', 'd', 'b'}

1. print(set(("abcdabcdabcd","abcdabcdabcd","abcdabcd"))) ----> {'abcdabcdabcd', 'abcdabcd'}

Day-6   06/11/2023

For **strings** if we perform string methods operations it will **create new index** **object** whereas **lists** **don’t create** new index object

1. l**=**[10,20,30,30,40,30,20,30]   
   for i **in** l:   
   **if** i**==**30:   
   l**.**remove(i)   
   print(l) ---------> [10, 20, 40, 20, 30]

#Whenever the two same elements are present next to next if we want to remove the element while iterating the index value changes for each iteration  when the condition is satisfied

1. l**=**[10,20,30,30,40,30,20,30]   
   k**=**l[::**-**1] #Here can take copy or deepcopy the output same   
   for i **in** k: # Here new index object created   
   **if** i**==**30:   
   l**.**remove(i) #Here removing item from original object   
   don’t effecting the k object    
   print(l) -----> [10, 20, 40, 20]

**Comprehensions: -**

1. l**=**["jay","vedha","chaithu","jay","ram","anvesh","jay","vedha"]   
   res **=** {"jay":3,"vedha":2,"chaithu":1,"ram":1,"anvesh":1}   
   d**=**{}   
   for i **in** l:   
   **if** i **not** **in** d:   
   d[i]**=**1   
   **else**:   
   d[i]**+=**1   
   print(d) ------> {'jay': 3, 'vedha': 2, 'chaithu': 1, 'ram': 1, 'anvesh': 1}
2. **from** collections **import** defaultdict ##Used to dictionary to assign items   
   l**=**["jay","vedha","chaithu","jay","ram","anvesh","jay","vedha"]   
   res **=** {"jay":3,"vedha":2,"chaithu":1,"ram":1,"anvesh":1}   
   d**=**defaultdict(int)   
   for i **in** l:   
   d[i]**+=**1   
   print(d) ---> defaultdict(<class 'int'>, {'jay': 3, 'vedha': 2, 'chaithu': 1, 'ram': 1, 'anvesh': 1})

1. d**=**{"rahul":60,"rohit":60,"kohli":60,"jadeja":40,"gil":80,"hardik":30}   
   res**=**{}   
   for key,value **in** d**.**items():   
   **if** value **not** **in** res:   
   res[value]**=**[key]*#{60:["rahul"]}*   
   **else**:   
   res[value]**.**append(key)   
      
   print(res) --> {60: ['rahul', 'rohit', 'kohli'], 40: ['jadeja'], 80: ['gil'], 30: ['hardik']

1. d**=**{"rahul":60,"rohit":60,"kohli":60,"jadeja":40,"gil":80,"hardik":30}   
   res**=**defaultdict(list)   
   for key,value **in** d**.**items():   
   **if** value **not** **in** res:   
   res[value]**=**[key]*#{60:["rahul"]}*   
   **else**:   
   res[value]**.**append(key)   
      
   print(res) -----> defaultdict(<class 'list'>, {60: ['rahul', 'rohit', 'kohli'], 40: ['jadeja'], 80: ['gil'], 30: ['hardik']})

1. **from** collections **import** Counter   
   l**=**["jay","vedha","chaithu","jay","ram","anvesh","jay","vedha"]   
   print(Counter(l)) ------> Counter({'a': 3, 'b': 3, 'c': 3, 'd': 2})
2. Counter used to count the number of items of each element in a sequence