Range

- 'range' is one of the pre-defined class and treated as Sequence Data Type
- The purpose of range data type is that "To Store Range Or Sequnece of Numerical Integer
- Values with equal Interval of Value(Step) eithet in Forward OR backward Direction".
- On the object on range, we can perform Indexing and Slicing Operations
- An object of range belongs to Immutable .
- An Object of range maintains Insertion Order.
- To Store Range of Values, we have 3 Pre-defined Functions and we can use them in 3

- Syntax-1: varname=range(Value)

- This Syntax generates range of Values from 0 to Value-1
- here varname is an object of <class, range>

- Syntax-2: varname=range(Begin,End)

- This Syntax generates range of Values from Begin to End-1.
- here varname is an object of &lt <class, range>

- Syntax-3: varname=range(Begin,End,Step)

- This Syntax generates range of Values from Begin to End-1 with Equal Interval of Specified step values

- here varname is an object of <class, range>

In [ ]:
```python
r = range(6)
```

In [3]:
```python
print(r,type(r))
```

```
range(0, 6) <class 'range'>
```

In [6]:
```python
for v in r:
    print(r)
```

```
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
```

In [7]:
```python
for val in r:
    print(r)
```

```
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
range(0, 6)
```

In [9]:
```
for val in range(10):
    print(val)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [10]:
```
r = range(10,16)
```

In [11]:
```
print(r,type(r))
```

```
range(10, 16) <class 'range'>
```

In [12]:
```
for val in range(10,16):
    print(val)
```

```
10
11
12
13
14
15
```

In [14]:
```
for val in range(10,16):
    print(val,end=" ")
```

```
10 11 12 13 14 15
```

In [18]:
```
r = range(10,11,2)
```

In [19]:
```
print(r, type(r))
```

```
range(10, 11, 2) <class 'range'>
```

In [20]:
```
for val in range(10,11,2):
    print(val)
```

```
10
```

In [21]:
```
r = range(10,21,2)
```

In [22]:
```
print(r,type(r))
```

```
range(10, 21, 2) <class 'range'>
```

In [23]:
```
for val in range(10,21,2):
    print(val)
```

```
10
12
14
16
18
20
```

In [24]:
```
for val in range(10,21):
    print(val)
```

```
10
11
12
13
14
15
16
17
18
19
20
```

In [25]:
```
for val in range(1000,1006):
    print(val)
```

```
1000
1001
1002
1003
1004
1005
```

In [26]:
```
for val in range(10,21,2):
    print(val)
```

```
10
12
14
16
18
20
```

In [27]:
```
for val in range(100,201,10):
    print(val)
```

```
100
110
120
130
140
150
160
170
180
190
200
```

In [29]:
```
for val in range(10,0,-1):
    print(val)
```

```
10
9
8
7
6
5
4
3
2
1
```

In [31]:
```
for val in range(100,-1,-10):
    print(val)
```

```
100
90
80
70
60
50
40
30
20
10
0
```

In [33]:
```
for val in range(-10,-16,-1):
    print(val)
```

```
-10
-11
-12
-13
-14
-15
```

In [36]:
```
for val in range(-100,-9,10):
    print(val)
```

```
-100
-90
-80
-70
-60
-50
-40
-30
-20
-10
```

In [39]:
```
for val in range(-5,6):
    print(val)
```

```
-5
-4
-3
-2
-1
0
1
2
3
4
5
```

In [40]:
```
for val in range(-1000,-1201,-50):
    print(val)
```

```
-1000
-1050
-1100
-1150
-1200
```

In [41]:
```
r = range(-1000,-1201,-50)
```

In [42]:
```
print(r,type(r))
```

```
range(-1000, -1201, -50) <class 'range'>
```

In [43]:
```
r[0]
```

Out[43]:
```
-1000
```

In [44]:
```
r[-1]
```

Out[44]:
```
-1200
```

In [45]:
```
for val in r[0:2]:
    print(val)
```

```
-1000
-1050
```

In [46]:
```
for val in range(10,21,2)[::-1]:
    print(val)
```

```
20
18
16
14
12
10
```

In [48]:
```
for val in range(-100,-59,10):
    print(val)
```

```
-100
-90
-80
-70
-60
```

List Data Type

- 'list' is one of the pre-defined class and treated as list type data type
- The purpose of List type is that to store multiple values either of same type or different type or both types with unique and duplicates values.
- The elements of list must be written within square brackets [ ] and elements must seprated by comma .
- An object of list maintains insertion order.
- On the object of list , we can perform indexing and slicing operations.
- An object of list belongs to mutable & We have two types of list objects.

- To convert ont type elements into list type elements, we use list()

```
    a) empty list
    b) non-empty list
```

a) empty list:

An empty list is one, which does not contain any elements and wose length =0

```
syntax:        listobj = []

                   (or)


               listobj =list ()
```

In [2]:
```
l1 = [10,20,30,40,50]
print(l1,type(l1))
```
```
[10, 20, 30, 40, 50] <class 'list'>
```

In [3]:
```
l2=[10,"KVR","OUCET",94.25,"HYD"]
print(l2,type(l2))
```
```
[10, 'KVR', 'OUCET', 94.25, 'HYD'] <class 'list'>
```

In [4]:
```
print(l2[0])
```
```
10
```

In [5]:
```
print(l2[2])
```
```
OUCET
```

In [6]:
```
print(l2[4])
```
```
HYD
```

In [7]:
```
print(l2[0:4])
```

```
[10, 'KVR', 'OUCET', 94.25]
```

In [8]:
```
 print(id(l2))
```
```
1981950497600
```

In [9]:
```
l2[0]
```
Out[9]:
```
10
```

In [10]:
```
print(l2)
```
```
[10, 'KVR', 'OUCET', 94.25, 'HYD']
```

In [11]:
```
print(id(l2))
```
```
1981950497600
```

In [12]:
```
l3=[]
```

In [13]:
```
print(l3, type(l3))
```
```
[] <class 'list'>
```

In [15]:
```
 l4 = list ()
```

In [16]:
```
 print(l4, type(l4))
```
```
[] <class 'list'>
```

In [17]:
```
len(l3)
```
Out[17]:
```
0
```

In [18]:
```
len(l4)
```
Out[18]:
```
0
```

In [20]:
```
print(len(l3),len(l4))
```
```
0 0
```

Function in List

- To perform additional operations on list object along with slicing and indexing, we use the following prte-defined functions.

  append():

- This function is used for adding any element to list object at end (knows appending).

```
  Syntax:-     listobj.append(element)
```

In [25]:
```
l1=[]
print(l1, len(l1), type(l1))
```

```
[] 0 <class 'list'>
```

In [26]:
```
l1.append(10)
print(l1,type(l1),len(l1))
```

```
[10] <class 'list'> 1
```

In [27]:
```
l1.append("Rossum")
print(l1, len(l1))
```

```
[10, 'Rossum'] 2
```

In [28]:
```
l1.append(12.34)
print(l1, len(l1))
```

```
[10, 'Rossum', 12.34] 3
```

In [29]:
```
l2=[10,"Bharat"]
print(l2)
```

```
[10, 'Bharat']
```

In [30]:
```
l2.append(23.45)
print(l2)
```

```
[10, 'Bharat', 23.45]
```

In [31]:
```
l2.append("Hyd")
print(l2)
```

```
[10, 'Bharat', 23.45, 'Hyd']
```

2. Insert

- This function is used for inserting the specified element in list object by specifying valid existing index.

Syntax:- listobj.insert(index, element)

In [34]:
```
l2=[10,"Bharat",23.45,"HYD"]
print(l2)
```

```
[10, 'Bharat', 23.45, 'HYD']
```

In [35]:
```
l2.insert(2,"python")
print(l2)
```

```
[10, 'Bharat', 'python', 23.45, 'HYD']
```

In [36]:
```
l2.insert(2,"ampt")
print(l2)
```

```
[10, 'Bharat', 'ampt', 'python', 23.45, 'HYD']
```

In [38]:
```
l2[4]=43.45
print(l2)
```

```
[10, 'Bharat', 'ampt', 'python', 43.45, 'HYD']
```

3. Remove

- This function is used for removing the first occurentce of specified element.

```
if the specified element then we get  ValueError:
```

  list.remove(x): x not in list

  syntax: listobj.remove(element)

In [39]:
```
l3=[10,"Raju",23.45,10,"Raju","HYD"]
print(l3)
```
```
[10, 'Raju', 23.45, 10, 'Raju', 'HYD']
```

In [40]:
```
l3.remove(10)
print(l3)
```
```
['Raju', 23.45, 10, 'Raju', 'HYD']
```

In [42]:
```
l3.remove(10)
print(l3)
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [42], in <cell line: 1>()
----> 1 l3.remove(10)
      2 print(l3)

ValueError: list.remove(x): x not in list
```

4. pop(index)

- This function is used for removing the element of list based on valid index.
- if the index is invalid then we get IndexError
- Syntax:- listobj.pop(index)

In [46]:
```
l3 = [10, 23.45, 10, 'Raju' ,'HYD']
```

In [47]:
```
l3.pop(0)
```
Out[47]:
```
10
```

In [48]:
```
print(l3)
```
```
[23.45, 10, 'Raju', 'HYD']
```

In [49]:
```
l3.pop(11)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [49], in <cell line: 1>()
----> 1 l3.pop(11)

IndexError: pop index out of range
```

5. pop()

- This function is used for removing last element of list object.
- when we call pop() upon empty list object then we get IndexError
- Syntax:- listobj.pop()

In [51]:
```
l3=[10,"Raju",23.45,10,"Raju","HYD"]
```

In [52]:
```
l3.pop()
```

Out[52]:
```
'HYD'
```

In [53]:
```
print(l3)
```

```
[10, 'Raju', 23.45, 10, 'Raju']
```

In [54]:
```
l3.pop()
```

Out[54]:
```
'Raju'
```

In [55]:
```
print(l3)
```

```
[10, 'Raju', 23.45, 10]
```

In [56]:
```
 l3.pop()
```

Out[56]:
```
10
```

In [57]:
```
 print(l3)
```

```
[10, 'Raju', 23.45]
```

In [58]:
```
l3.pop()
```

Out[58]:
```
23.45
```

In [59]:
```
print(l3)
```

```
[10, 'Raju']
```

In [60]:
```
l3.pop()
```

Out[60]:
```
'Raju'
```

In [61]:
```
print(l3)
```

```
[10]
```

```
In [62]: l3.pop()
Out[62]: 10
```

```
In [63]: print(l3)
[]
```

```
In [64]: l3.pop()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [64], in <cell line: 1>()
----> 1 l3.pop()

IndexError: pop from empty list
```

6. count

- This function is used for finding number of occurences of a specified element.
- The specified eleemnt does not exists then whose count 0.
- Syntax: listobj.count(element)

```
In [66]: l1=[10,20,30,40,10,20,10,20]
```

```
In [67]: print(l1)
[10, 20, 30, 40, 10, 20, 10, 20]
```

```
In [68]: l1.count(10)
Out[68]: 3
```

```
In [69]: l1.count(20)
Out[69]: 3
```

```
In [70]: l1.count(40)
Out[70]: 1
```

```
In [71]: l1.count(50)
Out[71]: 0
```

```
In [72]: l1.count("KVR")
Out[72]: 0
```

7. REVERSE

- This function is used for obtaing reverse order oginial elements of list.
- Syntax:- listobj.reverse()

In [74]:
```
l1=[10,20,30,40,40-50,0,12]
```

In [75]:
```
print(l1)
```

```
[10, 20, 30, 40, -10, 0, 12]
```

In [76]:
```
l1.reverse()
```

In [77]:
```
 print(l1)
```

```
[12, 0, -10, 40, 30, 20, 10]
```

8.Sort

- This function is used for sorting similar type data of list object in ASCending order by default.
- Syntax: listobj.sort() (or) listobj.sort(reverse=False)
- Syntax: listobj.sort(reverse=True)------DESC order

In [78]:
```
l1=[10,20,30,40,40-50,0,12]
```

In [79]:
```
print(l1)
```

```
[10, 20, 30, 40, -10, 0, 12]
```

In [80]:
```
 print(l1)
```

```
[10, 20, 30, 40, -10, 0, 12]
```

In [81]:
```
l1.reverse()
```

In [82]:
```
 print(l1)
```

```
[12, 0, -10, 40, 30, 20, 10]
```

special cases:

In [83]:
```
l1=[10,20,30,40,40-50,0,12]
```

In [84]:
```
print(l1)
```

```
[10, 20, 30, 40, -10, 0, 12]
```

In [85]:
```
l1.sort(reverse=True)
```

In [86]:
```
 print(l1)
```

```
[40, 30, 20, 12, 10, 0, -10]
```

In [87]:
```
l1.reverse()
```

In [88]:
```
print(l1)
```
```
[-10, 0, 10, 12, 20, 30, 40]
```

In [89]:
```
print(l1)
```
```
[-10, 0, 10, 12, 20, 30, 40]
```

In [90]:
```
print(l1)
```
```
[-10, 0, 10, 12, 20, 30, 40]
```

9. Copy

- This function is used for copying the content of one list object into another list object(shallow copy).
- Syntyax:- listobj2= listobj1.copy()
- We have We have two types of copy mechanisma. They are

```
                a) Shallow Copy
                b) Deep Copy
```

a) shallow copy

- In this Copy process,

```
              a) Initially, Both the objects content is same
               b) Both objects memory address are different.
                c) Modification on both the objects are
Indepenedent (or) modifications are     not reflected to each other.
```
- To implement shallow copy, we use copy()
- Syntax:- listobj2=listobj1

In [92]:
```
l1=[10,"KVR","Hyd","Python"]
```

In [93]:
```
l2=l1.copy()
```

In [94]:
```
print(l1)
```
```
[10, 'KVR', 'Hyd', 'Python']
```

In [95]:
```
print(l2)
```
```
[10, 'KVR', 'Hyd', 'Python']
```

In [96]:
```
print(id(l1))
```
```
1981979482240
```

In [97]:
```
print(id(l2))
```

```
1981979414400
```

In [98]:
```
l1.append(11.11)
```

In [99]:
```
l2.insert(2,"India")
```

In [100…
```
print(l1, id(l1))
```

```
[10, 'KVR', 'Hyd', 'Python', 11.11] 1981979482240
```

In [101…
```
 print(l2, id(l2))
```

```
[10, 'KVR', 'India', 'Hyd', 'Python'] 1981979414400
```

b) Deep copy

- In this Copy process,

```
              a) Initially, Both the objects content is same
              b) Both objects memory address are Same.
              c) Modification on both the objects are
 depenedent (or) modifications are reflected to each other.
```

- To implement deep copy, we use assignment operator (=)
- Syntax:- listobj2=listobj1

In [103…
```
l1=[10,"KVR","Hyd","Python"]
```

In [104…
```
l2=l1  # DEEP COPY
```

In [105…
```
print(l1, id(l1))
```

```
[10, 'KVR', 'Hyd', 'Python'] 1981979410240
```

In [106…
```
print(l2, id(l2))
```

```
[10, 'KVR', 'Hyd', 'Python'] 1981979410240
```

In [107…
```
l1.append(22.22)
```

In [108…
```
print(l1, id(l1))
```

```
[10, 'KVR', 'Hyd', 'Python', 22.22] 1981979410240
```

In [109…
```
 print(l2, id(l2))
```

```
[10, 'KVR', 'Hyd', 'Python', 22.22] 1981979410240
```

In [110…
```
 l2.insert(2,"India")
```

In [111…
```
 print(l1, id(l1))
```

```
[10, 'KVR', 'India', 'Hyd', 'Python', 22.22] 1981979410240
```

In [112…
```
print(l2, id(l2))
```

```
[10, 'KVR', 'India', 'Hyd', 'Python', 22.22] 1981979410240
```

10. extend()

- This function is used for extending functionality of one list object with another list object. In otherwords, we can add one list elements to another list.
- Syntax:- listobj1.extend(listobj2)
- Note:- Bysuing opereator +, we can add two or more list objects into another list object.

In [113…
```
l1=[10,"Anurag","Hyd"]
```

In [114…
```
print(l1, id(l1))
```
```
[10, 'Anurag', 'Hyd'] 1981979413696
```

In [115…
```
l2=["PYTHON","JAVA","DS with AI"]
```

In [116…
```
print(l2, id(l2))
```
```
['PYTHON', 'JAVA', 'DS with AI'] 1981979428096
```

In [117…
```
 l1.extend(l2)
```

In [118…
```
 print(l1)
```
```
[10, 'Anurag', 'Hyd', 'PYTHON', 'JAVA', 'DS with AI']
```

In [119…
```
l1=[10,"Anurag","Hyd"]
```

In [120…
```
 l2=["PYTHON","JAVA","DS with AI"]
```

In [121…
```
l3=["Oracle","MySql","SQLITE3"]
```

In [122…
```
 l1=l1+l2+l3   # using + we can extend the functionality two or more list objects.
```

In [123…
```
print(l1)
```
```
[10, 'Anurag', 'Hyd', 'PYTHON', 'JAVA', 'DS with AI', 'Oracle', 'MySql', 'SQLITE3']
```

In [124…
```
print(l2)
```
```
['PYTHON', 'JAVA', 'DS with AI']
```

In [125…
```
print(l3)
```
```
['Oracle', 'MySql', 'SQLITE3']
```

11. Inner (or) Nested list

- The process of defining one list in another list is called inner or nested list.
- Syntax:

listobj=[elements of list, [inner list elements]......[inner list elements]..

- Syntax accessing the lements of inner list:

- listobj[innerlist index]

- We can apply indexing and slicing Operations.
- We can apply all the functions of list on inner list.,

In [126...
```
sl=[10,"Mukul",[20,15,16,19],[66,77,80,78],"OUCET"]
```

In [127...
```
 print(sl)
```
```
[10, 'Mukul', [20, 15, 16, 19], [66, 77, 80, 78], 'OUCET']
```

In [128...
```
print(sl[0])
```
```
10
```

In [129...
```
print(sl[1])
```
```
Mukul
```

In [130...
```
print(sl[2])
```
```
[20, 15, 16, 19]
```

In [131...
```
print(sl[-2])
```
```
[66, 77, 80, 78]
```

In [132...
```
print(sl[-1])
```
```
OUCET
```

In [133...
```
 print(sl[2][-1])
```
```
19
```

In [134...
```
 print(sl[-3][-4])
```
```
20
```

In [135...
```
 sl[2][-3]=17
```

In [136...
```
 print(sl)
```
```
[10, 'Mukul', [20, 17, 16, 19], [66, 77, 80, 78], 'OUCET']
```

In [137...
```
sl[2].append(20)
```

In [138...
```
print(sl)
```
```
[10, 'Mukul', [20, 17, 16, 19, 20], [66, 77, 80, 78], 'OUCET']
```

In [139...
```
 sl[-2].insert(1,80)
```

In [140...
```
 print(sl)
```

```
[10, 'Mukul', [20, 17, 16, 19, 20], [66, 80, 77, 80, 78], 'OUCET']
```

In [141…
```
sl[-2].count(80)
```

Out[141]:
```
2
```

Tuple

- 'tuple' is one of the pre-defined class and treated as list type data type
- The purpose of tuple type is that to store multiple values either of same type or different type or both types with unique and duplicates values.
- The elements of tuple must be written within braces ( ) and elements must seprated by comma .
- An object of tuple maintains insertion order.
- On the object of tuple , we can perform indexing and slicing operations.
- An object of tuple belongs to immutable.
- To convert one type value into tuple type, we use tuple()
- We have two types of tuple objects.
```
 a) empty tuple
 b) non-empty tuple
```

a) empty tuple:

- It does not contain any elements and whose size is 0

```
                      Syntax:-   varname= ()


                       (or)

                   varname= tuple()
```

b) non-empty tuple:

- It contain elements and whose size is > 0

```
     Syntax:-   varname= (list elements)
```

- Note:- The functionality of tuple is extactly similar to the functionality of list but an object tuple belongs to immutable and list object belongs to mutable.

In [145…
```
t1 = ()
print(t1, type(t1))
```

```
() <class 'tuple'>
```

In [146…
```
t2=tuple()
print(t2, type(t2))
```
```
() <class 'tuple'>
```

In [147…
```
len(t1)
```
Out[147]:
```
0
```

In [148…
```
len(t2)
```
Out[148]:
```
0
```

In [149…
```
 t3=(10,30,20,10,30,45)
```

In [150…
```
 print(t3, type(t3))
```
```
(10, 30, 20, 10, 30, 45) <class 'tuple'>
```

In [151…
```
t3=(10,"Rossum","hyd",34.56)
```

In [153…
```
print(t3, type(t3))
```
```
(10, 'Rossum', 'hyd', 34.56) <class 'tuple'>
```

In [155…
```
t4=110,"JG","SUN",45.67
print(t4,type(t4))
```
```
(110, 'JG', 'SUN', 45.67) <class 'tuple'>
```

In [156…
```
a=10
print(a, type(a))
```
```
10 <class 'int'>
```

In [157…
```
b=10
print(b, type(b))
```
```
10 <class 'int'>
```

In [159…
```
b=(100,200,)
print(b, type(b))
```
```
(100, 200) <class 'tuple'>
```

In [160…
```
 t3=(10,30,20,10,30,45)
```

In [163…
```
print(t3[0])
```
```
10
```

In [164…
```
print(t3[1])
```
```
30
```

In [165…
```
print(t3[5])
```
```
45
```

In [166…
```
print(t3[len(t3)-1])
```

```
45
```

Pre-defined Function in tuple

- We know that on the object of tuple we can perform Both Indexing and Slicing Operations.
- Along with these operations, we can also perform other operations by using the following pre-defined functions

```
1) index()
2) count()
```

index

In [5]:
```
t1 = (10,"RS",45.67)
print(t1,type(t1))
```

```
(10, 'RS', 45.67) <class 'tuple'>
```

In [6]:
```
t1.index(10)
```

Out[6]:
```
0
```

In [7]:
```
t1.index("RS")
```

Out[7]:
```
1
```

count

In [8]:
```
t1 = (10, "RS", 45.67)
```

In [11]:
```
print(t1,type(t1))
```

```
(10, 'RS', 45.67) <class 'tuple'>
```

In [12]:
```
t1.count(10)
```

Out[12]:
```
1
```

In [13]:
```
t1.count(100)
```

Out[13]:
```
0
```

In [15]:
```
t1=(10,0,10,10,20,0,10)
print(t1,type(t1))
```

```
(10, 0, 10, 10, 20, 0, 10) <class 'tuple'>
```

In [16]:
```
t1.count(10)
```

Out[16]:
```
4
```

In [17]:
```
t1.count(0)
```

```
Out[17]: 2
```

```
In [18]: t1.count(100)
```

```
Out[18]: 0
```

SET

- 'set' is one of the pre-defined class and treated as set data type
- The purpose of set data type is that "To store Multiple Values of Either Same Type or Different types or both the type in single object with unique (duplicate are not allowed)"
- The Elements of set must enclosed with Curly Braces {} and the elements separated by comma

- An object of set never maintains insertion order bcoz PVM displays any one of Possibility of Avaliable

  ```
  - Element in set
  ```

- On the object of Set, we can not perform Indexing and slicing Operations bcoz set object never

- maintains Insertion Order.

- An object of set belongs to Both Mutable (in the case add, and other operations also) and immutable

- in the case case set' object does not support item assignment.

- In Python Programming, we have Two Types of set objects. They are

1) Empty set 2) Non-Empty set

---

1) Empty set

- An Empty set is one , which does not contain any Elements and whose length is 0

- Syntax1: setobj1=set()

---

2) Non-Empty set

- A Non-Empty set is one , which contains Elements and whose length is > 0

- Syntax1: setobj1={Val1,Val2...Val-n}
- Syntax2: setobj1=set([object])
- Syntax3: setobj1=set((val1,val2....van-n))
- Syntax4: setobj1=set([val1,val2....van-n])
- Syntax5: setobj1=set(str)

In [19]:
```
s1={10,20,30,10,30,40,50,60}
```

In [20]:
```
print(s1,type(s1))
```
```
{50, 20, 40, 10, 60, 30} <class 'set'>
```

In [21]:
```
 len(s1)
```
Out[21]:
```
6
```

In [22]:
```
s2={10,"RS",33.33,True,3+4.5j}
```

In [23]:
```
print(s2,type(s2))
```
```
{33.33, True, 'RS', 10, (3+4.5j)} <class 'set'>
```

In [24]:
```
s2[0]
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [24], in <cell line: 1>()
----> 1 s2[0]

TypeError: 'set' object is not subscriptable
```

In [25]:
```
s2[0:4]
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [25], in <cell line: 1>()
----> 1 s2[0:4]

TypeError: 'set' object is not subscriptable
```

In [28]:
```
s2 = {10,"RS",33.33,True,3+4.5j}
```

In [29]:
```
print(s2,type(s2))
```
```
{33.33, True, 'RS', 10, (3+4.5j)} <class 'set'>
```

In [30]:
```
s2[1]=False
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [30], in <cell line: 1>()
----> 1 s2[1]=False

TypeError: 'set' object does not support item assignment
```

In [31]:
```
s2 = {10,"RS",33.33,True,3+4.5j}
```

In [32]:
```
print(s2,type(s2),id(s2))
```
```
{33.33, True, 'RS', 10, (3+4.5j)} <class 'set'> 2653557570912
```

In [33]:
```
s2.add("KVR")
```

In [34]:
```
print(s2,type(s2),id(s2))
```
```
{'KVR', 33.33, True, 'RS', 10, (3+4.5j)} <class 'set'> 2653557570912
```

In [35]:
```
s1=set()
```

In [36]:
```
print(s1,type(s1))
len(s1)
```
```
set() <class 'set'>
```
Out[36]:
```
0
```

In [37]:
```
s2={10,"abc",2.3}
```

In [38]:
```
print(s2,type(s2))
```
```
{'abc', 10, 2.3} <class 'set'>
```

In [39]:
```
len(s2)
```
Out[39]:
```
3
```

In [42]:
```
s = "MISSISSIPPI"
print(s,type(s))
```
```
MISSISSIPPI <class 'str'>
```

In [43]:
```
s1=set(s)
print(s1,type(s1))
```
```
{'I', 'P', 'M', 'S'} <class 'set'>
```

Pre-defined Functions in set

- On the object of set, we can perform Different Operations by using Pre- defined functions present in set object. They are

    1. add()

- Syntax: setobj.add(value)

- This Function is used for adding Value to the set object.

```
In [44]: s1={10,"Adesh"}
```

```
In [45]: print(s1,type(s1),id(s1))
```

```
{10, 'Adesh'} <class 'set'> 2653557570688
```

```
In [46]: s1.add("PYTHON")
```

```
In [47]: print(s1,type(s1),id(s1))
```

```
{10, 'PYTHON', 'Adesh'} <class 'set'> 2653557570688
```

```
In [48]: s1.add(22.34)
```

```
In [49]: print(s1,type(s1),id(s1))
```

```
{10, 'PYTHON', 'Adesh', 22.34} <class 'set'> 2653557570688
```

```
In [50]: s1.add(True)
```

```
In [51]: print(s1,type(s1),id(s1))
```

```
{True, 10, 'Adesh', 'PYTHON', 22.34} <class 'set'> 2653557570688
```

```
In [52]: s1=set()
```

```
In [53]: print(s1,type(s1),id(s1))
```

```
set() <class 'set'> 2653557571136
```

```
In [54]: s1.add(100)
```

```
In [55]: s1.add("Har")
```

```
In [56]: s1.add(34.56)
```

```
In [57]: s1.add("python")
```

```
In [58]: print(s1,type(s1),id(s1))
```

```
{'python', 34.56, 100, 'Har'} <class 'set'> 2653557571136
```

```
In [59]: s1.add(100)
```

```
In [60]: print(s1,type(s1),id(s1))
```

```
{'python', 34.56, 100, 'Har'} <class 'set'> 2653557571136
```

2. clear()

- Syntax: setobj.clear()
- This Function is used for removing all the Values from non-empty setobject

- If we call this Function on empty set then we get None as Result

```
In [62]: s1={34.56, 100,'Hari','Python'}
```

```
In [63]: print(s1,type(s1),id(s1))
         len(s1)
```

```
{'Hari', 34.56, 100, 'Python'} <class 'set'> 2653558366496
Out[63]: 4
```

```
In [64]: s1.clear()
```

```
In [65]: print(s1,type(s1),id(s1))
```

```
set() <class 'set'> 2653558366496
```

```
In [66]: print(s1.clear())
```

```
None
```

3. remove

- syntax: setobj.remove(Value)
- This Function is used removing the value from non-empty set object
- If the Specified Value does not Exist in set object then we get KeyError
- If we call this Function on empty set then we get KeyError

```
In [73]: s1={100,"Punit",45.67,"TS","Python"}
```

```
In [74]: print(s1,type(s1),id(s1))
```

```
{'TS', 100, 'Punit', 'Python', 45.67} <class 'set'> 2653558368736
```

```
In [75]: s1.remove("TS")
```

```
In [70]: print(s1,type(s1),id(s1))
```

```
{'Python', 100, 'Punit'} <class 'set'> 2653557571136
```

```
In [76]: s1.remove(45.67)
```

```
In [77]: print(s1,type(s1),id(s1))
```

```
{100, 'Punit', 'Python'} <class 'set'> 2653558368736
```

```
In [78]: s1.remove(100)
```

```
In [79]: print(s1,type(s1),id(s1))
```

```
{'Punit', 'Python'} <class 'set'> 2653558368736
```

```
In [80]: s1.remove(1000)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [80], in <cell line: 1>()
----> 1 s1.remove(1000)

KeyError: 1000
```

4.Discard()

- Syntax: setobj.discard(Value)
- This function is used for Removing the value from non-set object
- If the Specified Value does not Exist in set object then we get None OR Space as Result
- If we call this Function on empty set then we get None OR Space as Result

In [81]:
```
s1={100,"Punit",45.67,"TS","Python"}
```

In [82]:
```
print(s1,type(s1),id(s1))
```
```
{'TS', 100, 'Punit', 'Python', 45.67} <class 'set'> 2653558368288
```

In [83]:
```
s1.discard("TS")
```

In [84]:
```
print(s1,type(s1),id(s1))
```
```
{100, 'Punit', 'Python', 45.67} <class 'set'> 2653558368288
```

In [85]:
```
s1.discard(45.67)
```

In [86]:
```
print(s1,type(s1),id(s1))
```
```
{100, 'Punit', 'Python'} <class 'set'> 2653558368288
```

In [87]:
```
print(s1,type(s1),id(s1))
```
```
{100, 'Punit', 'Python'} <class 'set'> 2653558368288
```

In [88]:
```
s1.discard(1000)
```

In [89]:
```
print(s1,type(s1),id(s1))
```
```
{100, 'Punit', 'Python'} <class 'set'> 2653558368288
```

In [90]:
```
print(s1.discard(1000))
```
```
None
```

In [91]:
```
print(set().discard(10))
```
```
None
```

5 . pop

In [92]:
```
s1={100, "Punit", 45.67, "TS","Python"}
```

In [93]:
```
s1.pop()
```

```
Out[93]: 'TS'
```

```
In [95]: s1.pop()
```

```
Out[95]: 100
```

```
In [96]: s1.pop()
```

```
Out[96]: 'Punit'
```

```
In [97]: s1.pop()
```

```
Out[97]: 'Python'
```

```
In [98]: s1.pop()
```

```
Out[98]: 45.67
```

```
In [99]: print(s1,type(s1))
```

```
set() <class 'set'>
```

```
In [100… s1={100,'Rossum',2+3j,"NL",34.56,"Python"}
```

```
In [101… print(s1,type(s1),id(s1))
```

```
{'NL', 34.56, 100, 'Rossum', 'Python', (2+3j)} <class 'set'> 2653558368064
```

```
In [102… s1.pop()
```

```
Out[102]: 'NL'
```

```
In [103… s1.pop()
```

```
Out[103]: 34.56
```

```
In [104… s1.pop()
```

```
Out[104]: 100
```

```
In [105… s1.pop()
```

```
Out[105]: 'Rossum'
```

```
In [106… s1.pop()
```

```
Out[106]: 'Python'
```

```
In [107… s1.pop()
```

```
Out[107]: (2+3j)
```

```
In [109… print(s1,type(s1),id(s1))
```

```
set() <class 'set'> 2653558368064
```

In [110…
```
s= "ABRAKADABRA"
```

In [111…
```
s1=set(s)
```

In [112…
```
s1.pop()
```

Out[112]:
```
'B'
```

In [113…
```
print(s1,type(s1),id(s1))
```

```
{'D', 'K', 'R', 'A'} <class 'set'> 2653558369408
```

In [114…
```
s1.pop()
```

Out[114]:
```
'D'
```

In [115…
```
s1.pop()
```

Out[115]:
```
'K'
```

In [116…
```
s1.pop()
```

Out[116]:
```
'R'
```

In [117…
```
s1.pop()
```

Out[117]:
```
'A'
```

8. COPY

- Syntax: setobj2=setobj1.copy()
- This Function is used for Copying the content of one set object to another set object

In [120…
```
s1={100,"Rossum",2+3j,"NL"}
```

In [121…
```
print(s1,type(s1),id(s1))
```

```
{'Rossum', 'NL', 100, (2+3j)} <class 'set'> 2653558370080
```

In [122…
```
s2=s1.copy() # Shallow Copy
```

In [123…
```
print(s2,type(s2),id(s2))
```

```
{'Rossum', 'NL', 100, (2+3j)} <class 'set'> 2653558369632
```

In [124…
```
s1.add(12.34)
```

In [125…
```
s2.add("PYTHON")
```

In [126…
```
print(s1,type(s1),id(s1))
```

```
{100, (2+3j), 12.34, 'Rossum', 'NL'} <class 'set'> 2653558370080
```

In [127…
```
print(s2,type(s2),id(s2))
```
```
{100, (2+3j), 'PYTHON', 'Rossum', 'NL'} <class 'set'> 2653558369632
```

7. isdisjoint()

- Syntax: setobj1.isdisjoint(setobj2)
- This Function returns True provided setobj1 and setobj2 contains Different Elements (There is no common element)
- This Function returns False provided setobj1 and setobj2 contains atleast one common Element

In [128…
```
s1={10,20,30}
```
In [129…
```
s2={10,45,55}
```
In [130…
```
s3={"apple","mango"}
```
In [131…
```
print(s1)
```
```
{10, 20, 30}
```
In [132…
```
print(s2)
```
```
{10, 45, 55}
```
In [133…
```
print(s3)
```
```
{'apple', 'mango'}
```
In [134…
```
s1.isdisjoint(s2)
```
Out[134]:
```
False
```
In [135…
```
s1.isdisjoint(s3)
```
Out[135]:
```
True
```
In [136…
```
s1.isdisjoint(s3)
```
Out[136]:
```
True
```
In [137…
```
set().isdisjoint({10,20,30})
```
Out[137]:
```
True
```
In [138…
```
set().isdisjoint(set())
```
Out[138]:
```
True
```

8. issuperset()

- Syntax: setobj1.issuperset(setobj2)
- This Function returns True provided setobj1 contains all the elements of setobj2 OR all the elements of setobj2 present in setobj1
- This Function returns False provided setobj1 does not contain all the elements of setobj2 OR all the elements of setobj2 does not present in setobj1.
- Every Non-empty set and empty set are the super and sub sets to themself.

```
In [1]: s1={10,20,30}
```

```
In [2]: s2={10,20,30,40,50,60}
```

```
In [3]: s3={40,50,60,70,80,90}
```

```
In [4]: s1.issuperset(s2)
Out[4]: False
```

```
In [6]: s2.issuperset(s1)
Out[6]: True
```

```
In [7]: s2.issuperset(s1)
Out[7]: True
```

```
In [8]: s2.issuperset(s3)
Out[8]: False
```

```
In [9]: s3.issuperset(s2)
Out[9]: False
```

```
In [10]: s3.issuperset(s1)
Out[10]: False
```

```
In [11]: s3.isdisjoint(s1)
Out[11]: True
```

```
In [12]: s3.isdisjoint(s2)
Out[12]: False
```

9. issubset()

- Syntax: setobj1.issubset(setobj2)

- This Function returns True provided All the elements setobj1 present in setobj2 OR setobj2 contians all the elements setobj1
- This Function returns False provided All the elements setobj1 does not present in setobj2 OR setobj2 does not contains all the elements setobj1
- Every Non-empty set and empty set are the super and sub sets to themself.

```
In [13]: s1={10,20,30}

In [14]: s2={10,20,30,40,50,60}

In [15]: s3={40,50,60,70,80,90}

In [16]: s1.issubset(s2)

Out[16]: True

In [17]: s2.issubset(s1)

Out[17]: False

In [18]: s2.issubset(s3)

Out[18]: False

In [19]: s1.issubset(s3)

Out[19]: False

In [20]: s3.issubset(s2)

Out[20]: False

In [21]: s3.issubset(s1)

Out[21]: False

In [26]: {10,20,30}.issuperset({10,20,30})

Out[26]: True

In [27]: {10,20,30}.issubset({10,20,30})

Out[27]: True

In [28]: set().issuperset(set())

Out[28]: True

In [29]: set().issubset(set())

Out[29]: True
```

10. union()

- Syntax : setobj3=setobj1.union(setobj2)
- This Function Takes all Unique elements of setobj1 and setobj2 and placed into setobj3.

```
In [31]: s1={10,20,30,40}
```

```
In [32]: s2={40,20,50,60}
```

```
In [33]: print(s1,type(s1))
```

```
{40, 10, 20, 30} <class 'set'>
```

```
In [34]: print(s2,type(s2))
```

```
{40, 50, 20, 60} <class 'set'>
```

```
In [35]: s3=s1.union(s2)
```

```
In [36]: print(s3,type(s3))
```

```
{40, 10, 50, 20, 60, 30} <class 'set'>
```

```
In [37]: {10,20,30}.union(set())
```

```
Out[37]: {10, 20, 30}
```

11. intersection()

- Syntax : setobj3=setobj1.intersection(setobj2)
- This Function Takes all Common Unique elements of setobj1 and setobj2 and placed into setobj3.

```
In [38]: s1={10,20,30,40}
```

```
In [39]: s2={40,20,50,60}
```

```
In [40]: print(s1,type(s1))
```

```
{40, 10, 20, 30} <class 'set'>
```

```
In [41]: print(s2,type(s2))
```

```
{40, 50, 20, 60} <class 'set'>
```

```
In [42]: s3=s1.intersection(s2)
```

```
In [43]: print(s3,type(s3))
```

```
{40, 20} <class 'set'>
```

```
In [44]: {10,20,30}.intersection({15,25,35})
```

Out[44]:
```
set()
```

12. difference()

- Syntax : setobj3=setobj1.difference(setobj2)
- This Function Removes the common elements from setobj1 and setobj2 and takes Remaining Elements from setobj1 and placed into setobj3.

In [46]:
```
s1={10,20,30,40}
```

In [47]:
```
s2={40,20,50,60}
```

In [48]:
```
print(s1,type(s1))
```
```
{40, 10, 20, 30} <class 'set'>
```

In [49]:
```
print(s2,type(s2))
```
```
{40, 50, 20, 60} <class 'set'>
```

In [50]:
```
s3=s1.difference(s2)
```

In [51]:
```
print(s3,type(s3))
```
```
{10, 30} <class 'set'>
```

In [52]:
```
s4=s2.difference(s1)
```

In [53]:
```
print(s4,type(s4))
```
```
{50, 60} <class 'set'>
```

In [55]:
```
{10,20,30}.difference({"apple","mango"})
```
Out[55]:
```
{10, 20, 30}
```

In [57]:
```
{10,20,30}.difference({10,20,30})
```
Out[57]:
```
set()
```

In [58]:
```
set().difference(set())
```
Out[58]:
```
set()
```

13.symmetric_difference()

- Syntax: setobj3=setobj1.symmetric_difference(setobj2)
- This Function Removes the common elements from setobj1 and setobj2 and takes Remaining Elements from both setobj1 and setobj2 and placed into setobj3.

In [59]:
```
s1={10,20,30,40}
```

In [60]:
```
s2={40,20,50,60}
```

In [61]:
```
print(s1,type(s1))
```
```
{40, 10, 20, 30} <class 'set'>
```

print(s2,type(s2))

In [63]:
```
s3=s1.symmetric_difference(s2)
```

In [65]:
```
print(s3,type(s3))
```
```
{25, 30} <class 'set'>
```

In [66]:
```
s3=set([10,20,30]).symmetric_difference(set((10,10,20,20,25)))
```

In [68]:
```
print(s3,type(s3))
```
```
{25, 30} <class 'set'>
```

In [69]:
```
s3=set([10,20,30]).symmetric_difference(set((10,10,20,20,30)))
```

In [70]:
```
print(s3,type(s3))
```
```
set() <class 'set'>
```

14.symmetric_difference()_update()

- Syntax: setobj3=setobj1.symmetric_difference_update(setobj2)
- This Function Removes the common elements from setobj1 and setobj2 and takes Remaining Elements from both setobj1 and setobj2 and placed into setobj1 but not setobj3(setobj3 contains none)

In [71]:
```
s1={10,20,30,40}
```

In [72]:
```
s2={40,20,50,60}
```

In [73]:
```
print(s1,type(s1))
```
```
{40, 10, 20, 30} <class 'set'>
```

In [74]:
```
print(s2,type(s2))
```
```
{40, 50, 20, 60} <class 'set'>
```

In [75]:
```
s3=s1.symmetric_difference_update(s2)
```

In [76]:
```
print(s3)
```
```
None
```

In [78]:
```
print(s1)
```

```
{10, 50, 60, 30}
```

In [79]:
```
s1={10,20,30,40}
```

In [80]:
```
s2={40,20,50,60}
```

In [81]:
```
print(s1,type(s1))
```

```
{40, 10, 20, 30} <class 'set'>
```

In [82]:
```
print(s2,type(s2))
```

```
{40, 50, 20, 60} <class 'set'>
```

In [83]:
```
s2.symmetric_difference_update(s1)
print(s2,type(s2))
```

```
{10, 50, 60, 30} <class 'set'>
```

In [84]:
```
s1={10,20,30}
```

In [85]:
```
s2={"Rossum","Purosottam"}
```

In [86]:
```
print(s1)
```

```
{10, 20, 30}
```

In [87]:
```
print(s2)
```

```
{'Purosottam', 'Rossum'}
```

In [88]:
```
s1.symmetric_difference_update(s2)
```

In [89]:
```
print(s1)
```

```
{10, 'Rossum', 'Purosottam', 20, 30}
```

In [90]:
```
print(set().symmetric_difference_update(set()))
```

```
None
```

15. update()

- Syntax: setobj1.update(setobj2)
- This Function is used for adding all the elements of setobj2 to setobj1

In [91]:
```
s1={10,20,30}
```

In [92]:
```
s2={15,25,35}
```

In [94]:
```
print(s1)
```

```
{10, 20, 30}
```

In [95]:
```
print(s2)
```

```
{25, 35, 15}
```

```
In [96]: s1.update(s2)
```

```
In [97]: print(s1)
```

```
{35, 20, 25, 10, 30, 15}
```

```
In [98]: s3={10,20,30}
```

```
In [99]: s1={10,20,30}
```

```
In [100…: s1.update(s3)
```

```
In [101…: print(s1)
```

```
{20, 10, 30}
```

Frozenset

- 'frozenset' is one of the pre-defined class and treated as set data type.
- The purpose of frozenset data type is that "To store Multiple Values either Simiar Type or Different Type or Both the Types in Single Object with Unique Values".
- The elements of frozenset must be obtained from different objects like set , tuple and list.
- Syntax: frozensetobj=frozenset(set/list/tuple)
- An Object of frozenset never maintains Insertion Order bcoz PVM can display any one of the possibility of elements of frozenset object.
- On the object of frozenset, we can't perform Indexing and Slicing Operations bcoz frozenset object never maintains Insertion Order.
- An object of frozenset belongs to Immutable bcoz frozenset' object does not support item assignment and not possible to modify / Change / add.
- we can create two types of frozenset objects. They are

```
a) Empty frozenset
b) Non-Empty frozenset
```

a) Empty frozenset:

- An Empty frozenset is one, which does not contain any elements and whose length is 0
- Syntax: frozensetobj=frozenset()

b) Non-Empty frozenset :

- A Non-Empty frozenset is one, which contains elements and whose length is >0
- Syntax: frozensetobj=frozenset({ val1, val2, ...val-n })

- Syntax: frozensetobj=frozenset((val1, val2, ...val-n))
- Syntax: frozensetobj=frozenset([val1, val2, ...val-n])

-

```
In [1]: s1={10,20,30,40,10}
```

```
In [2]: print(s1,type(s1))
```

```
{40, 10, 20, 30} <class 'set'>
```

```
In [3]: fs=frozenset(s1)
```

```
In [4]: print(fs,type(fs))
```

```
frozenset({40, 10, 20, 30}) <class 'frozenset'>
```

```
In [5]: t1=(10,"Rossum",34.56,"Python")
```

```
In [6]: fs=frozenset(t1)
```

```
In [7]: print(fs,type(fs))
```

```
frozenset({10, 34.56, 'Python', 'Rossum'}) <class 'frozenset'>
```

```
In [8]: len(fs)
```

```
Out[8]: 4
```

```
In [9]: fs1=frozenset()
```

```
In [10]: print(fs1,type(fs1))
```

```
frozenset() <class 'frozenset'>
```

```
In [11]: len(fs1)
```

```
Out[11]: 0
```

```
In [14]: fs
```

```
Out[14]: frozenset({10, 34.56, 'Python', 'Rossum'})
```

Pre-Defined Functions in frozenset

- frozenset contains the following Functions

```
a) copy()
b) isdisjoint()
c) issuperset()
d) issubset()
e) union()
f) intersection()
```

```
g) difference()
h) symmertic_difference()
```

1. copy

```
In [16]: fs1 = {50, 20, 70, 40, 10, 60, 30}
```

```
In [17]: print(fs1,type(fs1),id(fs1))
```

```
{50, 20, 70, 40, 10, 60, 30} <class 'set'> 3110989834496
```

```
In [18]: fs2=fs1.copy()
```

```
In [19]: print(fs2,type(fs2),id(fs2))
```

```
{50, 20, 70, 40, 10, 60, 30} <class 'set'> 3110989835168
```

```
In [20]: print(fs1)
```

```
{50, 20, 70, 40, 10, 60, 30}
```

2.issuperset

```
In [21]: fs1=frozenset({10,20,30,40,50,60,70})
```

```
In [22]: fs2=frozenset((10,20,30))
```

```
In [23]: fs1.issuperset(fs2)
```

```
Out[23]: True
```

```
In [24]: fs2.issuperset(fs1)
```

```
Out[24]: False
```

```
In [25]: fs2.issubset(fs1)
```

```
Out[25]: True
```

```
In [26]: fs1.issubset(fs2)
```

```
Out[26]: False
```

3. disjoint

```
In [28]: fs1=frozenset({10,20,30,40,50,60,70})
```

```
In [29]: fs2=frozenset((100,200,300))
```

```
In [30]: fs3=frozenset((10,2,3))
```

```
In [31]: fs1.isdisjoint(fs2)
```

```
Out[31]: True
```

```
In [32]: fs1.isdisjoint(fs3)
Out[32]: False
```

```
In [33]: print(fs1)
frozenset({50, 20, 70, 40, 10, 60, 30})
```

```
In [34]: print(fs2)
frozenset({200, 100, 300})
```

4. Union

```
In [35]: fs1.union(fs2)
Out[35]: frozenset({10, 20, 30, 40, 50, 60, 70, 100, 200, 300})
```

5. intersection

```
In [37]: fs1.intersection(fs2)
Out[37]: frozenset()
```

6.difference

```
In [38]: fs1.difference(fs2)
Out[38]: frozenset({10, 20, 30, 40, 50, 60, 70})
```

```
In [39]: fs2.difference(fs1)
Out[39]: frozenset({100, 200, 300})
```

```
In [40]: frozenset({10,20,30,40}).symmetric_difference(frozenset([10,20,50,60]) )
         frozenset({40, 50, 60, 30})
Out[40]: frozenset({30, 40, 50, 60})
```

```
In [41]: fs1|fs2
Out[41]: frozenset({10, 20, 30, 40, 50, 60, 70, 100, 200, 300})
```

Dict Categeory Data Types

- 'dict' is one of the pre-defined class and treated as Dict Data Type.
- The purpose of dict data type is that "To store (Key,value) in single variable
- In (Key,Value), the values of Key is Unique and Values of Value may or may not be unique.

- The (Key,value) must be organized or stored in the object of dict within Curly Braces {} and they separated by comma.
- An object of dict does not support Indexing and Slicing bcoz Values of Key itself considered as Indices.
- In the object of dict, Values of Key are treated as Immutable and Values of Value are treated as mutable.
- Originally an object of dict is mutable bcoz we can add (Key,Value) externally.
- We have two types of dict objects. They are

```
a) Empty dict
b) Non-empty dict
```

a) Empty dict

- Empty dict is one, which does not contain any (Key,Value) and whose length is 0
- Syntax:- dictobj1= { } or dictobj=dict()
- Syntax for adding (Key,Value) to empty dict:

```
 dictobj[Key1]=Val1
dictobj[Key2]=Val2
dictobj[Key-n]=Val-n
```

- Here Key1,Key2...Key-n are called Values of Key and They must be Unique
- Here Val1, Val2...Val-n are called Values of Value and They may or may not be unique.

b) Non-Empty dict

- Non-Empty dict is one, which contains (Key,Value) and whose length is >0
- Syntax:- dictobj1= { Key1:Val1,Key2:Val2......Key-n:Valn}
- Here Key1,Key2...Key-n are called Values of Key and They must be Unique
- Here Val1, Val2...Val-n are called Values of Value and They may or may not be unique.

In [45]:
```
d1={10:"Python",20:"Data Sci",30:"Django"}
```

In [46]:
```
print(d1,type(d1))
```

```
{10: 'Python', 20: 'Data Sci', 30: 'Django'} <class 'dict'>
```

In [47]:
```
d2={10:3.4,20:4.5,30:5.6,40:3.4}
```

In [48]:
```
print(d2,type(d2))
```

```
{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4} <class 'dict'>
```

```
In [49]: len(d1)

Out[49]: 3

In [50]: len(d2)

Out[50]: 4

In [51]: d3 = {}

In [52]: print(d3,type(d3))

{} <class 'dict'>

In [53]: len(d3)

Out[53]: 0

In [54]: d4=dict()

In [55]: print(d4,type(d4))

{} <class 'dict'>

In [56]: len(d4)

Out[56]: 0

In [57]: d2={10:3.4,20:4.5,30:5.6,40:3.4}

In [58]: print(d2)

{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4}

In [60]: d2[10]

Out[60]: 3.4

In [66]: d2[10]=10.44

In [67]: print(d2)

{10: 10.44, 20: 4.5, 30: 5.6, 40: 3.4, 50: 5.5}

In [68]: d2={10:3.4,20:4.5,30:5.6,40:3.4}

In [69]: print(d2,type(d2),id(d2))

{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4} <class 'dict'> 3110961494912

In [70]: d2[50]=5.5

In [71]: print(d2,type(d2),id(d2))

{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4, 50: 5.5} <class 'dict'> 3110961494912
```

```
In [72]: d3={}
         print(d3,type(d3),id(d3))
```
```
{} <class 'dict'> 3110982192128
```
```
In [73]: d4=dict()
```
```
In [74]: print(d4,type(d4),id(d4))
```
```
{} <class 'dict'> 3110990217856
```
```
In [75]: d4[10]= "Apple"
```
```
In [76]: d4[20]= "Mango"
```
```
In [77]: d4[30]= "Kiwi"
```
```
In [78]: d4[40] = "Sberry"
```
```
In [79]: d4[50]= "Orange"
```
```
In [80]: print(d4,type(d4),id(d4))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi', 40: 'Sberry', 50: 'Orange'} <class 'dict'> 311
0990217856
```
```
In [81]: d4[10] = "Guava"
```
```
In [82]: print(d4,type(d4),id(d4))
```
```
{10: 'Guava', 20: 'Mango', 30: 'Kiwi', 40: 'Sberry', 50: 'Orange'} <class 'dict'> 311
0990217856
```
```
In [83]: d2={10:3.4,20:4.5,30:5.6,40:3.4}
```
```
In [84]: print(d2,type(d2),id(d2))
```
```
{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4} <class 'dict'> 3110989760512
```
```
In [85]: d2[50]=1.2
```
```
In [86]: print(d2,type(d2),id(d2))
```
```
{10: 3.4, 20: 4.5, 30: 5.6, 40: 3.4, 50: 1.2} <class 'dict'> 3110989760512
```

Pre-Defined Functions in dict

- On the object of dict, we perfomed Inserting (Key,value), Update value of Value by passing
- Value of Key and we retrieved value of Value by passing Value of Key.
- Along with the above Operations, we can also perform Various Operations by using Pre-defined functions Present in dict object. They are

1) clear()

- Syntax: dictobj.clear()
- This Function is used for Removing all the Elements from non-empty dict object.
- if we this function on empty dict object then we get None OR Space as Result

In [88]:
```
d1={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [89]:
```
print(d1,type(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'>
```

In [90]:
```
len(d1)
```
Out[90]:
```
3
```

In [91]:
```
d1.clear()
```

In [92]:
```
print(d1,type(d1))
```
```
{} <class 'dict'>
```

In [93]:
```
len(d1)
```
Out[93]:
```
0
```

In [97]:
```
print(d1.clear())
```
```
None
```

In [98]:
```
print({}.clear())
```
```
None
```

In [99]:
```
print(dict().clear())
```
```
None
```

2.copy

- Syntax: dictobj2=dictobj1.copy()
- This Function is used for Copying the content of dictobj1 into dictobj2 (Implements Shallow copy)

In [101…
```
 d1={10:"Apple", 20: "Mango", 30: "Kiwi"}
```

In [102…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990299456
```

In [103…
```
d2=d1.copy()
```

In [104…
```
print(d2,type(d2),id(d2))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990212288
```

In [105…
```
d1[10]= "Guava"
```

In [106…
```
d2[10]= "Sberry"
```

In [107…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Guava', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990299456
```

In [108…
```
print(d2,type(d2),id(d2))
```
```
{10: 'Sberry', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990212288
```

In [109…
```
d1={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [110…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990072128
```

In [111…
```
d2=d1 # Deep Copy
```

In [112…
```
print(d2,type(d2),id(d2))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990072128
```

pop()

- Syntax: dictobj.pop(Key)
- This Funcrtion is used for Removing (Key,Value) from non-empty object by passing Value of keys
- If the Key does not exist in empty or non-empty dict objects then we get KeyError.

In [113…
```
d1={10: "Apple", 20: "Mango", 30:"Kiwi"}
```

In [114…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110989767872
```

In [115…
```
d1.pop(20)
```
Out[115]:
```
'Mango'
```

In [116…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 30: 'Kiwi'} <class 'dict'> 3110989767872
```

In [117…
```
d1.pop(30)
```
Out[117]:
```
'Kiwi'
```

In [118…
```
print(d1,type(d1),id(d1))
```

```
{10: 'Apple'} <class 'dict'> 3110989767872
```

In [119…
```
d1.pop(10)
```
Out[119]:
```
'Apple'
```

In [120…
```
print(d1,type(d1),id(d1))
```
```
{} <class 'dict'> 3110989767872
```

4) popitem()

- Syntax: dictobj.popitem()
- This Function is used for Removing last (key,value) from non-empty dict object.
- If we call this function on empty dict object then we get KeyError

In [121…
```
d1={10: "Apple", 20: "Mango", 30:"Kiwi"}
```

In [122…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990659648
```

In [123…
```
d1.popitem()
```
Out[123]:
```
(30, 'Kiwi')
```

In [124…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango'} <class 'dict'> 3110990659648
```

In [125…
```
d1.popitem()
```
Out[125]:
```
(20, 'Mango')
```

In [126…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple'} <class 'dict'> 3110990659648
```

In [127…
```
d1.popitem()
```
Out[127]:
```
(10, 'Apple')
```

In [128…
```
print(d1,type(d1),id(d1))
```
```
{} <class 'dict'> 3110990659648
```

get()

- Syntax1: varname=dictobj.get(key)

- This function is used for obtaining Value of Value by passing value of Key
- If the value of Key exist in non-empty dict then LHS Var name contains Value of Value which is corresponding to the value of Key

- If the value of Key does not exist in non-empty dict then LHS Var name contains None
- Syntax2: dictobj[Key]

```
This Syntax also Gives Value of Value by passing Value of Key
This Syntsx gives KeyError when Value of Key does not exist.
```

In [130…
```
d1={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [131…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110989919040
```

In [132…
```
v1=d1.get(10)
```

In [133…
```
print(v1)
```
```
Apple
```

In [134…
```
v1=d1.get(20)
```

In [135…
```
print(v1)
```
```
Mango
```

In [136…
```
v1=d1.get(100)
```

In [137…
```
print(v1)
```
```
None
```

In [138…
```
d1={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [139…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990659840
```

In [140…
```
d1[10]
```
Out[140]:
```
'Apple'
```

In [141…
```
d1[20]
```
Out[141]:
```
'Mango'
```

keys

- Syntax: varname=dictobj.keys()
- This Function is used for obtaining List of Values of Key and stored in LHS Varname and whose type is <class,'dict_keys'>

In [142…
```
d1={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [143…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990713856
```

In [144…
```
ks=d1.keys()
```

In [145…
```
print(ks,type(ks))
```
```
dict_keys([10, 20, 30]) <class 'dict_keys'>
```

In [146…
```
for k in ks:
    print(k)
```
```
10
20
30
```

In [147…
```
for k in d1.keys():
    print(k)
```
```
10
20
30
```

In [148…
```
for k in d1.keys():
    print(k,d1.get(k))
```
```
10 Apple
20 Mango
30 Kiwi
```

In [149…
```
for k in d1.keys():
    print(k,d1[k])
```
```
10 Apple
20 Mango
30 Kiwi
```

values()

- Syntax: varname=dictobj.values()
- This Function is used for obtaining List of Values of Values and stored in LHS Varname and whose type is <class,'dict_values'>

In [153…
```
d1={10: "Apple", 20: "Mango", 30:"Kiwi"}
```

In [154…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990714496
```

In [155…
```
vs=d1.values()
```

In [156…
```
print(vs,type(vs))
```
```
dict_values(['Apple', 'Mango', 'Kiwi']) <class 'dict_values'>
```

In [157…
```
for val in vs:
    print(val)
```
```
Apple
Mango
Kiwi
```

In [158…
```
for val in d1.values():
    print(val)
```
```
Apple
Mango
Kiwi
```

In [159…
```
for val in d1.values():
    print(val,d1.get(val))
```
```
Apple None
Mango None
Kiwi None
```

items()

- Syntax: varname=dictobj.items()
- This Function is used for for obtaining List of (Key,value) from non-empty dict object and stored in LHS Varname and whose type is <class,'dict_items'>

In [160…
```
d1={10:"Apple", 20: "Mango", 30: "Kiwi"}
```

In [161…
```
print(d1,type(d1),id(d1))
```
```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'} <class 'dict'> 3110990654080
```

In [162…
```
kvs=d1.items()
```

In [163…
```
print(kvs,type(kvs))
```
```
dict_items([(10, 'Apple'), (20, 'Mango'), (30, 'Kiwi')]) <class 'dict_items'>
```

In [164…
```
for kv in kvs:
    print(kv)
```
```
(10, 'Apple')
(20, 'Mango')
(30, 'Kiwi')
```

In [165…
```
for kv in d1.items():
    print(kv)
```
```
(10, 'Apple')
(20, 'Mango')
(30, 'Kiwi')
```

In [166…
```
for k,v in d1.items():
    print(k,"---->",v)
```

```
10 ----> Apple
20 ----> Mango
30 ----> Kiwi
```

update()

- Sytntax: dictobj1.update(dictobj2)
- This Function used for Merging (adding, updating) the content of dictobj1 with dictobj2.

In [167...
```
d1={10:"apple"}
```

In [168...
```
d2={20:"mango"}
```

In [169...
```
print(d1,id(d1))
```

```
{10: 'apple'} 3110990715520
```

In [170...
```
print(d2,id(d2))
```

```
{20: 'mango'} 3110990645696
```

In [171...
```
d1.update(d2)
```

In [172...
```
print(d1,id(d1))
```

```
{10: 'apple', 20: 'mango'} 3110990715520
```

In [173...
```
print(d2,id(d2))
```

```
{20: 'mango'} 3110990645696
```

In [174...
```
d1={10:"apple"}
```

In [176...
```
d2={20:"mango",10:"Guava"}
```

In [177...
```
d1.update(d2)
```

In [179...
```
print(d1,id(d1))
```

```
{10: 'Guava', 20: 'mango'} 3110990456448
```

In [180...
```
d1={10:"apple"}
```

In [181...
```
d2={10:"mango"}
```

In [182...
```
print(d1,id(d1))
```

```
{10: 'apple'} 3110990641024
```

In [183...
```
d2.update(d1)
```

In [184...
```
print(d2,id(d2))
```

```
{10: 'apple'} 3110990400384
```

In [185…
```
d1={}
```

In [186…
```
d2={10: "Apple", 20: "Mango", 30: "Kiwi"}
```

In [187…
```
d1.update(d2)
```

In [188…
```
print(d1)
```

```
{10: 'Apple', 20: 'Mango', 30: 'Kiwi'}
```

In [ ]: