XI – List Manipulation

- Python lists are containers that are used to store a list of values of any type.
- Python lists are **mutable** i.e., you can change the elements of a list in place.
- Strings and tuples are immutable.

Creating And Accessing Lists

```
[]                          ['a', 'b', 'c']
[1, 2, 3]                   ['a', 1, 'b', 3.5, 'zero']
[1, 2.5, 3.7, 9]            ['One', 'Two', 'Three']
```

- The memory address of a list will not change even after you change its values. ← mutable
- Lists and Dictionaries are mutable types; all other data types of Python are immutable.

```
L = list()   or   L = []    # empty list
L1 = [3, 4, [5, 6], 7]      # nested list
```

l1 = list('hello') ⟶ ['h', 'e', 'l', 'l', 'o']

l2 = list(('w', 'e', 'r', 't', 'y')) ⟶ ['w', 'e', 'r', 't', 'y']

- Most commonly used method to input lists is

```
eval(input())
```

eval()

- The eval() function of Python can be used to evaluate and return of an expression given as string.

eval('5+8') ⟶ 13 ⟶ stored as int

eval('3*10') ⟶ 30

- Sometimes (not always) eval() does not work in Python shell.

Accessing Lists

- l

0	1	2	3	4
a	e	i	o	u
-5	-4	-3	-2	-1

l[0] = 'a' = l[-5]

l[1] = 'e' = l[-4]

l[2] = 'i' = l[-3]

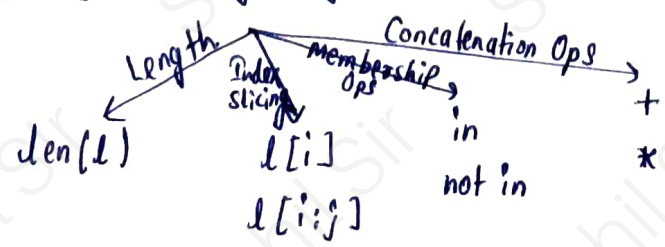
- L → 7050

0	→ 17
1	→ True
2	→ "good"
3	→ 4.59

Each of the individual items of the list are stored somewhere else in memory.

Here lie the memory addresses of the individual items (stored somewhere else)

- List [a:b] ⟶ give you elements between indexes a to b-1.



```
vowels = ['a', 'e', 'i', 'o', 'u']
```

vowels [0] → 'a'
vowels [4] → 'u'
vowels [-1] → 'u'
vowels [-5] → 'a'

vowels [5]
Index Error: list index out of range.

```
vowels [0] = 'A'  → ['A', 'e', 'i', 'o', 'u']
vowels [-1] = 'E' → ['A', 'E', 'i', 'o', 'u']
```

-
```
L = ['p', 'y', 't', 'h', 'o', 'n']
for a in L:
    print(a)
```

p
y
t
h
o
n

```
L = ['p', 'y', 't', 'h', 'o', 'n']
for i in range(len(L)):
    print(L[i])
```

p
y
t
h
o
n

-
```
L1, L2 = [1, 2, 3], [1, 2, 3]
L3 = [1, [2, 3]]

L1 == L2    # True
L1 == L3    # False

L1 < L2     # False
L1 < L3     # Type Error (int & list Comparison)
```

-
```
a = [2, 3] ,  c = ['2', '3']
a == c      # False
```

-
```
a = [2, 3] ,  d = [2.0, 3.0]
a == d      # True  {Compared values only}
```

List Operations

+ ← Concatenation operator

* ← Replication Op.

Joining Lists

```
l1 = [1, 3, 5]
l2 = [6, 7, 8]
l1 + l2
[1, 3, 5, 6, 7, 8]
```

⊙ l1 + 5
l1 + Complex no. } Type Error.
l1 + string

Repeating or Replicating Lists

```
l1 * 3
[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Slicing the Lists

not including stop

l[start : stop]

```
l1 = [1, 3, 5, 7, 9]
l1[1:4] → [3, 5, 7]
l1[:] → [1, 3, 5, 7, 9]
l1[::-1] → [9, 7, 5, 3, 1]
l1[:-3] → [1, 3]
```

Using Slices for List Modification

```
l = ["one", "two", "THREE"]
l[0:2] = [0, 1]
# [0, 1, "THREE"]
```

Imp
```
l[0:2] = "a"
# ["a", "THREE"]
```

The values being assigned must be a sequence i.e., list or string or tuple etc.

```
l = [1, 2, 3]
l[2:] = "604"
# [1, 2, 3, '6', '0', '4']
```

```
l[2:] = 345   ← is not a sequence
# Type Error: can only assign an iterable.
```

```
l = [1, 2, 3]
l[10:20] = "abcd"
# [1, 2, 3, 'a', 'b', 'c', 'd']
```

Pg 332

List Operations

+ ← Concatenation operator

* ← Replication Op.

Joining Lists

```
l1 = [1, 3, 5]
l2 = [6, 7, 8]
l1 + l2
[1, 3, 5, 6, 7, 8]
```

⊙ l1 + 5
l1 + Complex no.
l1 + string
} Type Error.

Repeating or Replicating Lists

```
l1 * 3
[1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Slicing the Lists

not including stop

l[start: stop]

```
l1 = [1, 3, 5, 7, 9]
l1[1:4] → [3, 5, 7]
l1[:]   → [1, 3, 5, 7, 9]
l1[::-1] → [9, 7, 5, 3, 1]
l1[:-3] → [1, 3]
```

Using Slices for List Modification

```
l = ["one", "two", "THREE"]
l[0:2] = [0, 1]
# [0, 1, "THREE"]
```

Imp
```
l[0:2] = "a"
# ["a", "THREE"]
```

The values being assigned must be a sequence i.e., list or string or tuple etc.

```
l = [1, 2, 3]
l[2:] = "604"
# [1, 2, 3, '6', '0', '4']

l[2:] = 345   ← is not a sequence.
# Type Error: can only assign an iterable.
```

```
l = [1, 2, 3]
l[10:20] = "abcd"
# [1, 2, 3, 'a', 'b', 'c', 'd']
```

} Pg 322.

Lists Methods

<listObject>.<method name>()

L.append(item)

Adds an item to the end of the list.
Takes exactly one element and returns no value.

```
l = [10, 12, 14]
l.append(16)
l
# [10, 12, 14, 16]
```

L[index] = new value

```
l[2] = 24
# [10, 12, 24]
```

L.extend(<list>)

Takes exactly one element (a list type) and returns no value.

```
t1 = ['a', 'b', 'c']
t2 = ['d', 'e']
t1.extend(t2)  # ['a','b','c','d','e']
```

```
t3 = t1.extend(t2)
t3  ← t3 is empty()
```

```
l = [1,2,3]
l2 = l.append(12)
l2  ← is empty() as append() did not return any value
```

del list[index] / del list[start:stop]

```
l = [1,2,3,4,5,6]
del l[3]
# [1,2,3,5,6]
del l[2:5]
# [1,2]
del l   (delete all elements & list object too.)
```

pop() method

L.pop(index)

```
l = [1,2,3,4,5]
l.pop()   # 5
l.pop(3)  # 4
```

pop() method is useful only when you want to store the element being deleting for later use.

```
item1 = L.pop()
item2 = L.pop(0)   ← index
item3 = L.pop(5)   ← index
```

List.insert(<pos>, <item>)

- Takes two arguments and returns no value.

```
t1 = ['a','e','u']
t1.insert(2, 'i')  # ['a','e','i','u']
```

List.index(<item>)

returns index of first matched item from the list.

```
l = [1, 3, 7, 8]
l.index(3) → 1
l.index(10) → ValueError : 10 is not in list.
```

- While del statement can remove a single element or a list-slice from a list, the pop() can remove only single element, not list slices.
  Also, pop() returns the deleted element too.

- While append() function adds one element to a list, extend() can add multiple elements from a list supplied to it as argument.

```
t2.extend(10) → Type Error 'int' object is not iterable
t2.extend([10]) → Correct.
```

```
t1.append(12, 14)
Type Error: ← one argument → two given.
```

```
t1.append([12,14])
[1, 3, 5, 10, [12, 14]]
```

L.index(<item>)	L.extend(<list>)
L.append(<item>)	L.insert(<pos>, <item>)

L.pop(<index>)	L.clear()	L.reverse()
L.remove(<value>)	L.count(<item>)	L.sort()

L.remove(<value>)

- Takes one essential argument and does not return anything.

```
l = ['a', 'e', 1, 2, 3]
l.remove('a')   # ['e', 1, 2, 3]
l.remove('i')   # ValueError : x not in list.
```

L.clear()

- removes all the items from the list and list becomes empty list
- This functions returns nothing.

```
l1 = [2, 3, 4, 5]
l1.clear()      # del l1, it will delete all the elements and list object too.
l1              # []
```

After clear(), the list object still exists as an empty list.

L.count(<item>)

- returns the count of the item

```
l = [13, 18, 20, 10, 18, 23]
l.count(18)   # 2
l.count(28)   # 0
```

L.reverse()

- Takes no argument, return no list; reverses the list 'in place' and does not return anything.

```
t1 = ['a', 'e', 'i', 'o', 'u']
t1.reverse()
t1   # ['u', 'o', 'i', 'e', 'a']
t3 = t1.reverse()
t3   # empty does not return anything.
```

L.sort()

- Sorts the items of the list, by default ascending order.
- "in place", does not return anything

```
l = [1, 2, 3, 6, 5]
l.sort()
l   # [1, 2, 3, 5, 6]
l.sort(reverse = True)   # decreasing order
```