LIST

```
In [ ]: #List is a datastructure
        #list is an ordered collection of items enclosed in square brackets[] and sepera
        #list is a mutable datatype where we can change,add or remove elements even afte
        #list allows duplicate values in to list
        #list is dynamic datatype, it can grow or shrink(increase or decrease)
        #list allows multiple datatypes(heterogeneous elements) in to list
        #list is ordered
        #list creation follows
```

CREATING A LIST

```
In [11]: Empty = []#empty list
```

```
In [12]: Empty
```

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

```
In [13]: empty = list()
```

```
In [14]: empty
```

```
Out[14]: []
```

```
In [15]: l1 = [1,2,3,4,5] # List of integers
         l1
```

```
Out[15]: [1, 2, 3, 4, 5]
```

```
In [16]: l2 = [1.2,2.1,3.2,4.3]# List of float numbers
         l2
```

```
Out[16]: [1.2, 2.1, 3.2, 4.3]
```

```
In [17]: l3 = ['hi','welcom','to','Python','world']# List of strings
         l3
```

```
Out[17]: ['hi', 'welcom', 'to', 'Python', 'world']
```

```
In [18]: l4 = [1,True,0,False] # List of boolean values
         l4
```

```
Out[18]: [1, True, 0, False]
```

```
In [19]: l5 = [1+2j,1-2j,2+1j,2-1j]#List of complex values
         l5
```

```
Out[19]: [(1+2j), (1-2j), (2+1j), (2-1j)]
```

In [31]:
```
# List items are heterogeneous
# List can contain multiple types in single list
hetero = [12,1.2,'Hello',1+2j,True,(1,2,3),{'key':'value'},{'apple','guava'}]
hetero
```

Out[31]:
```
[12,
 1.2,
 'Hello',
 (1+2j),
 True,
 (1, 2, 3),
 {'key': 'value'},
 {'apple', 'guava'}]
```

In [32]:
```
# type() is an inbuilt function used to know which type of data is stored in var
type(hetero)
```

Out[32]:
```
list
```

In [33]:
```
type(hetero[2])# type of a particular value inside a list
```

Out[33]:
```
str
```

In [34]:
```
h = hetero
```

In [35]:
```
h
```

Out[35]:
```
[12,
 1.2,
 'Hello',
 (1+2j),
 True,
 (1, 2, 3),
 {'key': 'value'},
 {'apple', 'guava'}]
```

In [36]:
```
print(type(h[0]))
print(type(h[1]))
print(type(h[2]))
print(type(h[3]))
print(type(h[4]))
print(type(h[5]))
print(type(h[6]))
print(type(h[7]))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'complex'>
<class 'bool'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
```

In [39]:
```
# Nested list....>A List inside other list is called nested list

h1 = [1,2,3,[4,2.1,'hi']]
```

```
In [40]: print(h1)
         print(type(h1))
```

```
[1, 2, 3, [4, 2.1, 'hi']]
<class 'list'>
```

INDEXING AND SLICING

List Elements can be accessed using Indexing and Slicing Indexing : Indexing can be done in three ways ---->Forward Indexing ---->Backward Indexing ---->Step Indexing Forward Indexing : Forward Indexing is also called Zero based indexing, where index starts from 0. Backward Indexing : Backward Indexing is also called Negative Indexing, where index stars from -1. Step Indexing : Step Indexing is a Slicing Technique that uses a value to skip a specific number of elements when extracting a sub list.

INDEXING

```
In [42]: h
```

```
Out[42]: [12,
          1.2,
          'Hello',
          (1+2j),
          True,
          (1, 2, 3),
          {'key': 'value'},
          {'apple', 'guava'}]
```

```
In [43]: # Forward Indexing
         h[0]# returns value at 0 index
```

```
Out[43]: 12
```

```
In [45]: h[4]#returns value at 4 index
```

```
Out[45]: True
```

```
In [46]: h[0,4]# only single argument
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[46], line 1
----> 1 h[0,4]

TypeError: list indices must be integers or slices, not tuple
```

```
In [48]: h[5][0]#returns value at 0 index from 5th index value
```

```
Out[48]: 1
```

```
In [49]: h[5][2]
```

```
Out[49]: 3
```

```
In [50]: # Backward Indexing
         h[-1]
```

```
Out[50]: {'apple', 'guava'}
```

```
In [51]: h[-4]
```

```
Out[51]: True
```

```
In [55]: h[-6]
```

```
Out[55]: 'Hello'
```

```
In [56]: h[-8]
```

```
Out[56]: 12
```

SLICING

# while indexing is used to access value at particular index # Slicing is used access multiple item between start and stop values #[start:stop:step] #start is where the value starts....it is optional as default value is 0 #stop is where the value ends (it wont print stop value as it follow (n-1))....it is must #step is value to skip no.of elements....>it is also optional as default value is 1

```
In [57]: lst = [1,2,3,6,4,7,8,3,5,2,3,8,9,6,2]
         lst
```

```
Out[57]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [61]: a = lst
         a
```

```
Out[61]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [62]: a[:] # single colon prints complete list
```

```
Out[62]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [63]: a[::]# it is also similar
```

```
Out[63]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [64]: a[::-1] # it is starndard to print a list in reverse order
```

```
Out[64]: [2, 6, 9, 8, 3, 2, 5, 3, 8, 7, 4, 6, 3, 2, 1]
```

```
In [65]: a[0:]# prints values from index 0 to last
```

```
Out[65]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [66]: a[3:]# prints values from index 3 to last
```

```
Out[66]: [6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6, 2]
```

```
In [67]: a[2:6] # prints values from 2 to 5th index as we saw above it skips stop index(n
```

```
Out[67]: [3, 6, 4, 7]
```

```
In [68]: a[:8] # from 0 to 7th index
```

```
Out[68]: [1, 2, 3, 6, 4, 7, 8, 3]

In [69]: a[:0]

Out[69]: []

In [70]: a[-1:] # prints from -1 to last....as it is only last it prints only that value

Out[70]: [2]

In [73]: a[-3:]# prints from-3 to -1

Out[73]: [9, 6, 2]

In [74]: a[:-1]# skips value at -1 index

Out[74]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3, 8, 9, 6]

In [75]: a[:-4]

Out[75]: [1, 2, 3, 6, 4, 7, 8, 3, 5, 2, 3]

In [76]: a[4:8]

Out[76]: [4, 7, 8, 3]

In [78]: a[2:8:2]#skips step value it means skip every second value

Out[78]: [3, 4, 8]

In [80]: a[2:9:1]# 2 to 8...>1 not skips anything

Out[80]: [3, 6, 4, 7, 8, 3, 5]

In [81]: a[0:13:3]

Out[81]: [1, 6, 8, 2, 9]

In [82]: a[0:-1:2]

Out[82]: [1, 3, 4, 8, 5, 3, 9]

In [84]: a[-1:6]

Out[84]: []

In [86]: a[6:-1]

Out[86]: [8, 3, 5, 2, 3, 8, 9, 6]

In [87]: a[2:-4]

Out[87]: [3, 6, 4, 7, 8, 3, 5, 2, 3]

In [88]: a[-6:-4]
```

Out[88]:

```
[2, 3]
```

In [91]:

```
a[2:8:-2]# it returns empty list
# because in slicing if the step is negative, the slice must move from higher to
#2 slice begins
#8 slice ends
#-2 tells com to move backward

# index 2 is already left of index 8, you cant reach 8 by moving backward
#since starting point can never meet the stoping point using step, it results em
```

Out[91]:

```
[]
```

In [92]:

```
# we can work by swaping start and stop
a[8:2:-2]
```

Out[92]:

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

In [93]:

```
a[3:13:-2] # in this type cases start should be greater than stop
```

Out[93]:

```
[]
```

In [94]:

```
a[13:3:-2]
```

Out[94]:

```
[6, 8, 2, 3, 7]
```

In [95]:

```
a[:]
```

Out[95]:

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

In [89]:

```
a
```

Out[89]:

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

METHODS IN LIST

ADDING ITEMS TO LIST

In [ ]:

```
# to add items we use following methods in list
#>>append()
#>>extend()
#>>insert()
```

append()

In [96]:

```
n = [1,2,3,'hi','hello']
n
```

Out[96]:

```
[1, 2, 3, 'hi', 'hello']
```

In [104…

```
# append is method which adds element at end of list
```

```
n.append(4)
n
```

Out[104…

```
[1, 2, 3, 'hi', 'hello', 4]
```

In [105…

```
n.append(5,6) #append only accepts single argument
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[105], line 1
----> 1 n.append(5,6)
      2 n

TypeError: list.append() takes exactly one argument (2 given)
```

In [106…

```
n.append([5,6,7,8])# append treats list as a single object and appends at end
n
```

Out[106…

```
[1, 2, 3, 'hi', 'hello', 4, [5, 6, 7, 8]]
```

extend()

In [109…

```
n.extend([9,10,11,12])# unlike append it adds each and every single element to t
n
```

Out[109…

```
[1, 2, 3, 'hi', 'hello', 4, [5, 6, 7, 8], 9, 10, 11, 12]
```

In [ ]:

```
# WE CAN OBSERVE THE DIFFERENCE BETWEEN append() AND extend()
# append() is treating list as single object and add as complete list
# whereas extend() is (UNPACK ELEMENTS FROM OBJECT)adding each of items in list
# GENERAL EXAMPLE : there is a box with pens and there are two students append a
# teacher said to add few more pens to the box
# stu append has box of pens so, he directly put that box in to teachers box
# stu extend also has box of pens, but he took (unpack the box)out all pens from
```

In [ ]:

```
# Even if it is string it adds individual characters at end
```

In [110…

```
n.extend('hello')
n
```

Out[110…

```
[1,
 2,
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o']
```

In [ ]:

```
# NOTE: it only adds iterables(any object that can be looped over...just like co
```

In [112…

```
n.extend(2)
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[112], line 1
----> 1 n.extend(2)
      2 n

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

In [113…

```
n.extend(1.2)
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[113], line 1
----> 1 n.extend(1.2)
      2 n

TypeError: 'float' object is not iterable
```

In [114…

```
n.extend((1,2,3))# adds tuple elements
n
```

Out[114…

```
[1,
 2,
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3]
```

In [115…

```
n.extend({'key':'value'})#extend dict adds only key
n
```

Out[115…

```
[1,
 2,
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'key']
```

In [118…

```
dict={'key':'values'}
n.extend(dict.values)
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[118], line 2
      1 dict={'key':'values'}
----> 2 n.extend(dict.values)
      3 n

TypeError: 'builtin_function_or_method' object is not iterable
```

In [119…

```
n.extend(dict.values())
n
```

Out[119…

```
[1,
 2,
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'key',
 'key',
 'values']
```

In [120…

```
n.extend(dict.items()) # adds as tuple
n
```

Out[120…

```
[1,
 2,
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'key',
 'key',
 'values',
 ('key', 'values')]
```

insert()

In [ ]:

```
# insert is a method used to insert a value at particular index
```

In [121…

```
n.insert(2)
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[121], line 1
----> 1 n.insert(2)
      2 n

TypeError: insert expected 2 arguments, got 1
```

In [124…

```
n.insert('python',2)
n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[124], line 1
----> 1 n.insert('python',2)
      2 n

TypeError: 'str' object cannot be interpreted as an integer
```

In [ ]:

```
# unlike append insert uses 2 args
# one is index where to insert value(by default 1st arg is index)
#one is value to be inserted(by default 2nd arg is value)
```

In [123…

```
n.insert(2,'python')
```

```
n
```

Out[123…

```
[1,
 2,
 'python',
 3,
 'hi',
 'hello',
 4,
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'key',
 'key',
 'values',
 ('key', 'values')]
```

DIFFERENCE B/W append and insert append accepts only 1 arg and add elements at last of list insert accepts 2 args and adds elements at given index

In [ ]:

```
# append is a method used to add objects at last of list
# extend is a a method used to add object-elements at last of list
# insert is a method used to insert a value at particular index
```

METHODS TO REMOVE ELEMENTS FROM LIST

In [ ]:

```
# to remove items from list we use following methods
# remove()
# pop()
# clear()
```

remove()

In [ ]:

```
# remove() is a method
# it removes the first occurance of the value x
#if x is not there then it throws error
```

In [126…

```
n.remove('key')
n
```

```
Out[126…  [1,
           2,
           'python',
           3,
           'hi',
           'hello',
           4,
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           'values',
           ('key', 'values')]
```

```
In [127…  n.extend([4,4,4,4])
          n
```

```
Out[127…  [1,
           2,
           'python',
           3,
           'hi',
           'hello',
           4,
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           'values',
           ('key', 'values'),
           4,
           4,
           4,
           4]
```

```
In [128…  n.remove(4)
          n
```

Out[128…

```
[1,
 2,
 'python',
 3,
 'hi',
 'hello',
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'values',
 ('key', 'values'),
 4,
 4,
 4,
 4]
```

In [129…

```
n.remove(4)
n
```

Out[129…

```
[1,
 2,
 'python',
 3,
 'hi',
 'hello',
 [5, 6, 7, 8],
 9,
 10,
 11,
 12,
 'h',
 'e',
 'l',
 'l',
 'o',
 1,
 2,
 3,
 'values',
 ('key', 'values'),
 4,
 4,
 4]
```

In [130…

```
n.remove(4)
n
```

```
Out[130…  [1,
           2,
           'python',
           3,
           'hi',
           'hello',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           'values',
           ('key', 'values'),
           4,
           4]
```

```
In [ ]:  # we can see it is remove via occurence of item
```

```
In [131…  n.remove(2,4)# it takes only single arg
          n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[131], line 1
----> 1 n.remove(2,4)
      2 n

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

```
In [132…  n.remove('hello')
          n
```

```
Out[132…  [1,
           2,
           'python',
           3,
           'hi',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           'values',
           ('key', 'values'),
           4,
           4]
```

pop()

```
In [ ]:  # pop is a method
         # it is used to remove value at paticular index
         # pop(i)..>i=index
```

```
In [133…  n.pop(-1)
          n
```

```
Out[133…  [1,
           2,
           'python',
           3,
           'hi',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           'values',
           ('key', 'values'),
           4]
```

```
In [134…  n.pop(-3)
          n
```

```
Out[134…  [1,
           2,
           'python',
           3,
           'hi',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           ('key', 'values'),
           4]
```

```
In [136…  n.pop(1,2)# it takes only single arg
          n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[136], line 1
----> 1 n.pop(1,2)
      2 n

TypeError: pop expected at most 1 argument, got 2
```

```
In [137…  n.pop(3)
          n
```

```
Out[137…  [1,
           2,
           'python',
           'hi',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           ('key', 'values'),
           4]
```

```
In [138…  n.pop(18)
          n
```

```
Out[138…  [1,
           2,
           'python',
           'hi',
           [5, 6, 7, 8],
           9,
           10,
           11,
           12,
           'h',
           'e',
           'l',
           'l',
           'o',
           1,
           2,
           3,
           ('key', 'values')]
```

```
In [139…  n.pop(18)
          n
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[139], line 1
----> 1 n.pop(18)
      2 n

IndexError: pop index out of range
```

clear()

```
In [ ]:  # clear() is a method
         # it is used to remove all the items from list
         # note:only values are remove, variable remains in memory
```

```
In [141…  n.clear(2)# it dont take any arg
          n
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[141], line 1
----> 1 n.clear(2)# it dont take any arg
      2 n

TypeError: list.clear() takes no arguments (1 given)
```

```
In [142…  n.clear()
          n
```

```
Out[142…  []
```

METHODS FOR ORDERING AND SORTING

In [ ]:

```
# we use following methods
# sort()
# reverse()
```

sort()

In [ ]:

```
# sort is a method used to sort elements in accending order
```

In [144…

```
s = [1,4,3,2,6,5,7,2,9,24]
```

In [145…

```
s
```

Out[145…

```
[1, 4, 3, 2, 6, 5, 7, 2, 9, 24]
```

In [161…

```
s.sort()
s
```

Out[161…

```
[1, 2, 2, 3, 4, 5, 6, 7, 9, 24]
```

In [162…

```
s.sort(reverse=True)
s
```

Out[162…

```
[24, 9, 7, 6, 5, 4, 3, 2, 2, 1]
```

In [163…

```
s.sort(reverse=False)
s
```

Out[163…

```
[1, 2, 2, 3, 4, 5, 6, 7, 9, 24]
```

In [164…

```
s.sort(reverse=False)
s
```

Out[164…

```
[1, 2, 2, 3, 4, 5, 6, 7, 9, 24]
```

In [ ]:

```
# you can observe it remain same even we run multiple times
```

reverse()

In [ ]:

```
# reverse is a method used to reverse the items in list

# it also used along with sort() to arrange items in decending order

#sort()==sort(reverse=False)..>accending order

#sort(reverse=True)..>decending order
```

s.reverse() s

In [157…

```
s.reverse()
s
```

Out[157…

```
[24, 9, 7, 6, 5, 4, 3, 2, 2, 1]
```

In [158…

```
s.reverse()
s
```

Out[158…

```
[1, 2, 2, 3, 4, 5, 6, 7, 9, 24]
```

In [ ]:

```
#we can observe every time we run it is changing viceversa
```

In [160…

```
s.sort(reverse=True)
s
```

Out[160…

```
[24, 9, 7, 6, 5, 4, 3, 2, 2, 1]
```

In [165…

```
s.sort(reverse=True)
s
```

Out[165…

```
[24, 9, 7, 6, 5, 4, 3, 2, 2, 1]
```

In [ ]:

```
# you can observe it remain same even we run multiple times
```

METHODS FOR INFORMATION AND COPYING

In [ ]:

```
# we have several methods use to find an element or duplicate the list

#index()
#count()
#copy
```

index()

In [ ]:

```
#index() is a method used to return the index of first item(if duplicate of x is
```

In [168…

```
s # as at last we reverse list it remain same,lets change
```

Out[168…

```
[24, 9, 7, 6, 5, 4, 3, 2, 2, 1]
```

In [171…

```
s.reverse()
s
```

Out[171…

```
[1, 2, 2, 3, 4, 5, 6, 7, 9, 24]
```

In [172…

```
s.index(2)
```

Out[172…

```
1
```

In [173…

```
s.index(2)# every time it is refering to only first occurance
```

Out[173…

```
1
```

In [174…

```
s.index(24)
```

Out[174…

```
9
```

In [178…

```
s.append(20)
```

```
In [179…  s

Out[179…  [1, 2, 2, 3, 4, 5, 6, 7, 9, 24, 20]

In [180…  s.index(20)

Out[180…  10
```

count()

```
In [ ]:  # count is a method returns no.of times a values repeats

In [183…  s.count(2)

Out[183…  2

In [184…  s.count(20)

Out[184…  1
```

copy()

```
In [185…  # copy is a method used to create duplicate(shallow copy) of a list
          #so, the original remains same

In [186…  s1 = s.copy()

In [187…  s1

Out[187…  [1, 2, 2, 3, 4, 5, 6, 7, 9, 24, 20]

In [188…  id(s)

Out[188…  1752966250304

In [189…  id(s1)

Out[189…  1752975488768

In [190…  # ids are different
          #so, changes made to s1 wont affect s

In [191…  s1.remove(2)

In [192…  s1

Out[192…  [1, 2, 3, 4, 5, 6, 7, 9, 24, 20]

In [193…  s1.pop(7)

Out[193…  9

In [194…  s1
```

```
Out[194…   [1, 2, 3, 4, 5, 6, 7, 24, 20]

In [195…   s1.pop(7)

Out[195…   24

In [196…   s1

Out[196…   [1, 2, 3, 4, 5, 6, 7, 20]

In [197…   s1.pop(7)

Out[197…   20

In [198…   s1.pop(-1)

Out[198…   7

In [199…   s1

Out[199…   [1, 2, 3, 4, 5, 6]

In [200…   s

Out[200…   [1, 2, 2, 3, 4, 5, 6, 7, 9, 24, 20]

In [ ]:    # we can see difference
```

BUILT-IN FUNCTIONS

```
In [ ]:    # These are global python function that takes list as input

In [213…   N = [1,2,3,4,5]
           len(N)# returns no.of items in list

Out[213…   5

In [214…   max(N)# returns largest item in list

Out[214…   5

In [215…   min(N)# returns smallest item in list

Out[215…   1

In [216…   sum(N)# returns sum of all numerical values in list

Out[216…   15

In [217…   num = [1,2,5,24,2,12,61,4]
           sorted(num)#returns new sorted list....remain original unchanged

Out[217…   [1, 2, 2, 4, 5, 12, 24, 61]
```

In [219…

```
num# no change in list
```

Out[219…

```
[1, 2, 5, 24, 2, 12, 61, 4]
```

In [218…

```
num = [1,2,5,24,2,12,61,4]
enumerate(num) # pairing an item with its index
# enumerate is a built-in function used to keep track of the index(i)
# of items while you are iterating through a list,string or any iterable
# it just like counter that walks alongside your loop
```

Out[218…

```
<enumerate at 0x198247cdda0>
```

In [13]:

```
string = 'HELLO'
for i  in  enumerate(string):
    print((i))
```

```
(0, 'H')
(1, 'E')
(2, 'L')
(3, 'L')
(4, 'O')
```

In [221…

```
zip(num,N)# pairing an item with another item
# aggregate elements from two or more list in to tuples
# zip() function is used to stitch two or more iterables together
# think of it is like a zipper on jacket : it takes the first item from list1 an
```

Out[221…

```
<zip at 0x198252f5e00>
```

In [16]:

```
lst1 = [1,2]
lst2 = [1,2]
z = zip(lst1,lst2)
z
```

Out[16]:

```
<zip at 0x2d368bcd380>
```

In [ ]:

```
# it is useful when you have related data stored in seperate lists and you want
#process them at the same time in a single loop
#names = ['x','y']
#score = [1,2]
# for name, score in zip(names, scores):
    #print(f"{name} got a score of {score}")

# if length of two lists are different then zip() stops as soon as the short lis
#eg:if l1 = 5items and l2 = 3items then result given only up to 3pairs

#UNZIP()
#you can unzip a collection back into separate list using the *operator
# names, score = zip(*zipped_list)
```

In [ ]:

```
```

LIST COMPREHENSION

In [ ]:

```
'''The Definition
"List comprehension is a concise(giveing a lot of information in simple wors....
one-line syntax in Python

used to create a new list by iterating over an existing sequence.

It combines the functionality of a for loop and an optional

if statement into a single bracketed expression, making the code

more readable and efficient."
```

In [ ]:

```
'''Syntax:
new_list = [expression for item in iterable if condition]
```

In [207…

```
L = [1,2,3,4]
squares = [(n)**2 for n in L]
squares
```

Out[207…

```
[1, 4, 9, 16]
```

In [208…

```
str = 'NareshiTechnologies'
lp = [char for char in str]
lp
```

Out[208…

```
['N',
 'a',
 'r',
 'e',
 's',
 'h',
 'i',
 'T',
 'e',
 'c',
 'h',
 'n',
 'o',
 'l',
 'o',
 'g',
 'i',
 'e',
 's']
```

In [211…

```
str = 'NareshiTechnologies'
vowels =('a','e','i','o','u')
lp = [char for char in str if char in vowels]
lp
```

Out[211…

```
['a', 'e', 'i', 'e', 'o', 'o', 'i', 'e']
```

In [212…

```
str = 'NareshiTechnologies'
vowels =('a','e','i','o','u')
cnts = [char for char in str if char not in vowels]
cnts
```

Out[212…

```
['N', 'r', 's', 'h', 'T', 'c', 'h', 'n', 'l', 'g', 's']
```

CONSTRUCTOR IN LIST

List()

List Constructor is used to convert one iterable/datatype to another list() it is used to modify data in datatypes like tuple and set as tuple is immutable and set is unordered

```
In [2]: string = "Hello,World"
        tuple = (1,2,[3,4],'hi')
        set = {'cse','csm','css'}
        dict = {'key1':'value1','key2':'value2'}
```

```
In [3]: s = list(string)
        s
```

```
Out[3]: ['H', 'e', 'l', 'l', 'o', ',', 'W', 'o', 'r', 'l', 'd']
```

```
In [4]: t = list(tuple)
        t
```

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

```
In [5]: st = list(set)
        st
```

```
Out[5]: ['css', 'cse', 'csm']
```

```
In [6]: d = list(dict)
        d
```

```
Out[6]: ['key1', 'key2']
```

```
In [7]: dt = list(dict.keys())
        dt
```

```
Out[7]: ['key1', 'key2']
```

```
In [8]: dtt = list(dict.values())
        dtt
```

```
Out[8]: ['value1', 'value2']
```

```
In [9]: dct = list(dict.items())
        dct
```

```
Out[9]: [('key1', 'value1'), ('key2', 'value2')]
```

MEMBERSHIP OPERATORS

```
In [ ]: # Membership Operators
        # IN/NOT IN
        #IT checks whether the item exits in the list or not.
```

```
#if it is in list it returns True
#if it is not there it returns False
```

In [1]:
```
lst = ['pen','pencil','book','school']
'pen' in lst
```

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

In [3]:
```
'fruit' in lst
```

Out[3]:
```
False
```

In [4]:
```
'book' not in lst
```

Out[4]:
```
False
```

In [5]:
```
'veg' not in lst
```

Out[5]:
```
True
```

In [7]:
```
# we can use it in conditional statements.
if 'book' in lst:
    print('book is present in list')
else:
    print('not present')
```

```
book is present in list
```

In [8]:
```
if 'fruit' in lst:
    print('fruit is in list')
else:
    print('not present')
```

```
not present
```

In [9]:
```
if 'veg' not in lst:
    print('yes')
else:
    print('no')
```

```
yes
```

List Concatenation & Repetition

In [ ]:
```
#concatenation(+) : concatenation in list is done with (+):Joins two lists into


#Repetition(*) : Repeats the elements of a list a specific number of times
```

In [25]:
```
#Concatenation(+)
l1 = [1,2,3,4]
l2 = [5,6,7,8]
l3 = l1 + l2#Concatenation
l3
```

Out[25]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

In [31]:
```
l4 = [9,10]
l4
```

Out[31]:
```
[9, 10]
```

In [34]:
```
l5 = l3 + l4
l5
```

Out[34]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [37]:
```
#Repetation(*)
l6 = [1,2]
l6*2#Repeatation
```

Out[37]:
```
[1, 2, 1, 2]
```

In [38]:
```
l6 * 5
```

Out[38]:
```
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

In [39]:
```
l7 = [3,4,"hi"]
l7 * 4
```

Out[39]:
```
[3, 4, 'hi', 3, 4, 'hi', 3, 4, 'hi', 3, 4, 'hi']
```

In [40]:
```
l8 = [[]]*3
l8[0]=1
print(l8)
```

```
[1, [], []]
```

In [44]:
```
l8 = [[]]*3
l8[0]=2
print(l8)
```

```
[2, [], []]
```

Slicing For Modification

In [52]:
```
# we have seen how to use slicing to access data, but you can also use it


#to change multiple values at once
lst = [1,2,3,4,5,6,7,8]
new_lst = [9,16,25,36]
lst[2:6] = new_lst
```

In [54]:
```
lst[2:6]
```

Out[54]:
```
[9, 16, 25, 36]
```

In [55]:
```
lst[3:5]
```

Out[55]:
```
[16, 25]
```

The del Keyword

In [56]:
```
# The del keyword is used to simply delete a reference from memory
l = [2,4,6,8]
del l[0]
```

In [57]:
```
l
```

Out[57]:
```
[4, 6, 8]
```

In [58]:
```
del l[0:1]
l
```

Out[58]:
```
[6, 8]
```

In [59]:
```
del l
```

In [60]:
```
l
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[60], line 1
----> 1 l

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

Deep Copy or Shallow Copy

In [ ]:
```
# shallow copy: if you have a nested list,a shollow copy still points to the sam
#list object.changing an item inside the inner list of the original will also ch
```

In [83]:
```
l1 = [1,2,3,4]
l2 = l1.copy()
l2
```

Out[83]:
```
[1, 2, 3, 4]
```

In [84]:
```
l2[3] = 5
l2
```

Out[84]:
```
[1, 2, 3, 5]
```

In [ ]:
```
#Deep copy: uses the copy module (copy.deepcopy(list))
#creates a completely independent version of even the nested list
```

In [76]:
```
import copy
l3 = copy.deepcopy(l1)
```

In [77]:
```
l3[2]=2
```

In [78]:
```
l3
```

```
Out[78]:  [1, 2, 2, 4]
```

```
In [79]:  l1
```

```
Out[79]:  [1, 2, 3, 4]
```