List

List is one of the important built in data type in python. List literals are written within square brackets [ ] where elements are separated by comma.

Characteristics of a Python List

The various characteristics of a list are:

- Ordered: Lists maintain the order in which the data is inserted.
- Mutable: In list element(s) are changeable. It means that we can modify the items stored within the list.
- Heterogenous: Lists can store elements of various data types.
- Dynamic: List can expand or shrink automatically to accommodate the items accordingly.
- Duplicate Elements: Lists allow us to store duplicate data.

In [1]:
```
# We can use square bracket without any element to create the empty list.
l1 = []
print(l1)
```

```
[]
```

In [2]:
```
# Enter the elements and separate by comma inside the square bracket to create the
l1=[1,2,3,4,5,6]         #Integer list
print(l1)
```

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

In [3]:
```
l1=[1.5, 2.5, 6.2, 8.2, 15.6]         #Float list
print(l1)
```

```
[1.5, 2.5, 6.2, 8.2, 15.6]
```

In [4]:
```
l1=['A', "B", "C", 'D']         #String list
print(l1)
```

```
['A', 'B', 'C', 'D']
```

In [5]:
```
l1=[10, 35, 45.62, 78.12, "Hello", 'Abhi']         #Heteregeneous datatype list
print(l1)
```

```
[10, 35, 45.62, 78.12, 'Hello', 'Abhi']
```

In [6]:
```
l1=[10, 35, 20, 65, 35, 10, 10]         #Duplicate items in the list
print(l1)
```

```
[10, 35, 20, 65, 35, 10, 10]
```

In [8]:
```
l1=[10, 35, 45.62, 78.12, ["Hello", 'Abhi'], 12]         #Nested list
print(l1)
```

```
[10, 35, 45.62, 78.12, ['Hello', 'Abhi'], 12]
```

In [2]:
```
l=list((1,2,3,4,5))    # The list() function creates a list object.
print(l)
```

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

In [3]:
```
l1=[10, 35, 20, 65, "Abhi", 10, "Kevin"]*2
print(l1)
# Here we are multiply the existed list by 2. The elements repeats two times in the
```

```
[10, 35, 20, 65, 'Abhi', 10, 'Kevin', 10, 35, 20, 65, 'Abhi', 10, 'Kevin']
```

In [4]:
```
l1=[10, 35, 20, 65, 10]
l2=["Abhi", "Ben", "kevin"]
print(l1+l2)
# Here we are adding another list to the existed list.
```

```
[10, 35, 20, 65, 10, 'Abhi', 'Ben', 'kevin']
```

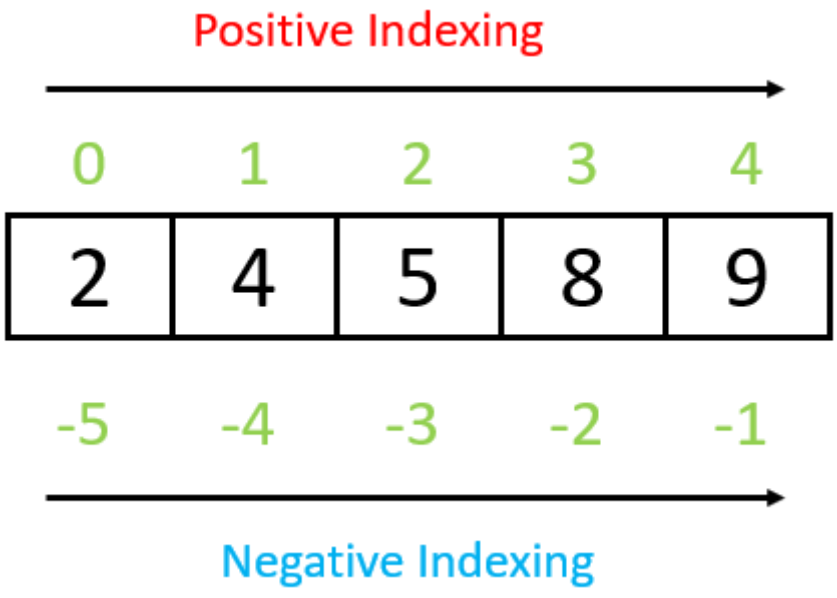
Accessing items from the list.

1. Indexing

Since list is ordered which means that every element in the list has certain position in the list. The position of any elemnt in the list is called the index of the element. With the help of index, we can easily find a particular element from the list. In list indexing is start from 0 and then 1, 2, and so on. Index 0 means the first element of the element, index 1 means the second element of the list and so on. We use [ ] (square brackets) to access the elements from the list using indexing. We pass index inside the square bracket to get the element from the list.

There are two type of indexing in the list.

- Positive Indexing: It always start from 0, 1, 2, and so on from left to right (--->).
- Negative Indexing: always start from -1, -3, -3, and so on from right to left (<---).



Accessing elements from the list using Positive Indexing

In [3]:
```
l=[2, 4, 5, 8, 9]
print(f"The element at 0th index is {l[0]}.")    # ---> Here we are using 0th index
print(f"The element at 1st index is {l[1]}.")    # ---> Here we are using 1st index
print(f"The element at 2nd index is {l[2]}.")    # ---> Here we are using 2nd index
```

```
print(f"The element at 3rd index is {l[3]}.")    # ---> Here we are using 3rd index
print(f"The element at 4th index is {l[4]}.")    # ---> Here we are using 4th index
```

```
The element at 0th index is 2.
The element at 0th index is 4.
The element at 0th index is 5.
The element at 0th index is 8.
The element at 0th index is 9.
```

Note: Positive indexing start from 0 not 1 so take care of this while using positive indexing.

In [4]:
```
l = ["Hello", "Namaste", "Sayo Nara", "Vanakkam"]
print(l[2])
# Here we want to access "Sayo Nara" from the list, so starting from 0, the index o
#  when we counting the position of element "Sayo Nara" in the list, it comes out i
# positive indexing, the index is always 1 less than the position of the element in
```

```
Sayo Nara
```

Consider the nested list.

m = [2, 4, 5, ["Hello", "Namaste", "Sayo Nara", "Vanakkam"], 8, 9]

Now we want to access the element "Namaste" from the nested list using positive indexing. Since the element "Namaste" is inside the list which is also inside another list m. The list " ["Hello", "Namaste", "Sayo Nara", "Vanakkam"]" is considered as one element of the main list m. This means the list m have only 6 elements where the list "["Hello", "Namaste", "Sayo Nara", "Vanakkam"]" is at 4th position or at 3rd index of the main list m.

Now we have the index of the inside list which is 3, we can access this inside list using m[3], this will give the complete inside list, "["Hello", "Namaste", "Sayo Nara", "Vanakkam"]". Now again we have to find a particular element from the inside list say "Namaste" which is at 2nd position but at index of 1. So, m[3][1] will give the element "Namaste".

In [5]:
```
m = [2, 4, 5, ["Hello", "Namaste", "Sayo Nara", "Vanakkam"], 8, 9]
print(m[3][1])
```

```
Namaste
```

Accessing elements from the list using Negative Indexing

In [7]:
```
l=[2, 4, 5, 8, 9]
print(f"The element at -1 index is {l[-1]}.")    # ---> Here we are using -1 index
print(f"The element at -2 index is {l[-2]}.")    # ---> Here we are using -2 index
print(f"The element at -3 index is {l[-3]}.")    # ---> Here we are using -3 index
print(f"The element at -4 index is {l[-4]}.")    # ---> Here we are using -4 index
print(f"The element at -5 index is {l[-5]}.")    # ---> Here we are using -5 index
```

```
The element at -1 index is 9.
The element at -2 index is 8.
The element at -3 index is 5.
The element at -4 index is 4.
The element at -5 index is 2.
```

Note: Negative indexing start from right to left whereas positive indexing start from left to right.

In [9]:
```
m = [2, 4, 5, ["Hello", "Namaste", "Sayo Nara", "Vanakkam"], 8, 9]
print(m[-3][-3])
```

```
# Here we are using negative indexing to access the element from the nested list.
```

```
Namaste
```

In [13]:
```
print(f"Using positive indexing: {m[1]}")
print(f"Using negative indexing: {m[-5]}")
```

```
Using positive indexing: 4
Using negative indexing: 4
```

2. Slicing

Slicing is another technique like indexing to access elements from the list. It is basically used to access the range of element from the list. Using slicing, we can access one element as well as more than one element from the list. We again use square brackets [ ] for slicing but along with square bracker we alse use : (colon) to give the range of index for the particular range of elements.

Syntax: list[Start_index : Stop_index : Step]

By default, step is 1 until any other value given to it. if we ignore the step, then we don't need to use the second colon, we can avoid it.

Syntax -->list[Start_index : Stop_index] (by default step is 1.)

Start_index: This basically tells from which index we have to fetch the elements.

Stop_index: This basically tells upto which index we have to fetch the elements. Note: Stop index is exclusive.

Step: This basically tells to go the each mentioned step element in the list after the fetched element. For example; step=1 means go the immediate next element, step=2 means go the each second element from the fetched elements.

Note: Slicing gives the element in the list form irrespective of the number of elements.

Positive Slicing with positive step

Here we used positive index and positive step to slice out the list.

Note: The starting index must be **smaller than** the stop index else it will display blank or empty element.

In [18]:
```
l=[2, 4, 5, 8, 9]
print(l[1:4:1])
# Here, staring index is 1 and at this index, the element is 4. Stopping index is 4
# since the stopping index is exclusive so the last value will be 8 instead of 9 th
# The step size is 1 which means fetch all element consecutive without skipping ele
```

```
[4, 5, 8]
```

In [20]:
```
l=[2, 4, 5, 8, 9]
print(l[1:4:2])
# The step size is 2 which means fetch all the second elements.
```

```
[4, 8]
```

In [21]:
```
l=[2, 4, 5, 8, 9]
print(l[1:4])
# Here we didn't mentioned the step size, so it takes default value which is 1.
# The step size is 2 which means fetch all the second elements.
```

```
[4, 5, 8]
```

In [33]:
```
l=[2, 4, 5, 8, 9]
print(l[:3])
# Here the starting index is not given, so the starting index become 0 and the stop
# the element from the 0 index to the 2nd index as 3rd index is exclusive.
```

```
[2, 4, 5]
```

In [34]:
```
l=[2, 4, 5, 8, 9]
print(l[2:])
# Here the starting index is 2 and the stop index is not given that is why it displ
# index to the last element of the list.
```

```
[5, 8, 9]
```

In [22]:
```
l=[2, 4, 5, 8, 9]
print(l[:])
# Here we neither mentioned stating index, stop index or step size, so it display c
```

```
[2, 4, 5, 8, 9]
```

Note: If we didn't mentioned the starting index, then by default the starting index become 0. In the same way, if we didn't mentioned the stopping index, then by default the stop index become the last index of the list.

In [23]:
```
print(l[5:2])
```

```
[]
```

Note: Also, if you observe that in positive slicing, the starting index is always less than or equal to stop index. if the starting index is greater than the stop index, then a blank or empty element is displayed.

Negative Slicing with positive step

Here, we used negative index and positive step to slice out the list.

Note: The starting index must be **smaller than** the stop index else it will display blank or empty element.

In [31]:
```
l=[2, 4, 5, 8, 9]
print(l[-5:-1:1])
# Here, staring index is -5 and at this index, the element is 2. Stopping index is
# since the stopping index is exclusive so the last value will be 8 instead of 9 th
# -1. The step size is 1 which means fetch all element consecutive without skipping
```

```
[2, 4, 5, 8]
```

In [32]:
```
l=[2, 4, 5, 8, 9]
print(l[-1:-5:1])
```

```
[]
```

Note: when we want to fetch the elements in the same order as the list, then remember staring index must be smaller than the stopping index (-5 < -1),else blank or empty element

will displayed.

In [35]:
```
l=[2, 4, 5, 8, 9]
print(l[-4:])
# Here, the stop index is not given that is why it displayed all the elements from
```

```
[4, 5, 8, 9]
```

In [38]:
```
l=[2, 4, 5, 8, 9]
print(l[:-2])
# Here, the start index is not given that is why it displayed all the elements fro
# exclusive.
```

```
[2, 4, 5]
```

Positive Slicing with negative step

Here, we used positive index and negative step to slice out the list.

Note: The starting index must be **greater than** the stop index else it will display blank or empty element. It display the list of element in reverse order.

In [51]:
```
l=[2, 4, 5, 8, 9]
print(l[5:2:-1])
# Here, the start index is greater than the stop index. Also, the list of elements
```

```
[9, 8]
```

In [52]:
```
l=[2, 4, 5, 8, 9]
print(l[:1:-2])
# Here, the step is negative that is why the fetching of element takes from right
# is not given that is why starting index become 0. The stopping index is 1 and at
# this element and display element prior to this element that is upto 5.
# Also, the step is -2 so it display each second element from the starting index v
```

```
[9, 5]
```

In [56]:
```
l=[2, 4, 5, 8, 9]
print(l[4::-3])
# Here, the step is negative that is why the fetching of element takes from right
# is 4, so it display all element from the index 4 upto the one element before the
# Also, the step is -3 so it display each third element from the starting index va
```

```
[9, 4]
```

In [57]:
```
l=[2, 4, 5, 8, 9]
print(l[::-2])
# Here, the step is -2 so it display each second element from the starting negative
```

```
[9, 5, 2]
```

Negative Slicing with negative step

Here, we used negative index and negative step to slice out the list.

Note: The starting index must be **greater than** the stop index else it will display blank or empty element. It display the list of element in reverse order.

In [46]:
```
l=[2, 4, 5, 8, 9]
print(l[-1:-4:-1])
# Here, the start index is greater than the stop index. Also, the list of elements
```

```
[9, 8, 5]
```

In [47]:
```
l=[2, 4, 5, 8, 9]
print(l[:-4:-2])
# Here, the step is negative that is why the fetching of element takes from right
# is not given that is why starting index become -1. The stopping index is -4 and
# this element and display element prior to this element that is upto 5.
# Also, the step is -2 so it display each second element from the starting index va
```

```
[9, 5]
```

In [48]:
```
l=[2, 4, 5, 8, 9]
print(l[-2::-3])
# Here, the step is negative that is why the fetching of element takes from right
# is -2, so it display all element from the index -2 upto the one element before th
# Also, the step is -3 so it display each third element from the starting index val
```

```
[8, 2]
```

In [50]:
```
l=[2, 4, 5, 8, 9]
print(l[::-2])
# Here, the step is -2 so it display each second element from the starting negative
```

```
[9, 5, 2]
```

Updating elements in the list

Since list is mutable which means the elements in the list can be change.

In [58]:
```
l=[2,5,4,6,7,8]
print(f"Before changes : {l}")
l[2]=1     #----> Here, we are changing the element at index of 2 from 4 to 1.
print(f"After changes : {l}")
```

```
Before changes : [2, 5, 4, 6, 7, 8]
After changes : [2, 5, 1, 6, 7, 8]
```

In [59]:
```
l=[2,5,4,6,7,8]
print(f"Before changes : {l}")
l[2:5]=[10,20,30]     #----> Here, we are changing the list of elements using slic
print(f"After changes : {l}")
```

```
Before changes : [2, 5, 4, 6, 7, 8]
After changes : [2, 5, 10, 20, 30, 8]
```

In [61]:
```
l=[2,5,4,6,7,8]
print(f"Before changes : {l}")
l[-2]=15     #----> Here, we are changing the element at index of -2 from 7 to 15.
print(f"After changes : {l}")
```

```
Before changes : [2, 5, 4, 6, 7, 8]
After changes : [2, 5, 4, 6, 15, 8]
```

List methods.

Since list is dynamic in nature which means we can expand or shrink the list that is we can add or remove the elements from the list.

1. append() method

This method append or add only single element at the end of the list. This is one of most used method of list.

Syntax: List.append(item)

This method takes one argument which can be string, number, list, set, tuple, etc.

In [74]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
l.append("SQL")     #---> The element 'SQL' added to the last of the list
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'Java', 'HTML', 'JavaScript', 'SQL']
```

In [75]:
```
l2=["MYSQL", "Oracle", "MongoDB", "Casandra"]
l.append(l2) #---> Here, we are adding another list. Note list l2 considered as one
print(l)
```

```
['Python', 'Java', 'HTML', 'JavaScript', 'SQL', ['MYSQL', 'Oracle', 'MongoDB', 'Ca
sandra']]
```

2. extend() method

This method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.

Syntax: List.extend(iterable)

This method takes an iterable such as list, tuple, string etc.

In [76]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
l.extend(["MYSQL", "Oracle", "MongoDB", "Casandra"])   #--> Here, all the element
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'Java', 'HTML', 'JavaScript', 'MYSQL', 'Oracl
e', 'MongoDB', 'Casandra']
```

Note: In append method, only a single element is added to the list while in extend method, all the elements of another list, set, tuple added to the first list.

That is, in append method, the length of list is increas by only 1 whereas in extend method, the length of the list can be increase by 1 or more than 1.

3. insert() method

This method inserts an element to the list at the specified index.

Syntax: List.insert(index, element)

The insert() method takes two parameters:

- index - the index where the element needs to be inserted
- element - this is the element to be inserted in the list

In [77]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
```

```
l.insert(2, "Scala")   #--> Here, the element "Scala" is added at 2nd index.
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'Java', 'Scala', 'HTML', 'JavaScript']
```

4. remove() method

The remove() method removes the first matching element (which is passed as an argument) from the list.

Syntax: List.remove(element)

The remove() method takes a single element as an argument and removes it from the list. If the element doesn't exist, it throws ValueError: list.remove(x): x not in list exception.

In [78]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
l.remove("Java")   #--> Here, the element "Java" is removed from the list.
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'HTML', 'JavaScript']
```

Note: If a list contains duplicate elements, the remove() method only removes the first matching element.

In [79]:
```
l.remove("Scala")   #--> Here, the element "Scala" is not in the list. That is why
print(f"{l}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[79], line 1
----> 1 l.remove("Scala")   #--> Here, the element "Scala" is not in the list. Tha
t is why it throws an error.
      2 print(f"{l}")

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

5. pop() method

The pop() method removes the item at the given index from the list and returns the removed item.

Syntax: List.pop(index)

The argument passed to the method is optional. If not passed, the default index -1 is passed as an argument (index of the last item).

In [80]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
l.pop(2)   #--> Here, the element "HTML"  which is at index 2 is removed from the
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'Java', 'JavaScript']
```

In [81]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using the method: {l}")
```

```
l.pop()   #--> Here, the last element "Javascript" is removed from the list as no
print(f"After using the method: {l}")
```

```
Before using the method: ['Python', 'Java', 'HTML', 'JavaScript']
After using the method: ['Python', 'Java', 'HTML']
```

In [82]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(l.pop())     #---> Here, the element "Javascript is removed from the list."
```

```
JavaScript
```

Note: The pop() method returns the item present at the given index. This item is also removed from the list.

Note: When we have to remove a particular element, then we will use remove() method but when we have to remove an element at a particular index or using the index, then we will use pop() method.

6. clear() method

The clear() method removes all items from the list.

Syntax: List.clear()

This method doesn't take any argument.

In [85]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(f"Before using method: {l}")
l.clear()
print(f"After using method: {l}")     #---> Here, all the elements are removed fro
```

```
Before using method: ['Python', 'Java', 'HTML', 'JavaScript']
After using method: []
```

7. index() method

The index() method gives the index of a particular element from the list.

Syntax: List.index(element, start, end)

The index() method can take a maximum of three arguments:

- element - the element to be searched.
- start (optional) - start searching from this index
- end (optional) - search the element up to this index

The index() method returns the index of the given element in the list. If the element is not found, a ValueError exception is raised.

In [87]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(l.index("HTML"))     #---> Here, the index of element "HTML" is returned.
```

```
2
```

In [88]:
```
l=["Python", "Java", "HTML", "JavaScript"]
print(l.index("Java"))     #---> Here, the index of element "HTML" is returned.
```

```
1
```

Note: The index() method only returns the first occurrence of the matching element.

In [90]:
```
l = [ 2, 5, 6, 5, 7, 5, 8, 9]
l.index(5, 3, 8)
# Here 5 is the element that we need to look for, 3 is the start index and 8 is the
# 5 within the range of index 3 to 8.
```

Out[90]:
```
3
```

8. count() method

The count() method returns the number of times the specified element appears in the list.

Syntax: List.count(element)

The count() method can take one argument and return the number of times that element appear in the list.

In [91]:
```
l = [ 2, 5, 6, 5, 7, 5, 8, 9]
l.count(5)
# Here 5 is the element that we need to look for. 5 comes three times in the list.
```

Out[91]:
```
3
```

9. reverse() method

The reverse() method reverses the elements of the list.

Syntax: List.reverse()

The reverse() method doesn't take any arguments. The reverse() method doesn't return any value. It updates the existing list.

In [94]:
```
l = [ 2, 5, 6, 5, 7, 5, 8, 9]
l.reverse()
print(l)
```

```
[9, 8, 5, 7, 5, 6, 5, 2]
```

In [95]:
```
l = ["Keyboard", "Mouse", "Laptop"]
l.reverse()
print(l)
```

```
['Laptop', 'Mouse', 'Keyboard']
```

10. sort() method

The sort() method sorts the items of a list in ascending or descending order.

Syntax: list.sort(key=..., reverse=...)

By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

- reverse - If True, the sorted list is reversed (or sorted in Descending order)
- key - function that serves as a key for the sort comparison

Note: The sort() method doesn't return any value. Rather, it changes the original list.

In [96]:
```
l = [ 2, 5, 6, 5, 7, 5, 8, 9]
l.sort()
print(l)
# By default, it sort the list in ascending order.
```

```
[2, 5, 5, 5, 6, 7, 8, 9]
```

In [98]:
```
l = [ 2, 5, 6, 5, 7, 5, 8, 9]
l.sort(reverse=True)
print(l)
# when reverse set to True, it sort the list in descending order.
```

```
[9, 8, 7, 6, 5, 5, 5, 2]
```

In [99]:
```
l = ["Keyboard", "Mouse", "Laptop"]
l.sort()
print(l)
```

```
['Keyboard', 'Laptop', 'Mouse']
```

In [101…
```
l = ["Keyboard", "Mouse", "Laptop"]
l.sort(key=len)     #---> Here we use len function for key. Len function gives the
print(l)       #---> Here sorting is done according to the length of the element i
```

```
['Mouse', 'Laptop', 'Keyboard']
```

11. copy() method

The copy() method returns a shallow copy of the list.

Syntax: list.copy()

The copy() method doesn't take any parameters. The copy() method returns a new list. It doesn't modify the original list.

In [102…
```
l = ["Keyboard", "Mouse", "Laptop"]
l2=l.copy()
print(l)
print(l2)
```

```
['Keyboard', 'Mouse', 'Laptop']
['Keyboard', 'Mouse', 'Laptop']
```

In [103…
```
l.append("Monitor")
print(l)
print(l2)
```

```
['Keyboard', 'Mouse', 'Laptop', 'Monitor']
['Keyboard', 'Mouse', 'Laptop']
```

Note: If we make any changes in the original list then there will no changes take place in the copy list.

Usage of some common built in python function with List

len()

len() function gives the length of the object.

In [1]:
```
l=[10,20,30,40,50,60]
print(len(l))
# Here the list l have 6 elements that is why the length of the list l is 6.
```

```
6
```

In [2]:
```
l=[10,20,30,["Hello", 'Namaste', "Good Morning"],50,60]
print(len(l))
# Here, the given list is a nested list, if you count, the list l have still 6 elem
# 6. Although the inner list have 3 elements but this whole list considered as one
```

```
6
```

max()

max() function return the greater element in the list compare to the other elements.

In [3]:
```
l=[10,20,50,40,30,15]
print(max(l))
# Here the largest element in the list is 50.
```

```
50
```

In [22]:
```
l=["Hello", "Namaste", "Good Morning", "abhi"]
print(max(l))
# Here the max() function works on the based of ASCII value. since the first charac
# which is greater than the all ASCII value of each element first character.
```

```
abhi
```

In [24]:
```
l=[10,20,30,"Bye",["Hello", 'Namaste', "Good Morning"],50,60]
print(max(l))
# max() function works on homogeneous datatype instead of heteregeneous datatype.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[24], line 2
      1 l=[10,20,30,"Bye",["Hello", 'Namaste', "Good Morning"],50,60]
----> 2 print(max(l))

TypeError: '>' not supported between instances of 'str' and 'int'
```

min()

min() function return the smallest element in the list compare to the other elements.

In [25]:
```
l=[10,20,50,40,30,15]
print(min(l))
# Here the minimum element in the list is 10.
```

```
10
```

In [26]:
```
l=["Hello", "Namaste", "Good Morning", "abhi"]
print(min(l))
# min() function also works on the based of ASCII value.
```

```
Good Morning
```

Note: min() function also works on homogeneous datatype.

sum()

sum() function return the sum of all the element in the list.

In [28]:
```
l=[10,20,50,40,30,15]
print(sum(l))
# Here this function gives the sum of all the elements. 10+20+50+40+30+15=165
```

```
165
```

Note: sum() function works on integer or float datatype elements.

Click this link to learn more: 👉 https://github.com/Abhishekk-Git