List

- It is one of the most used datatype in python and is very flexible.
- The purpose of List datatype is that "To store multiple values either Similar Type or Different Type or Both the Types in Single Object with Unique and Duplicate Values".
- The elements are must be enclosed or written within Square Brackets [] and the elements must be separated by comma.
- Syntax: list obj=[val1, val2, ...val-n]
- An object of list maintains Insertion Order.
- On the object of list, we can perform both Indexing and Slicing Operations.

Empty List

- An Empty List is one, which does not contain any elements and whose length is 0.
- To create an empty list, we can use empty square brackets or use the list() function with no arguments.
- Syntax: listobj=[]

(OR)

listobj=list()

In [1]:
```
l = []   # Empty list
l
```

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

In [2]:
```
l = list()    # Empty list
l
```

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

Non-Empty List

- Each element or value that is inside a list is called as item.
- All the items in the list may be same type (homogeneous) or different type (heterogeneous).

In [3]:
```
l1 = [1,2,3,4]     # List is homogeneous.
l1
```

Out[3]:
```
[1, 2, 3, 4]
```

In [4]:
```
l1 = [1,2,3,4]     # List is homogeneous.
print(l1,type(l1))
```

```
[1, 2, 3, 4] <class 'list'>
```

In [5]:
```
l2 = [1,2.4,3+4j,True,"New"]     # List is heterogeneous.
print(l2,type(l2))
```

```
[1, 2.4, (3+4j), True, 'New'] <class 'list'>
```

Length of List

In [6]:
```
my_list = [10,"New",20.3,False]
len(my_list)
```

Out[6]: 4

Nested List

In [7]:
```
nest = [1,[2,3],4,5]
print(nest)
```

```
[1, [2, 3], 4, 5]
```

In [9]:
```
len(nest)
```

Out[9]: 4

List Indexing

- If we specify invalid index, we get IndexError.
- If we specify indexing on int, float, bool, complex datatypes within the list then we get TypeError

In [10]:
```
nest = [1,[2,3],4,5]
print(nest)
```

```
[1, [2, 3], 4, 5]
```

In [11]:
```
nest[0]
```

Out[11]: 1

In [12]:
```
nest[1]
```

Out[12]: [2, 3]

In [13]:
```
nest[2]
```

Out[13]: 4

In [14]:
```
nest[3]
```

Out[14]: 5

In [15]:
```
nest[1][0]
```

Out[15]: 2

In [16]:
```
nest[1][1]
```

Out[16]: 3

If we specify invalid index, we get IndexError.

In [20]:
```
nest[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\3372124199.py in <module>
----> 1 nest[5]

IndexError: list index out of range
```

If we specify indexing on int, float, bool, complex datatypes within the list then we get TypeError.

In [21]:
```
nest[2][0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\999881641.py in <module>
----> 1 nest[2][0]

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

In [22]:
```
a = [2,4,[1,3,[8,"Hello",6]],9,88]
len(a)
```

Out[22]: 5

In [23]:
```
# We want to extract "Hello"
a[2][2][1]
```

Out[23]: 'Hello'

Lists are Mutable

- If you are able to modify the values within the same object then it is called "Mutable".

In [24]:
```
a = [1,2,3]
print(a)

id(a)
```

```
[1, 2, 3]
```

Out[24]: 2238321612864

In [25]:
```
a[1]=40
print(a)

id(a)
```

```
[1, 40, 3]
```

Out[25]: 2238321612864

List Slicing

In [26]:
```
numbers = [10,20,30,40,50,60,70,80,90]
numbers[0:6]
```

Out[26]: [10, 20, 30, 40, 50, 60]

List Concatenation

In [27]:
```
l1 = ["das","a","b"]
l2 = [1,"Ram",5.7]
l3=l1+l2
print(l3)
```

```
['das', 'a', 'b', 1, 'Ram', 5.7]
```

In [28]:
```
l1 = ["das","a","b"]
l2 = [1,"Ram",5.7]
l3=l2+l1
print(l3)
```

```
[1, 'Ram', 5.7, 'das', 'a', 'b']
```

List Methods

list.append()

- append is used for adding the elements at the end of existing elements of list.
- append() takes a single element as argument and the length of the list is increased by 1.
- If we try to specify multiple elements in list by append(), then we get TypeError.

Syntax : - list_name.append(element)

In [29]:
```
my_list = [10,20,30,40]
my_list.append(50)
print(my_list)
```

```
[10, 20, 30, 40, 50]
```

In [31]:
```
lst = [1,2,3,4]
lst.append([5,6,7])
print(lst)
```

```
[1, 2, 3, 4, [5, 6, 7]]
```

In [32]:
```
my_list = [10,20,30,40]
my_list.append(50,60,70)
print(my_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\1102635847.py in <module>
      1 my_list = [10,20,30,40]
----> 2 my_list.append(50,60,70)
      3 print(my_list)

TypeError: list.append() takes exactly one argument (3 given)
```

list.extend()

- extend() can add multiple individual elements to the end of the list.
- extend() takes an iterable as argument (list, tuple, dictionaries, sets, strings) and increases the length of the list by the number of elements of the iterable passed as an argument.

Syntax :- list_name.extend(iterable)

In [33]:
```
lst = [1,2,3,4]
lst.extend([5,6,7])
print(lst)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

In [35]:
```
lst = [1,2,3,4]
lst.extend([8,[5,6,7],9,[11,12]])
print(lst)
```

```
[1, 2, 3, 4, 8, [5, 6, 7], 9, [11, 12]]
```

list.insert()

- This method can be used to insert a value at any desired position.
- It takes two arguments-element and the index at which the element has to be inserted and Length of the list increases by 1.
- If we specify Invalid Index then the element will be added at the end of existing elements of list.

Syntax: - list_name.insert(index,element)

In [40]:
```
lst = ["one","two","four"]
lst.insert(2,"three")
print(lst)
```

```
['one', 'two', 'three', 'four']
```

list.remove()---(Based On Value / Element)

- This Function is used for Removing First Occurrence of list object.
- If the specified element does not exist in list object then we get ValueError.

Syntax: - listobj.remove(Element)

In [39]:
```
numbers = [10,20,30,40,10]
numbers.remove(10)
print(numbers)
```

```
[20, 30, 40, 10]
```

list.pop()

- The pop() method removes the element at the specified position based on index number.
- If we not specify any index in pop(), then it will removes end element of the list.
- If we specify invalid Index, then we get IndexError.

Syntax: -

- listobj.pop(index)

  (OR)

- listobj.pop()

In [43]:
```
numbers = [10,20,30,40,10]
numbers.pop(2)
print(numbers)
```

```
[10, 20, 40, 10]
```

In [44]:
```
numbers = [10,20,30,40,10]
numbers.pop(7)
print(numbers)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\2128187119.py in <module>
      1 numbers = [10,20,30,40,10]
----> 2 numbers.pop(7)
      3 print(numbers)

IndexError: pop index out of range
```

In [46]:
```
lst = [10,"Ram",20,"Siva",30]
print(lst)
```

```
[10, 'Ram', 20, 'Siva', 30]
```

In [47]:
```
lst = [10,"Ram",20,"Siva",30]
lst.pop()
print(lst)
```

```
[10, 'Ram', 20, 'Siva']
```

In [48]:
```
lst.pop()
print(lst)
```

```
[10, 'Ram', 20]
```

In [49]:
```
lst.pop()
print(lst)
```

```
[10, 'Ram']
```

del operator

- "del" operator is used for removing the elements of any object based on Indexing, Slicing Operation.
- It can remove entire object at once only.
- After deleting entaire object, if we try to print that object then we get NameError.

Syntax1:

- del lisobj[Index]

  (OR)

- del listobj[Begin:End:Step]
  (OR)
- del listobj

In [90]:
```
l1=[10,10,10,20,20,30,40,40]
print(l1)
```

```
[10, 10, 10, 20, 20, 30, 40, 40]
```

In [94]:
```
del l1[1]
print(l1)
```

```
[10, 20, 20, 30, 40, 40]
```

In [95]:
```
del l1[0:3]
print(l1)
```

```
[30, 40, 40]
```

In [96]:
```
del l1
print(l1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\659096169.py in <module>
      1 del l1
----> 2 print(l1)

NameError: name 'l1' is not defined
```

DIfference between del Vs pop()

del	pop()
- Removes multiple items at time.	- Removes only one item at a time.

DIfference between del Vs clear()

del	clear()
- It will delete the whole object and data.	- It will delete the items (data) not object.

list.clear()

- The clear() method clears all the elements from the list and returns empty list.

Syntax: - listobj.clear()

In [51]:
```
a = [1,2,3,4]
a.clear()
print(a)
```

```
[]
```

In [52]:
```
a = [1,2,3,4]
a.clear()
a.extend([5,6,7,8])
print(a)
```

```
[5, 6, 7, 8]
```

list.reverse()

- This Function is used for obtaining Reverse of given list elements (Front to back and Back to front).

Syntax: - listobj.reverse()

In [54]:
```
a = [1,2,3,4]
a.reverse()
print(a)
```

```
[4, 3, 2, 1]
```

In [55]:
```
a = [1,2.8,[3,4],6]
a.reverse()
print(a)
```

```
[6, [3, 4], 2.8, 1]
```

list,sort()

- This Function is used for Sorting the data either in Ascending Order(reverse=False) or Descending Order (reverse=True).
- It will sort the data Similar type of Data otherwise we get TypeError.

Syntax: -

- listobj.sort()

  (OR)

- listobj.sort(reverse=True / False)

In [64]:
```
a = [4,3,2,1,8]
a.sort()                    # Ascending Order
print(a)
```

```
[1, 2, 3, 4, 8]
```

In [62]:
```
a = [4,3,2,1,8.3]
a.sort(reverse=True)      # Descending Order
print(a)
```

```
[8.3, 4, 3, 2, 1]
```

In [63]:
```
a = [4,3,2,1,8.3]
a.sort(reverse=False)      # Ascending Order
print(a)
```

```
[1, 2, 3, 4, 8.3]
```

In [71]:
```
a = ["ram","a","siva","b","c"]
a.sort(reverse=False)      # Ascending Order
print(a)
```

```
['a', 'b', 'c', 'ram', 'siva']
```

In [72]:
```
a = ["ram","a","siva","b","c"]
a.sort(reverse=True)      # Ascending Order
print(a)
```

```
['siva', 'ram', 'c', 'b', 'a']
```

In [73]:
```
a = ["Ram","amit","Siva","bala","car"]        # Case Sensitive
a.sort(reverse=True)      # Descending Order
print(a)
```

```
['car', 'bala', 'amit', 'Siva', 'Ram']
```

Note:- sort() is applicable for either only numeric values or alphabet.

In [74]:
```
lst = [10,20.45,"ram",40]
lst.sort()
print(lst)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\2335402968.py in <module>
      1 lst = [10,20.45,"ram",40]
----> 2 lst.sort()
      3 print(lst)

TypeError: '<' not supported between instances of 'str' and 'float'
```

Difference between list.sort() Vs sorted(list)

list.sort()

- After sorting it will stores in the original list.

sorted(list)

- After sorting it will print the value, but not stores in the original list.

Difference between list.reverse() Vs reversed(list)

list.sort()

- After reversing it will stores in the original list.

sorted(list)

- After reversing it will print the value, but not stores in the original list.

---

list.count()

- This Function is used for finding number of Occurrences of Specified Element / value.
- If specified value is not present in the list, then it will returns 0.

Syntax: listobj.count(Value)

In [85]:
```
lst = [1,20,30,10,4,30,20,8,7,20,30]
print(lst.count(20))
```

```
3
```

In [86]:
```
lst = [1,20,30,10,4,30,20,8,7,20,30]
print(lst.count(30))
```

```
3
```

In [87]:
```
lst = [1,20,30,10,4,30,20,8,7,20,30]
print(lst.count(1))
```

```
1
```

In [88]:
```
lst = [1,20,30,10,4,30,20,8,7,20,30]
print(lst.count(200))
```

```
0
```

copy()

Shallow copy

The Properties of Shallow Copy are,

- Initial Content of Both objects is Same.
- The Memory Address of Both objects is Different.

- Modification of Both objects are Independent (The changes made to the objects are not reflecting to each other).
- To implement Shallow Copy Technique, we always use copy().

Syntax:- destobj=srcobj.copy()

In [101]:
```
l1=[10,20,30]
print(l1,type(l1),id(l1))
```

```
[10, 20, 30] <class 'list'> 2238322099264
```

In [102]:
```
l2 = l1.copy()
print(l2,type(l2),id(l2))
```

```
[10, 20, 30] <class 'list'> 2238322096064
```

In [103]:
```
l1.append("srk")
l2.append(5000)
print(l1,type(l1),id(l1))
print(l2,type(l2),id(l2))
```

```
[10, 20, 30, 'srk'] <class 'list'> 2238322099264
[10, 20, 30, 5000] <class 'list'> 2238322096064
```

Deep copy

The Properties of Deep Copy are,

- Initial Content of Both objects is Same.
- The Memory Address of Both objects is SAME.
- Modification of Both objects are Dependent (The changes made to the objects are reflecting to each other).
- To implement Deep Copy Technique, we always use Assignment Operator " = "

Syntax:- destobj=srcobj

In [108]:
```
l1=[10,20,30]
print(l1,type(l1),id(l1))
```

```
[10, 20, 30] <class 'list'> 2238321140032
```

In [109]:
```
l2 = l1
print(l2,type(l2),id(l2))
```

```
[10, 20, 30] <class 'list'> 2238321140032
```

In [110]:
```
l1.append("srk")
l2.append(2000)
print(l1,type(l1),id(l1))
print(l2,type(l2),id(l2))
```

```
[10, 20, 30, 'srk', 2000] <class 'list'> 2238321140032
[10, 20, 30, 'srk', 2000] <class 'list'> 2238321140032
```

list.index()

- This Function is used for finding Index of First Occurrence of Specified Element / Value.
- If the value not existing then we get ValueError.

Syntax: listobj.index(Value)

In [124]:
```
l1=[10,20,30,40,50,10,20]
print(l1)
```

```
[10, 20, 30, 40, 50, 10, 20]
```

In [125]:
```
l1.index(10)
```

Out[125]:
```
0
```

In [126]:
```
l1.index(20)
```

Out[126]:
```
1
```

In [127]:
```
l1.index(30)
```

Out[127]:
```
2
```

In [128]:
```
l1.index(80)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5356\2665505524.py in <module>
----> 1 l1.index(80)

ValueError: 80 is not in list
```

Tuples

- Purpose of tuple data type is that "To store Multiple Values either Similar Type or Different Type or Both the Types in Single Object with Unique and Duplicate Values".
- The elements of tuple must be enclosed or written within Braces ( ) and the elements must be separated by comma.
- Tuple is similar to list except tuples are immutable which means we cannot change the elements of tuple once assigned.
- When we do not want to change the data over time, tuple data type is preferred.
- Iterating over the elements of a tuple is faster as compared to iterating over a list.
- An object of tuple maintains Insertion Order.
- On the object of tuple, we can perform both Indexing and Slicing Operations.

Syntax: - tplobj=(val1, val2, ...val-n).

Empty tuple

- An Empty tuple is one, which does not contain any elements and whose length is 0.

Syntax:

```
tupleobj=()
  (OR)
tupleobj=tuple()
```

In [4]:
```
tup1 = ()   # Empty tuple
print(tup1)
type(tup1)
```

```
()
```

Out[4]:
```
tuple
```

In [9]:
```
len(tup1)
```

Out[9]:
```
0
```

In [5]:
```
tup2 = tuple()   # Empty tuple
print(tup2)
type(tup2)
```

```
()
```

Out[5]:
```
tuple
```

In [10]:
```
len(tup2)
```

Out[10]:
```
0
```

Non Empty tuple

- A Non-Empty tuple is one, which contains elements and whose length is >0.

Syntax:

```
tupleobj=(val1, val2, ....val-n )

        (OR)

tupleobj=val1, val2, ....val-n
```

In [6]:
```
t = (10,12.45,True,3+8j,"Hari")   # tuple is heterogeneous
t
```

Out[6]:
```
(10, 12.45, True, (3+8j), 'Hari')
```

In [7]:
```
t1=(10,20,30,40,10,10,20)
print(t1,type(t1))
```

```
(10, 20, 30, 40, 10, 10, 20) <class 'tuple'>
```

In [8]:
```
len(t1)
```

Out[8]:
```
7
```

tuple indexing and slicing

In [11]:
```
t2=(10,"Laxmi",88.88,"OUCET", True)
print(t2,type(t2))
```

```
(10, 'Laxmi', 88.88, 'OUCET', True) <class 'tuple'>
```

In [12]:
```
t2[1]
```

Out[12]:
```
'Laxmi'
```

In [13]:
```
t2[4]
```

Out[13]:
```
True
```

In [14]:
```
t2[0:5]
```

Out[14]:
```
(10, 'Laxmi', 88.88, 'OUCET', True)
```

In [15]:
```
t2[2:4]
```

Out[15]:
```
(88.88, 'OUCET')
```

Nested tuple

In [18]:
```
x=(10,"UmaMaheswari",66.66,[20,"Ram"],"JNTU",(50,60))
print(x,type(x))
```

```
(10, 'UmaMaheswari', 66.66, [20, 'Ram'], 'JNTU', (50, 60)) <class 'tuple'>
```

tuples are immutable

In [19]:
```
t1=(10,20,30,40,10,10,20)
print(t1,type(t1))
```

```
(10, 20, 30, 40, 10, 10, 20) <class 'tuple'>
```

In [20]:
```
t1[0]=100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\469800251.py in <module>
----> 1 t1[0]=100

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

In [21]:
```
del t1[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\2483702605.py in <module>
----> 1 del t1[1]

TypeError: 'tuple' object doesn't support item deletion
```

In [22]:
```
del t1          # Deleting entaire object is possible.
```

Interview Question

In [24]:
```
tup=(1,2,3,[4,5],(10,11,12))
print(tup)
```

```
(1, 2, 3, [4, 5], (10, 11, 12))
```

In [25]:
```
tup[4][1]=15        # We are unable to modify the values within tuple of tuple (tu
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\3695194722.py in <module>
----> 1 tup[4][1]=15

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

In [27]:
```
tup[3][1]=15        # We are able to modify the values within tuple of list (list
print(tup)
```

```
(1, 2, 3, [4, 15], (10, 11, 12))
```

Tuple Methods

In [28]:
```
t1=(10,20,30,40,10,10,20)
print(t1)
```

```
(10, 20, 30, 40, 10, 10, 20)
```

tupleobj.count(value)

- Number of times item occured in the tuple.

In [30]:
```
t1.count(10)      # Number of times item (10) occured in the tuple.
```

Out[30]:
```
3
```

tupleobj.index()

- It returns index of first occurance of the element.

In [32]:
```
t1.index(10)   # Index of first element equal to 10
```

Out[32]:
```
0
```

sorting

In [34]:
```
t1=(10,20,30,40,10,10,20)
print(sorted(t1))
```

```
[10, 10, 10, 20, 20, 30, 40]
```

Set

- The purpose of set data type is that "To store Multiple Values either Similar Type or Different Type or Both the Types in Single Object with Unique Values".
- The elements of set must be written within Curly braces { } and Elements must separate by comma.
- An Object of set never maintains Insertion Order bcoz PVM can display any one of the possibility of elements of set object.
- On the object of set, we can't perform Indexing and Slicing Operations bcoz set object never maintains Insertion Order.
- An object of set belongs to both Immutable bcoz set' object does not support item assignment and mutable in the case of add().
- Sets can be used to perform mathematical set operations like union, intersection, difference and symmetric_difference.

Empty set

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

Syntax: - setobj=set()

In [53]:
```
s1 = set()
print(s1)      # it will print set() not like only ().
type(s1)
```

```
set()
```

Out[53]: set

In [54]:
```
len(s1)
```

Out[54]: 0

Non Empty set

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

Syntax: - setobj={ val1, val2, ...val-n }

In [55]:
```
s1={10,20,30,40}
print(s1)
type(s1)
```

```
{40, 10, 20, 30}
```

Out[55]: set

In [56]:
```
len(s1)
```

Out[56]: 4

- **If you assign duplicate values / elements, it will takes unique values only.**

In [41]:
```
s2={10,20,10,20,10,30,15,12,-34}
print(s2)
```

```
{20, -34, 10, 12, 30, 15}
```

- **set doesn't support indexing and slicing, because its unorded collection of items.**

In [47]:
```
s1[1]     # indexing is not possible
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\2163967105.py in <module>
----> 1 s1[1]     # indexing is not possible

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

In [46]:
```
s1[0:4]   # slicing is not possible
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\986998996.py in <module>
----> 1 s1[0:4]   # slicing is not possible

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

- **Creating a set by using tuple list object.**

In [49]:
```
l2=[10,20,10,20,10,30,15,12,-34]
s2=set(l2)
print(l2,type(l2))
print(s2,type(s2))
```

```
[10, 20, 10, 20, 10, 30, 15, 12, -34] <class 'list'>
{10, 12, 15, 20, -34, 30} <class 'set'>
```

- **Creating a set by using tuple tuple object.**

In [57]:
```
t2=(10,20,10,20,10,30,15,12,-34)
s2=set(t2)
print(t2,type(t2))
print(s2,type(s2))
```

```
(10, 20, 10, 20, 10, 30, 15, 12, -34) <class 'tuple'>
{10, 12, 15, 20, -34, 30} <class 'set'>
```

Set Methods

In [59]:
```
c = set()
c
```

Out[59]:
```
set()
```

set.add()

- This Function is used for adding the elements to set object.
- We can add only single element at a time.
- If we try to add more than one element at a time, then we get TypeError.

Syntax: - setobj.add(Element or Value)

In [62]:
```
c.add(10)
print(c)
```

```
{10}
```

- **If we try to add more than one element at a time, then we get TypeError.**

In [66]:
```
c.add(20,30,40)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\3462233837.py in <module>
----> 1 c.add(20,30,40)

TypeError: set.add() takes exactly one argument (3 given)
```

set.update

- This Function is used for adding the multiple elements to set object.
- We can add only single element at a time.

In [67]:
```
s = {10,20}
print(s)
```

```
{10, 20}
```

In [68]:
```
s.update([30,40,50,60])
print(s)
```

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

In [70]:
```
s.update((3,4,5,6))
print(s)
```

```
{3, 4, 5, 6, 40, 10, 50, 20, 60, 30}
```

set.copy()

- This Function is used for Copying the content of one set object to another set object (Shallow Copy).

In [71]:
```
s1={10,20,30,40,50}
print(s1,type(s1))
id(s1)
```

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

Out[71]:
```
2203548488416
```

In [72]:
```
s2 = s1.copy()
print(s2,type(s2))
id(s2)
```

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

Out[72]:
```
2203548488192
```

In [75]:
```
s1.add("Siva")
print(s1,type(s1))
id(s1)
```

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

Out[75]:
```
2203548488416
```

In [76]:
```
s2.add("Ram")
print(s2,type(s2))
id(s2)
```

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

Out[76]:
```
2203548488192
```

set.discard()

- This Function is used for Removing the element from set object.
- It will accept only one value / element at a time. If we assign multiple values then we get TypeError.
- If the Value does not exist in set object then we never get KeyError(No Output--None).

Syntax: - setobj.discard(Value)

In [78]:
```
s1={10,20,30,40,50}
print(s1,type(s1))
```

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

In [79]:
```
s1.discard(40)
print(s1)
```

```
{50, 20, 10, 30}
```

In [80]:
```
s1.discard(100)
print(s1)
```

```
{50, 20, 10, 30}
```

In [83]:
```
s1.discard(10,30)
print(s1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\3320671307.py in <module>
----> 1 s1.discard(10,30)
      2 print(s1)

TypeError: set.discard() takes exactly one argument (2 given)
```

set.remove()

- This Function is used for Removing the Value from set object.
- remove() takes exactly one argument, if we assign more than one arguments / values then we get TypeError.
- If the Value does not exist in set object then we get KeyError.

Syntax: - setobj.remove(Value)

In [84]:
```
s3={10,"Rossum","Python",34.56,True}
print(s3)
```

```
{True, 34.56, 'Python', 10, 'Rossum'}
```

In [85]:
```
s3.remove(10)
print(s3)
```

```
{True, 34.56, 'Python', 'Rossum'}
```

In [88]:
```
s3.remove("Python",34.56)   # remove() takes exactly one argument, if we assign m
print(s3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\3107391513.py in <module>
----> 1 s3.remove("Python",34.56)   # remove() takes exactly one argument, if w
e assign more than one arguments then we get TypeError.
      2 print(s3)

TypeError: set.remove() takes exactly one argument (2 given)
```

In [89]:
```
s3.remove(200)    # If the Value does not exist in set object then we get KeyErro
print(s3)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\456230022.py in <module>
----> 1 s3.remove(200)    # If the Value does not exist in set object then we g
et KeyError.
      2 print(s3)

KeyError: 200
```

set.clear()

- This Function is used for removing all the elements of set object.
- **Syntax: - setobj.clear()**

In [91]:
```
s3={10,"Rossum","Python",34.56,True}
s3.clear()
print(s3)
```

```
set()
```

In [92]:
```
len(s3)
```

Out[92]:
```
0
```

set.pop()

- This Function is used for Removing any Arbitrary Element from set object.
- When we call pop() upon empty set then we get KeyError.
- **Syntax: - setobj.pop()**

In [94]:
```
s3={10,"Rossum","Python",34.56,True}
s3.pop()
print(s3)
```

```
{34.56, 'Python', 10, 'Rossum'}
```

In [95]:
```
s3.pop()
print(s3)
```

```
{'Python', 10, 'Rossum'}
```

In [96]:
```
s3.pop()
print(s3)
```

```
{10, 'Rossum'}
```

In [97]:
```
s3.pop()
print(s3)
```

```
{'Rossum'}
```

In [98]:
```
s3.pop()
print(s3)
```

```
set()
```

In [99]:
```
s3.pop()
print(s3)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\2473085721.py in <module>
----> 1 s3.pop()
      2 print(s3)

KeyError: 'pop from an empty set'
```

del operator

- This is used to delete the object and data of a given object.
- After deleting the entaire object if we try to print, then we get NameError.

In [100]:
```
s1={10,20,30,40,50}
print(s1)
```

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

In [101]:
```
del s1
print(s1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\4039378741.py in <module>
      1 del s1
----> 2 print(s1)

NameError: name 's1' is not defined
```

Set Operations

union

- This Function is used for obtaining all the Unique elements of Setobj1 and setobj2 and place them in setobj3.
- **syntax: - setobj3=setobj1.union(setobj2)**

In [105]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s1.union(s2)
print(s3,type(s3))
```

```
{35, 40, 10, 20, 25, 30} <class 'set'>
```

In [104]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s2.union(s1)
print(s3,type(s3))
```

```
{35, 40, 10, 20, 25, 30} <class 'set'>
```

intersect

- This Function is used for obtaining all common elements of Setobj1 and setobj2 and place them in setobj3.
- **syntax: - setobj3=setobj1.intersection(setobj2)**

In [106]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s1.intersection(s2)
print(s3,type(s3))
```

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

In [107]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s2.intersection(s1)
print(s3,type(s3))
```

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

difference

- This Function Removes the common elements from both Setobj1 and setobj2 and It takes remaining elements from setobj1 and place them in setobj3.
- **syntax: - setobj3=setobj1.difference(setobj2)**

  (OR)

- This Function Removes the common elements from both Setobj1 and setobj2 and It takes remaining elements from setobj2 and place them in setobj3.
- **syntax: - setobj3=setobj2.difference(setobj1)**

In [108]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s1.difference(s2)
print(s3)
```

```
{40, 10}
```

In [109]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s2.difference(s1)
print(s3)
```

```
{25, 35}
```

symmetric_difference

- This Function Removes the common elements from both Setobj1 and setobj2 and It takes remaining elements from both setobj1 and setobj2 and place them in setobj3.
- **syntax: -**

```
setobj3=setobj1.symmetric_difference()(setobj2)
```

(OR)

```
setobj3 = (s1 union s2) - (s1 intersect s2)
```

In [111]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s1.symmetric_difference(s2)
print(s3)
```

```
{35, 40, 10, 25}
```

In [112]:
```
s1={10,20,30,40}
s2={20,30,25,35}
s3=s2.symmetric_difference(s1)
print(s3)
```

```
{35, 40, 10, 25}
```

issubset()

- This Function returns True provided all the elements of setobj2 present in setobj1.
- This Function returns False provided at least one element of setobj2 not present in setobj1.
- **Syntax: - setobj2.issubset(setobj1)**

In [113]:
```
s1={10,20,30,40}
s2={10,20}
s3={10,15,25,35}
s1.issubset(s2)
```

Out[113]:
```
False
```

In [114]:
```
s1.issubset(s3)
```

Out[114]:
```
False
```

In [115]:
```
s2.issubset(s1)
```

Out[115]:
```
True
```

In [116]:
```
s2.issubset(s3)
```

Out[116]:
```
False
```

In [117]:
```
s3.issubset(s1)
```

Out[117]:
```
False
```

In [118]:
```
s3.issubset(s2)
```

Out[118]:
```
False
```

issuperset()

- This Function returns True provided setobj1 contains all the elements of setobj2.
- This Function returns False provided setobj1 does not contain at least one element of setobj2.

- **Syntax: - setobj1.issuperset(setobj2)**

In [119]:
```
s1={10,20,30,40}
s2={10,20}
s3={10,15,25,35}
s1.issuperset(s2)
```

Out[119]: True

In [120]:
```
s1.issuperset(s3)
```

Out[120]: False

In [121]:
```
s2.issuperset(s1)
```

Out[121]: False

In [122]:
```
s2.issuperset(s3)
```

Out[122]: False

In [123]:
```
s3.issuperset(s1)
```

Out[123]: False

In [124]:
```
s3.issuperset(s2)
```

Out[124]: False

In [125]:
```
s2.issuperset(s2)
```

Out[125]: True

isdisjoint()

- This Function returns True provided there is no common element in both setobj1 and setobj2.
- This Function returns False provided there is common element in both setobj1 and setobj2.

- **Syntax: - setobj1.isdisjoint(setobj2)**

In [126]:
```
s1={10,20,30,40}
s2={15,25,35}
s3={10,50,60,70}
s1.isdisjoint(s2)
```

Out[126]: True

In [127]:
```
s1.isdisjoint(s3)
```

Out[127]: False

In [128]:
```
s2.isdisjoint(s1)
```

Out[128]: True

In [129]:
```
s2.isdisjoint(s3)
```

Out[129]: True

In [130]:
```
s3.isdisjoint(s2)
```

Out[130]: True

In [131]:
```
s3.isdisjoint(s1)
```

Out[131]: False

Dictionaries or Dict

- The purpose of dict data type is that "To store (Key, value) in single variable".
- In (Key, Value), the value of Key is Unique and Value of Value may or may not be unique.
- keys can be of fundamental data type - generally the keys are int or str and values can be of any data type.
- 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.
- Dictionary is an unordered collection, it will not maintain any order.

Empty dict

- Empty dict is one, which does not contain any (Key, Value) and whose length is 0.
- **Syntax: -**

```
dictobj1= { }
  or
dictobj=dict()
```

In [132]:
```
d = {}
print(d)
type(d)
```

```
{}
```

Out[132]: dict

In [134]:
```
len(d)
```

Out[134]: 0

In [133]:
```
d1 =dict()
print(d)
type(d)
```

```
{}
```

Out[133]: dict

In [135]:
```
len(d1)
```

Out[135]: 0

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

In [136]:
```
d1={10:"Python",20:"Data Sci",30:"Django"}
print(d1,type(d1))
```

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

In [137]:
```
len(d1)
```

Out[137]: 3

In [139]:
```
d2={10:3.4,20:4.5,30:5.6,40:3.4}
print(d2,type(d2))
```

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

In [140]:
```
len(d2)
```

Out[140]: 4

In [141]:
```
d3={10:3.4,20:[45,7,8],30:(10,34,0,90),40:3.4}
print(d3,type(d3))
```

```
{10: 3.4, 20: [45, 7, 8], 30: (10, 34, 0, 90), 40: 3.4} <class 'dict'>
```

In [145]:
```
len(d3)
```

Out[145]: 4

Nested dict

- If atleast one value is dict, then it ia called as nested dict.

In [147]:
```
d1={10:"Python",20:"Data Sci",30:"Django",40:{11:"Hari"}}
print(d1,type(d1))
```

```
{10: 'Python', 20: 'Data Sci', 30: 'Django', 40: {11: 'Hari'}} <class 'dict'>
```

In [148]:
```
len(d1)
```

Out[148]:
```
4
```

Dict Methods

clear()

- This Function is used for removing all the elements of dict but object remains same (empty).
- **Syntax: - dictobj.clear()**

In [149]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4}
```

In [150]:
```
len(d1)
```

Out[150]:
```
5
```

In [151]:
```
d1.clear()
print(d1)
```

```
{}
```

In [152]:
```
len(d1)
```

Out[152]:
```
0
```

pop()

- This Function is used for removing the (Key, Value) from dict object.
- If the value of Key does not exist then we get KeyError.
- **Syntax: - dictobj.pop(Key)**

In [202]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4}
```

In [154]:
```
d1.pop(10)
print(d1)
```

```
{20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4}
```

In [155]:
```
d1.pop(20,30)
print(d1)
```

```
{30: 4.7, 40: 6.7, 50: 3.4}
```

In [156]:
```
d1.pop(1000)
print(d1)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\2705328253.py in <module>
----> 1 d1.pop(1000)
      2 print(d1)

KeyError: 1000
```

- **replace key in dictionary.**
- **Syntax: - dicobj[New_Key] = dictobj.pop(Old_key)**

In [203]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4}
```

In [205]:
```
d1[100] = d1.pop(10)     # Replacing "key=10" to "key=100".
print(d1)
```

```
{20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4, 100: 3.4}
```

popitem()

- This Function is used for removing last (Key, Value) from dict object.
- When we call this function from empty dict object, then we get KeyError.
- **Syntax: - dictobj.popitem()**

In [158]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4}
```

In [159]:
```
d1.popitem()         # 50: 3.4 deleted
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7}
```

In [160]:
```
d1.popitem()         # 40: 6.7 deleted
print(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7}
```

In [161]:
```
d1.popitem()         # 30: 4.7 deleted
print(d1)
```

```
{10: 3.4, 20: 4.5}
```

In [162]:
```
d1.popitem()         # 20: 4.5 deleted
print(d1)
```

```
{10: 3.4}
```

In [163]:
```
d1.popitem()         # 20: 4.5 deleted
print(d1)
```

```
{}
```

In [164]:
```
d1.popitem()         # When we call this function from empty dict object, then we
print(d1)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\3009285763.py in <module>
----> 1 d1.popitem()         # When we call this function from empty dict objec
t, then we get KeyError.
      2 print(d1)

KeyError: 'popitem(): dictionary is empty'
```

copy()

- This Function is used for Copying the content of One dict object into another dict object (Shallow Copy).

In [168]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
id(d1)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

Out[168]:
```
2203548557184
```

In [169]:
```
d2=d1.copy()
print(d2,type(d2))
id(d2)
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

Out[169]:
```
2203548564992
```

In [170]:
```
d1[10]=12.3
d2[50]=0.9
print(d1,type(d1),id(d1))
print(d2,type(d2),id(d2))
```

```
{10: 12.3, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'> 2203548557184
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 0.9} <class 'dict'> 2203548564992
```

get()

- This Function is used for getting the Value of Value by passing value of Key.
- **Syntax: - dictobj.get(key)**
- If the key does not exist then we get None but not KeyError.

(OR)

- **Syntax: - dictobj[key]**
- This Notation also used for obtaining Value of Value by passing value of Key.
- If the key does not exist in this Notation then we get KeyError but not None.

In [171]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [172]:
```
d1[10]
```

Out[172]: 3.4

In [173]:
```
d1.get(10)
```

Out[173]: 3.4

In [175]:
```
d1[20]
```

Out[175]: 4.5

In [176]:
```
d1.get(20)
```

Out[176]: 4.5

In [178]:
```
d1.get(1000)
print(d1.get(1000))
```

```
None
```

In [179]:
```
d1[1000]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\1669416833.py in <module>
----> 1 d1[1000]

KeyError: 1000
```

keys()

- This Function is used for obtaining Values of Key.
- **Syntax: - varname=dictobj.keys()**

In [180]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [182]:
```
ks = d1.keys()
print(ks,type(ks))
```

```
dict_keys([10, 20, 30, 40, 50]) <class 'dict_keys'>
```

In [183]:
```
d1.keys()
```

Out[183]:
```
dict_keys([10, 20, 30, 40, 50])
```

values()

- This Function is used for obtaining Values of Value.
- **Syntax: - varname=dictobj.values()**

In [184]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [185]:
```
val = d1.values()
print(val,type(val))
```

```
dict_values([3.4, 4.5, 4.7, 6.7, 3.4]) <class 'dict_values'>
```

In [186]:
```
d1.values()
```

Out[186]:
```
dict_values([3.4, 4.5, 4.7, 6.7, 3.4])
```

items()

- This Function is used for obtaining List of tuples of (Key,Value) from dict object.
- **Syntax: - varname=dictobj.items()**

In [187]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [188]:
```
kvs = d1.items()
print(kvs,type(kvs))
```

```
dict_items([(10, 3.4), (20, 4.5), (30, 4.7), (40, 6.7), (50, 3.4)]) <class 'dic
t_items'>
```

In [190]:
```
d1.items()
```

Out[190]:
```
dict_items([(10, 3.4), (20, 4.5), (30, 4.7), (40, 6.7), (50, 3.4)])
```

update()

- This Function is used for updating one dict object content with another dict object content.
- This function is also used for adding multiple items into the dictionary.
- **Syntax: - dictobj1.update(dictobj2)**

In [193]:
```
d1={10:"Python",20:"C++"}
d2={20:2.3,10:3.4}
print(d1,type(d1))
print(d2,type(d2))
```

```
{10: 'Python', 20: 'C++'} <class 'dict'>
{20: 2.3, 10: 3.4} <class 'dict'>
```

In [192]:
```
d1.update(d2)
print(d1,type(d1))
print(d2,type(d2))
```

```
{10: 3.4, 20: 2.3} <class 'dict'>
{20: 2.3, 10: 3.4} <class 'dict'>
```

In [194]:
```
d2.update(d1)
print(d1,type(d1))
print(d2,type(d2))
```

```
{10: 'Python', 20: 'C++'} <class 'dict'>
{20: 'C++', 10: 'Python'} <class 'dict'>
```

- **adding a multiple items to dictionary by using update()**

In [199]:
```
d1.update({50: "Java",60: "AWS"})
print(d1,type(d1))
```

```
{10: 47, 20: 'C++', 50: 'Java', 60: 'AWS'} <class 'dict'>
```

- **adding a single item to dictionary.**

In [200]:
```
d1[70] = 47
print(d1,type(d1))
```

```
{10: 47, 20: 'C++', 50: 'Java', 60: 'AWS', 70: 47} <class 'dict'>
```

- **replace value in dictionary.**

In [206]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [208]:
```
d1[20] = "Hari"       # Replacing "value=4.5" to "value="Hari"".
print(d1)
```

```
{10: 3.4, 20: 'Hari', 30: 4.7, 40: 6.7, 50: 3.4}
```

del operator

- This is used for removing the object along with data.

In [209]:
```
d1={10:3.4,20:4.5,30:4.7,40:6.7,50:3.4}
print(d1,type(d1))
```

```
{10: 3.4, 20: 4.5, 30: 4.7, 40: 6.7, 50: 3.4} <class 'dict'>
```

In [212]:
```
del d1          # delting d1 object.
print(d1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5388\660573689.py in <module>
----> 1 del d1          # delting d1 object.
      2 print(d1)

NameError: name 'd1' is not defined
```

Range

- The purpose of range data type is that " To Store Sequence of Numerical Integer Values maintaining equal Interval of value".

- An object of range data type belongs to immutable bcoz range object does not support Item Assignment.
- To convert one type of value to range type of value, we use range() rangeobject=range(object).
- range data type contains 3 syntaxes for generating range of values with equal interval.
- On the object of range, we can perform Indexing and Slicing Operations.

- **Syntax-1 : - rangeobj=range(Value)**

- This Syntax generates range of values from 0 to Value-1

(OR)

- **Syntax-2 : - rangeobj=range(Begin,End)**

- This Syntax generates range of values from BEGIN to END-1. If we not specified Begin value then it will takes as default "0".

(OR)

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

- This Syntax generates range of values from BEGIN to END with Equal STEP as Interval.

In [213]:
```
range(10)           # It will returns (0 to 9)
```

Out[213]:
```
range(0, 10)
```

In [214]:
```
print(range(10))
type(range(10))
```

```
range(0, 10)
```

Out[214]:
```
range
```

In [215]:
```
list(range(10))    # It will returns list[0 to 9]
```

Out[215]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [217]:
```
type(list(range(10)))
```

Out[217]:
```
list
```

In [216]:
```
tuple(range(10))   # It will returns tuple(0 to 9)
```

Out[216]:
```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

In [218]:
```
type(tuple(range(10)))
```

Out[218]:
```
tuple
```

In [219]:
```
list(range(5,10))
```

Out[219]:
```
[5, 6, 7, 8, 9]
```

In [222]:
```
list(range(5,21,2))
```

Out[222]:
```
[5, 7, 9, 11, 13, 15, 17, 19]
```

In [223]:
```
list(range(5,51,5))
```

Out[223]:
```
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

In [224]:
```
tuple(range(10,21))
```

Out[224]:
```
(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
```

In [225]:
```
tuple(range(10,21,2))
```

Out[225]:
```
(10, 12, 14, 16, 18, 20)
```

In [226]:
```
tuple(range(10,41,4))
```

Out[226]:
```
(10, 14, 18, 22, 26, 30, 34, 38)
```

In [232]:
```
r = range(6)
print(r,type(r))
```

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

In [233]:
```
for val in r:
    print(val)
```

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

In [234]:
```
r = range(3,8)
print(r,type(r))
```

```
range(3, 8) <class 'range'>
```

In [235]:
```
for val in r:
    print(val)
```

```
3
4
5
6
7
```

In [236]:
```
r = range(2,11,2)
print(r,type(r))
```

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

In [237]:
```
for val in r:
    print(val)
```

```
2
4
6
8
10
```

In [242]:
```
r = range(100,0,-10)
print(r,type(r))
```

```
range(100, 0, -10) <class 'range'>
```

In [243]:
```
for val in r:
    print(val)
```

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

In [250]:
```
r = range(-10,5,3)
print(r,type(r))
```

```
range(-10, 5, 3) <class 'range'>
```

In [251]:
```
for val in r:
    print(val)
```

```
-10
-7
-4
-1
2
```

In [252]:
```
r = range(-1,-11,-1)
print(r,type(r))
```

```
range(-1, -11, -1) <class 'range'>
```

In [253]:
```
for val in r:
    print(val)
```

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