CLASS – XI

LISTS MANIPULATION

STUDY NOTE

Period -01

Introduction:

The Python lists are containers that are used to store a list of values of any type. Unlike other variables Python lists are mutable i.e, you can change the elements of a list in place; Python will not create a fresh list when you make changes to an element of a list. List is a type of sequence like strings and types but it differs from them in the way that lists are mutable but strings and tuples are immutable. This chapter is dedicated to basic list manipulation in Python. We shall be talking about creating and accessing lists, various list operations and list manipulations through some built-in functions.

Creating and Accessing lists:-

A list is a standard data type of Python that can stone a sequence of values belonging to any type. The lists are depicted through square brackets, e.g following are some lists in Python.

```
[ ]                          # list with no member, empty list
[1, 2, 3]                    # list of integers
[1, 2, 5, 3, 7, 9]           # list of numbers (integers and floating point)
['a', 'b', 'c']              # list of characters
['a', 1, 'b', 3.5, 'zero']   # list of mixed value types
['One', 'Two', 'Three']      # list of strings
```

Before we proceed and discuss how to create lists, one thing that must be clear is that Lists are mutable (i.e modifiable) i.e you can change elements of a list in place. In other words, the memory address of a list will not change even after you change its values. List is one of the two mutable types of Python - List and Dictionaries are mutable types; all other data types of Python are immutable.

Creating Lists:-

To create a list, put a number of expressions in square brackets. That is, use square brackets to indicate the start and end of the list, and separate the items by commas. For example:

```
[2, 4, 6]
['abc', 'def']
[1, 2.0, 3, 4.0]
```

[ ]

Thus to create a list you can write in the form given below:

L = [ ]

L = [value, ...]

This construct is known as a list display construct.

Consider some more examples.

- **The empty list** :- The empty list is [ ]. It is the list equivalent of 0 or ' ' and like them it also has truth value as false. You can also create an empty list as: L = list ( ). It will generate an empty list and name that list as L.
- **Long lists**:- If a list contains many elements, then to enter such long lists, you can split it across several lines, like below. sqrs = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625]. Notice the opening square bracket and closing square brackets appear just in the beginning and end of the list.
- **Nested lists** :- A list can have an element in it, which itself is a list. Such a list is called nested list, e.g. L1 = [3, 4, {5, 6], 7]. L1 is a nested list with four elements : 3, 4, [5, 6] and 7. L1[2] element is a list [5, 6]. Length of L1 is 4 as it counts [5, 6] as one element. Also, as L1[2] is a list (i.e [5, 6]), which means L1[2][0] will give 5 Li[2][1] will give 6. We shall talk about nested lists in details in class XII.

Creating lists from Existing sequences:- You can also use the built-in list type object to create lists from sequences as per the syntax given below:

L = list (<sequence>)

Where <sequence> can be any kind of sequence object including strings, tuples, and lists. Python creates the individual elements of the list from the individual elements of passed sequence. If you pass in another list, the list function makes a copy.

Consider following example:-

```
>>> 11 = list ('hello')
>>> 11
['h', 'e', '1', '1', 'o']
>>> t = ('w', 'e', 'r', 't', 'y')
>>> 12 = list (t)
>>> 12
['w', 'e', 'r', 'y']
```

You can use this method of creating lists of single characters or single digits via keyboard input. Consider the code below.

```
11 = list (input('Enter list elements:'))
enter list elements : 234567
>>> 11
['2', '3', '4', '5', '6', '7']
```

Notice, this way the data type of all characters entered is string even though we entered digits. To enter a list of integers through keyboard, you can use the method given below.

```
list = eval (input("Enter list to be added:"))
print ("list you entered:", list)
```

When you execute it, it will work somewhat like:

```
Enter list to be added : [67, 78, 46, 23]
list you entered : [67, 78, 46, 23]
```

Please note, sometimes (not always) eval () does not work in Python Shell. At that time, you can run it through a script or program too.

The eval Function:-

The eval () function of Python can be used to evaluate and return the result of an expression given as string. For example:

```
eval ('5 + 8')
```

will give you result as 13

Similarly, following code fragment

```
y = eval ("3 * 10")
print (y)
```

will print value as 30

Since eval () can interpret an expression given as string, you can use it with input () too:

```
var 1 = eval(input("Enter value :"))
print(val1, type (var1))
```

Executing this code will result as:

```
Enter value: 15 + 3
18 <class 'int'>
```

See, the eval () has not only interpreted the string "15+3" as 1B but also stored the result as int value. Thus will eval (), if you enter an integer or float value, it will interpret the value as the intended types.

```
var1 = eval(input ("Enter Value:"))
print (var1, type (var1))
var1 = eval (input("Enter Value:"))
print (var1, type(var1))
Enter value : 75
75 <class 'int'>
Enter value : 89.9
89.9 <class 'float'>
```

You can use eval ( ) to enter a list or tuple also. Use [ ] to enter lists' and () to enter tuple values.

```
var1 = eval (input("Enter Value:"))
print (var1, type (var1))
var1 = eval (input("Enter Value:"))
print (var1, type(var1))
Enter Value : [1, 2, 3]
[1, 2, 3] <class 'list'>
Enter Value : (2, 4, 6, 8)
(2, 4, 6, 8) <class 'tuple'>
```

However, use of eval() may lead to unforeseen problems, and thus, use to eval () is always discouraged.

Quick Interesting Facts:

- Lists are mutable sequences of Python i.e., you can change elements of a List in place.

PERIOD – 2

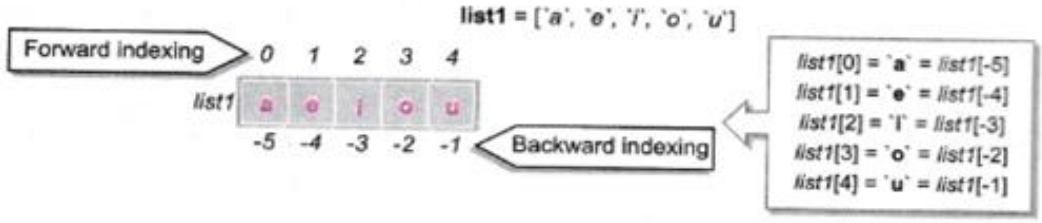
List Operations (1)

Accessing Lists:-

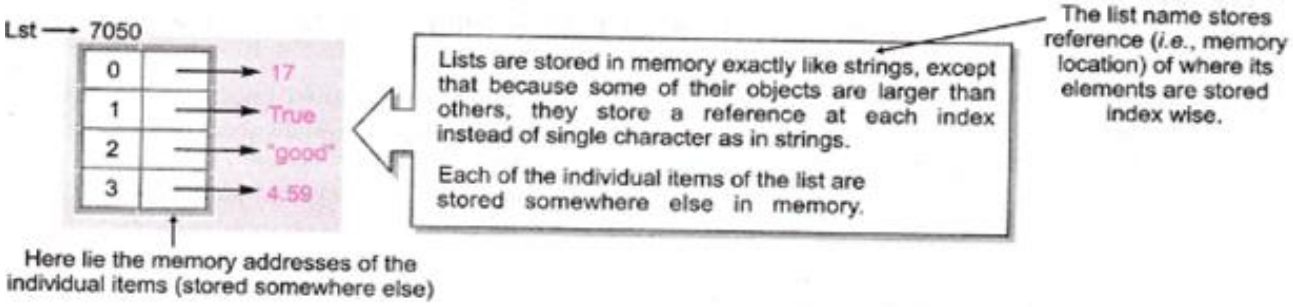
Lists are mutable (editable) sequences having a progression of elements. There must be a way to access its individual elements and certainly there is. But before we start accessing individual elements, let us discuss the similarity of Lists with strings that will make it very clear to you how individual elements are accessed in lists - the way you access string elements. Following subsection will make the things very clear to you, I bet!

Similarity with strings

Lists are sequences just like strings that you have read in previous chapter. They also index their individual elements, just like strings do. Recall figure of chapter 7 that talks about indexing in strings. In the same manner, list-elements are also indexed, i.e forward indexing as 0, 1, 2, 3, ... and backward indexing as -1, -2, -3, ... see the figure.



(a) List Elements' two way indexing
Lst = [47, True, "good", 4.59]



(b) How lists are internally organized

Thus, you can access the list elements just like you access a string's elements e.g List[i] will give you the element at i^{th} index of the list; List[a:b] will give you elements between indexes a to b - 1 and so on.

Put in another words, lists are similar to strings in following ways:

Length :- Function len(L) returns the number of items (count) in the list L.

Indexing and slicing: - L[i] returns the items at index i (the first item has index 0), and L[i : j] returns a new list, containing the objects at indexes between i and j (excluding index j).

Membership operators: - Both 'in' and 'not in' operators work on Lists just like they work for other sequences. This is, in tells if an element is present in the list or not, and not in does the opposite.

Concatenation and replication operators + and * :- The + operator adds one list to the end of another. The * operator repeats a list. We shall be talking about these two operations in a later section 11.3 - List Operations.

Accessing Individual Elements:- As mentioned, the individual elements of a list are accessed through their indexes. Consider following examples:

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels [0]
'a'
>>> vowels[4]
'u'
>>> vowels [-1]
'u'
>>> vowels [-5]
'a'
```

Like strings, if you give index outside the legal indices (0 to length -1 or - length, - length +1, ... uptill -1) while accessing individual elements, Python will raise Index Error (see below)

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> vowels [5]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    vowels[5]
IndexError: list index out of range
```

Quick Interesting Facts:

- To create a List, put a number of comma-separated expressions in Square brackets.
- The empty Square brackets i.e [ ] indicate an empty List.

PERIOD – 3

LIST OPERATIONS (2)

Difference from Strings:- Although lists are similar to strings in many ways, yet there is an important difference in mutability of the two. Strings are not mutable, while lists are. You cannot change individual elements of a string in place, but Lists allow you to do so. That is following statement is fully valid for Lists (though not for strings)

L(i) = <element>

For example, consider the same vowels list created above, that stores ll vowel in lower case. Now, if you want to change some of these vowels, you may write something as shown below?

```
>>> vowels [0] = 'A'
>>> vowels
['A', 'e', 'i', 'o', 'u']
>>> vowels [-4] = 'E'
>>> vowels
['A', 'E', 'i', 'o', 'u']
```

Traversing a list:- Recall that traversal of a sequence means accessing and processing each element of it. Thus traversing a list also means the same and same is the tool for it, i.e, the Python loops. That is why sometimes we call a traversal as looping over a sequence.

The for loop makes it easy to traverse or loop over the items in a list, as per following syntax:

```
for <item> in < List> :
        Process each item here
```

For example, following loop shows each item of a list L in separate lines:

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

The above loop will produce result as:

```
p
y
t
h
o
n
```

How it works:-

The loop variable a in above loop will be assigned the list elements, one at a time. So, loop - variable a will be assigned 'P' in first iteration and hence 'P' will be printed; in second iteration, a will get element 'y' and 'y' will be printed; and so on.

If you only need to use the indexes of elements to access them, you can use functions range () and len() as:

```
for index in range (len (L)):
        Process List [index] here
```

Consider the following program that traverses through a list using above format and prints each item of a list L in separate lines along with its index.

Program:- Program to print elements of a list ['q', 'w', 'e', 'r', 't', 'y'] in separate lines along with element's both indexes (positive and negative)

```
L = ['q', 'w', 'e', 'r', 't', 'y']
length = len (L)
for a in range (length) :
    print ("At indexes", a, "and", (a - length), "element :", L[a])
```

Sample run of above program is:

```
At indexes 0 and -6 element: q
At indexes 1 and -5 element: w
At indexes 2 and -4 element: e
At indexes 3 and -3 element: r
At indexes 4 and -2 element: t
At indexes 5 and -1 element: y
```

Comparing Lists:- You can compare two lists using standard comparison operation operators of Python, i,e, <, >, ==, !, =, etc. Python internally compares individual elements of lists (and tuples) in lexicographical order. This means that to compare equal, each corresponding element must compare equal and the two sequences must be of the same type i.e, having comparable types of values.

Consider following examples:-

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

True

>>> L1 == L3

False

For comparison operators <, <, >=, <=, the corresponding elements of two lists must be of comparable types, otherwise Python will give error.

Consider the following considering the above two lists:

>>> L1 < L2

False

>>> L1 < L3

Traceback (most recent call last) :

File "<ipython-input - 180-84 fdf598c3f1>", line 1, in <module>

L1 < L3

TypeError : '<' not supported between instances of 'int' and 'list'

Python raised error above because the corresponding second elements of list i.e., L1[1] and L3[1] are not of comparable types. L1[1] is integer (2) and L3[1] is a list[2,3] and list and numbers are not comparable types in Python.

Python gives the final result of non-equality comparisons as soon as it gets a result in terms of True/False from corresponding elements' comparison. If corresponding elements are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big) see below.

Table Non-equality comparison in list sequences:

Comparison	Result	Reason
[1, 2, 8, 9] < [9, 1]	True	Gets the result with the comparison of corresponding first elements of two lists 1 < 9 is True
[1, 2, 8, 9] < [1, 2, 9, 1]	True	Gets the result with the comparison of corresponding third elements of two lists 8 < 9 is True
[1, 2, 8, 9] < [1, 2, 9, 10]	True	Gets the result with the comparison of corresponding third elements of two lists 8 < 9 is True
[1, 2, 8, 9] < [1, 2, 8, 4]	False	Gets the result with the comparison of corresponding fourth elements of two lists 9 < 4 is False

So, for comparison purposes, Python internally compares individual elements of two lists, applying all the comparison rules that you have read earlier. Consider following code:

```
>>> a = [2, 3]                    >>> a > b
>>> b = [2, 3]                    False
>>> c = ['2', '3']
>>> d = [2.0, 3.0]                >>> d > a
>>> e = [2, 3, 4]                 False
>>> a = = b                       >>> d ==a
True                              True
>>> a = = c                       >>> a < e
False                             True
```

List Operations:-

The most common operations that you perform with lists include joining lists, replicating lists and slicing lists. In this section, we are going to talk about the same.

Joining Lists:-

Joining two lists is very easy just like you perform addition, literally ;-). The concatenation operator +, when used with two lists, joins two lists. Consider the example given below.

```
>>> 1st1 = [1, 3, 5]
>>> 1st 2 = [6, 7, 8]
>>> 1st 1 + 1st 2
[1, 3, 5, 6, 7, 8]
```

As you can see that the resultant list has firstly elements of first list lst1 and followed by elements of second list lst2. You can also join two or more lists to form a new list, e.g.

```
>>> 1st 1 = [10, 12, 14]
>>> 1st 2 = [20, 22, 24]
>>> 1st 3 = [20, 32, 34]
>>> 1st = 1st 1 + 1st 2 + 1st 3
>>> 1st
[10, 12, 14, 20, 22, 24, 30, 32, 32]
```

The '+' operator when used with lists requires that both the operands must be of list types. You cannot add a number or any other value to a list. For example, following expression will result into error:

list + number

list + complex - number

list + string

Consider the following examples

```
>>> 1st1 = [10, 12, 14]
>>> 1st 1 + 2
Traceback (most recent call last) :
File "<pyshell #42>", line 1, in <module>
        1st1+2
TypeError : can only concatenate list (not "int") to list
>>> 1st1 + "abc"
Traceback (most recent call last) :
File "<phshell#44>", line 1, in <module>
        1st 1 + "abc"
TypeError : can only concatenate list (not "str") to list
```

Quick Interesting Facts:

- Lists index their elements just like strings , i.e two way indexing.
- Lists are stored in memory exactly like strings, except that because some of their objects are larger than others, they store a reference at each index instead of single character as in strings.

PERIOD – 4

LIST OPERATIONS (3):

Slicing the Lists:-

List slices, like string slices are the sub part of a list extracted out. You can use indexes of list elements to create list slices as per following format:

seq = L [start : stop]

The above statement will create a list slice namely seq having elements of list L on indexes start, start+1, start+2, ..., stop -1. Recall that index on last limit is not included in the list slice. The list slice is a list in itself, that is, you can perform all operations on it just like you perform on lists.

Consider the following example:

```
>>> 1st = [10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> seq = 1st [3 : -3]
>>> seq
[20, 22, 24]
>>> seq [1] = 28
>>> seq
[20, 28, 24]
```

For normal indexing, if the resulting index is outside the list, Python raises an IndexError exception. Slices are treated as boundaries instead, and the result will simply contain all items between the boundaries. For the start and stop given beyond list limits in a list slice (i.e out of bounds), Python simply returns the elements that fall between specified boundaries, if any, without raising any error.

For example, consider the following:

```
>>> 1st = [10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> 1st [3 : 30]
[20, 22, 24, 30, 32, 34]
>>> 1st [-15 : 7]
[10, 12, 14, 20, 22, 24, 30]
>>> L1 [2, 3, 4, 5, 6, 7, 8]
>>> L1 [2 : 5]
[4, 5, 6]
>>> L1 [6 : 10]
```

```
[8]
>>> L1 [10 : 20]
[ ]
```

Lists also support slice steps. That is, if you want to extract, not consecutive but every other element of the list, there is a way out - the slice steps. The slice steps are used as per following format:

seq = L [start : stop : step]

Consider some examples to understand this.

```
>>> 1st
[10, 12, 14, 20, 22, 24, 30, 32, 34]
>>> 1st [0 : 10 : 2]
[10, 14, 22, 30, 34]
>>> 1st [2 : 10 : 3]
[14, 24, 34]
>>> 1st [: : 3]
[10, 20, 30]
```

Consider some more examples:

```
seq = L [: : 2]          # get every other item, starting with the first
seq = L [5 : : 2]        # get every other item, starting with the sixth element, i.e index 5
```

Like strings, if you give <Listname> {: : -1], it will reverse the list e.g for a list say

List = [5, 6, 8, 11, 3], following expression will reverse it :

```
>>> List [: : -1]
[3, 11, 8, 6, 5]
```

Program: - Extract two list-slices out of a given list of numbers. Display and print the sum of elements of first list-slice which contains every other element of the list between indexes 5 to 15. Program should also display the average of elements in second list slice that contains every fourth element of the list. The given list contains numbers from 1 to 20.

```
1st = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
slc1 = 1st [5 : 15 : 2]
slc2 = 1st [: : 4]
sum = avg = 0
print ("slice 1')
```

```
for a in slc1 :
        sm + = a
        print (a, end = ' ')
print ( )
print ("sum of elements of slice 1:", sum)
print ("Slice 2")
sum = 0
for a in slc2 :
        sum + = a
        print (a, end = ' ')
print ( )
avg = sum/ len (slc2)
print ("Average of elements of slice 2:", avg)
```

Using Slices for List Modification:-

You can use slices to overwrite one or more list elements with one or more other elements.

Following examples will make it clear to you:

```
>>> L = ["one", "two", "THREE"]
>>> L[0:2] = [0, 1]
>>> L
[0, 1, "THREE"]
>>> L = ["one", "two", "THREE"]
>>> L[0 : 2] = "a"
>>> L
["a", "THREE"]
```

In all the above examples, we have assigned new values in the form of a sequence. The values being assigned must be a sequence, i.e, a list or string or tuple etc.

For example, following assignment is also valid.

```
>>> L1 = [1, 2, 3]
>>>L1 = [2 :  ] = "608"
>>> L1
[1, 2, '6', '0', '4']
```

But if you try to assign a non-sequence value to a list slice such as a number, Python will give an error, i.e

```
>>> L1 [2:] = 345
File "<.....>", line 1, in <module>
L1 [2:] = 345
Type Error : can only assign an iterable
```

But here, you should also know something. If you give a list slice with range much outside the length of the list, it will simply add the values at the end e.g

```
>>> L1 = [1, 2, 3]
>>> L1 = [10 : 20] = "abcd"
>>> L1
[1, 2, 3, 'a', 'b', 'c', 'd']
```

Working with lists:-

Now that you have learnt to access the individual elements of a list, let us talk about how you can perform various operations on lists like : appending, updating, deleting etc.

Appending Elements to a list

You can also add items to an existing sequence. The append () method adds a single item to the end of the list. It can be done as per following format:

L . append (item)

Consider some examples:

```
>>> 1st 1 = [10, 12, 14]
>>> 1st1.append (16)
>>> 1st 1
[10, 12, 14, 16]
```

Updating elements to a list

To update or change an element of the list in place, you just have to assign new value to the element's index in list as per syntax:

L[index] = <new value>

Consider following examples:

```
>>> 1st1 = [10, 12, 14, 16]
>>> 1st1[2] = 24
>>> 1st1
```

[10, 12, 24, 16]

Deleting Elements from a list:

You can also remove items from lists. The del statement can be used to remove an individual item, or to remove all items identified by a slice. It is to be used as per syntax given below:

```
del List [<index>]          # to remove element at index
del List [<start> : <stop>] # to remove elements in list slice
```

Consider following examples:

```
>>> 1st = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> del 1st [10]
>>> 1st
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> del 1st [10 : 15]
>>> 1st
[1, 2, 3, 4, 5, 7, 8, 9, 10, 17, 18, 19, 20]
```

If you use del <lstname> only e.g del lst, it will delete all the elements and the list object too. After this, no object by the name lst would be existing.

You can also use pop () method to remove single element, not list slices. The pop() method removes an individual item and returns it. The del statement and the pop method do pretty much the same thing except that pop method also returns the removed item along with deleting it from list.

The pop () method is used as per following format:

```
List.pop (<index>)    # index optional; if skipped, last element is deleted
```

Consider examples below:

```
>>> 1st = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> 1st. pop ()
20
>>> 1st.pop (10)
11
```

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

```
item 1 = L.pop ()          # last item
item 2 = L.pop (0)         # first item
item3 = L.pop (5)          # sixth item
```

Now you can use item 1, item2 and item3 in your program as per your requirement.

Quick Interesting Facts:

- Lists are similar to strings in many ways like indexing, slicing and accessing individual elements and they are mutable just like strings are

PERIOD-05

List Functions and Methods (1):-

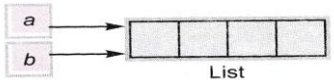
Making True Copy of a List

Sometimes you need to make a copy of a list and you generally tend to do it using assignment, e.g.,

```
a = [ 1, 2, 3 ]
b = a
```

Trying to make copy of list a in list b

But it will not make b as a duplicate list of a; rather just like Python does, it will make label b to point to where label a is pointing to, i.e., as shown in adjacent figure.



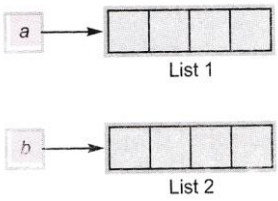
It will work fine as long as we do not modify lists. Now, if you make changes in any of the lists, it will be reflected in order because a and b are like aliases for the same list. See below.

```
>>> a = [1, 2, 3]
>>>b = a
>>>a[1] = 5
>>> a
[ 1, 5, 3 ]
>>>b
[1, 5, 3]
```

See, list b is also reflecting the changed value of list a.

See, along with list a, list b also got modified. What if you want that the copy list b should remain unchanged while we make changes in list a?

That is, if you want both lists to be independent of each other as shown in adjacent figure. For this, you should create copy of list as follows:




```
b = list (a)
```

Now a and b are separate lists, See below :

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> a[1] = 5
>>> a
[1, 5, 3]
>>>b
[1, 2, 3]
```

Now, list b is not reflect the changed value. List b is independent of list a.

Thus, true independent copy of a list is made via list( ) method; assigning a list to another identifier just creates an alias for same list.

Python also offers many built-in functions and methods for list manipulation. You have already worked with one such method len () in earlier chapters. In this chapter, you will learn about many other built-in powerful list methods of Python used for list manipulation. Every list object that you create in Python is actually an instance of List class (you need not do anything specific for this; Python does it for you- you know built-in). The list manipulation methods that are being discussed below can be applied to list as per following syntax:

<listobject>.<method name> ()

In the following examples, we are referring to <listObject> as list only (no angle brackets but the meaning is intact i.e you have to replace List with a legal list (i.e either a list literal or a list object that holds a list).

1. The index method:

   This function returns the index of first matched item from the list. It is used as per following format:

List. index (<item>)

For example, for a list L1 = [13, 18, 11, 16, 18, 14]

```
>>> L1.index (18)
1
```

However, if the given item is not in the list, it raises exception value Error (see below)

```
List.index (33)
Traceback (most recent call last):
File "(ipython-input-60-038fc9bf9c5c>", line 1, in <module>
list.index (33)
ValueError : 33 is not in list
```

2. The append method:

As you have read earlier in section 12.4 that the append () method adds an item to the end of the list. It works as per following syntax:

List.append (<item>)

- Takes exactly one element and returns no value

For example, to add a new item "yellow" to a list containing colours, you may write:

```
>>> colours = ['red', 'green', 'blue']
>>> colours . append ('yellow')
>>> colours
['red', 'green', 'blue', 'yellow']
```

The append() does not return the new list, just modifies the original.

To understand this, consider the following example:

```
>>> 1st = [1, 2, 3]
>>> 1st2=1st.append(12)
>>> 1st2
>>> 1st
[1, 2, 3, 12]
```

3. The extend method:

The extend () meth of is also used for adding multiple elements (given in the form of a list) to a list. But it is different from append (). First, let us understand the working of extend () function then we will talk about the difference between these two functions. The extend () function works as per following format:

List. extend (<list>)

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

That is extend () takes a list as an argument and appends all of the elements of the argument list to the list object on which extend() is applied. Consider following example:

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

The above example left list t2 unmodified. Like append (), extend () also does not returned any value. Consider following example:

```
>>> t3 = t1 . extend (t2)
>>> t3
```

Difference between append () and extend ()

While append () function adds one element to a list, extend () can add multiple elements from a list supplied to it as argument. Consider the following example that will make it clear to you:

```
>>> t1 = [1, 3, 5]
>>> t2 = [7, 8]
>>> t1.append (10)
>>> t1
[1, 3, 5, 10]
>>> t1.append (12, 14)
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module>
        t1 . append (12, 14)
TypeError : append () takes exactly one argument (2 given)
>>> t1 . append ([12, 14])
>>> t1
[1, 3, 5, 10, [12, 14]]
>>> len (t1)
5
```

```
>>> t2.extend (10)
Traceback (most recent call last):
        File "<pyshe11#7>, line 1, in <module>
        t2.extend (10)
TypeError : 'int' object is not iterable
>>> t2. extend ([12, 14])
>>> t2
[7, 8, 12, 14]
>>> t3 = [20, 40]
>>> t2.extend (t3)
>>> t2
[7, 8, 12, 14, 20, 40]
>>> len (t2)
6
```

Quick Interesting Facts:

- Membership operator 'in' tells an element is present in the List or not and 'not in' does the opposite.

PERIOD-06

List Functions and Methods (2):-

4. The insert method: - The insert () method is also an insertion method for lists, like append and extend methods. However, both append () and extend () insert the element (s) at the end of the list. If you want to insert an element somewhere in between or any position of your choice, both append () and extend () are of no use. For such a requirement insert () is used. The insert () function inserts an item at a given position. It is used as per following syntax:

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

   - Takes two arguments and returns to no value.

   The first argument <pos> is the index of the element before which the second argument <item> is to be added.

Consider the following example:

```
>>> t1 = ['a', 'e', 'u']
>>>t1 . insert (2, 'i')          # inset element 'i' at index 2.
>>> t1
['a', 'e', 'i', 'u']
```

For function insert (), we can say that:

list. insert (0, x) will insert element x at the front of the list i.e at index 0 (zero)

list.insert (len (a), x) will insert element x at the end of the list-index equal to length of the list;

thus it is equivalent to list.append(x). If index is greater than len(list), the object is simply appended. If, however, index is less than zero and not equal to any of the valid negative indexes of the list (depends on the size of the list), the object is pretended, i.e added in the beginning of list e.g., for a list t1 = ['a', 'e', 'i', 'u'] if we do:

```
>>> t1 . insert (-9, 'k')          # valid negative indexes are -1, -2, -3, -4
>>> t1
["k', 'a', 'e', 'i', 'u']          # list pretended with element 'k'
```

5. The pop method :- You have read about this method earlier. The pop () is used to remove the item from the list. It is used as per following syntax:

List. pop (<index>) # <index is optional argument

- Takes one optional argument and returns a value - the item being deleted

Thus, pop () removes an element from the given position in the list, and return it. If no index is specified, pop () removes and returns the last item in the list. Consider some examples:

```
>>> t1
['k', 'a', 'e', 'i', 'p', 'q', 'u']
>>> ele1 = t1.pop (0)
>>> ele1
'k'
>>> t1
['a', 'e', 'i', 'p', 'q', 'u']
>>> ele2 = t1.pop ()
>>> ele 2
'u'
>>> t1
```

['a', 'e', 'i', 'p', 'q']

The pop () method raises an exception (runtime error) if the list is already empty.

Consider this:

```
>>> t2 = [ ]                # empty list
>>> t2.pop ()
Traceback (most recent call last) :
File "<pyshell #31>", line 1, in <module>
        t2.pop ()
IndexError : pop from empty list
```

6. The remove method:- While pop () removes an element whose position is given, what if you know the value of the element to be removed, but you do not know its index or position in the list? Well, Python though it in advance and made available the remove () method. The remove () method removes the first occurrence of given item from the list. It is used as per following format:

   List. remove (<value>)- Takes one essential argument and does not return anything.

   The remove () will report an error if there is no such item in the list. Consider some examples:

```
>>> t1 = ['a', 'e', 'i', 'p', 'q', 'a', 'q', 'p']
>>> t1.remove ('a')
>>> t1
['e', 'i', 'p', 'a', 'q', 'p']
>>> t1.remove ('p')
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.remove ('ak')
Traceback (most recent call last) :
File "<physhell #45>", line 1, in <module>
t1.remove ('k')
ValueError : list. remove (x) : x not in list
```

7. The clear method: - This method removes all the items from the list and the list becomes empty list after this function. This function returns nothing. It us used as per following format.

   List. clear ()

   For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
```

```
>>> L1.clear ()
>>> L1
[ ]
```

Unlike del <1stname)> statement, clear () removes only the elements and not the list element. After clear (), the list object still exists as an empty list.

8. The count method:- This function returns the count of the item that you passed as argument. If the given item is not in the list, it returns zero. It is used as per following format:

   List. count (<item>)

   For instance, for a list L1 = [13, 18, 20, 10, 18, 23]

```
>>> L1.count (18)
2
>>> L1.count (28)
0
```

9. The reverse method:- The reverse() reverses the items of the list. This is done "in place", i.e, it does not create a new list. The syntax to use reverse method is:

   List.reverse ()

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

For example,

```
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.reverse ()
>>> t1
['p', 'q', 'a', 'q', 'i', 'e']
>>> t2 = [3, 4, 5]
>>> t3 = t2. reverse ()
>>> t3
>>> t3
[5, 4, 3]
```

10. The sort method:- The sort () function sorts the items of the list, by default in increasing order. this is done "in place", i.e it does not create a new list. It is used as per following syntax:

List.sort ()

For example,

```
>>> t1 = ['e', 'i', 'q', 'a', 'q', 'p']
>>> t1.sort ()
>>> t1
['a', 'e', 'i', 'p', 'q', 'q']
```

Like reverse (), sort () also performs its function and does not return anything. To sort a list in decreasing order using sort (), you can write:

```
>>> List.sort (reverse = True)
```

Quick Interesting Facts:

- The + operator adds one List to the end of another. The * operator repeats a List.
- List slice is an extracted part of List; List slice is a List in itself.
- Common List manipulation functions are: len (), index (), and List ().

PERIOD-07

Writing Programs in Lists

Program: - Program to find minimum element from a list of element along with its index in the list

```
1st = eval (input ("Enter list: "))
length = len (1st)
min_ele = 1st [0]
min_index = 0
for i in range (1, length) :
    if 1st [i] < min_ele :
    min_ele = 1st [i]
min_index = i
print ("Given list is :", 1st)
print ("The minimum element of the given list is :")
print (min_ele, "at index", min_index)
```

Output:

```
Enter list : [2, 3, 4, -2, 6, -7, 8, 11, -9, 11]
Given list is : [2, 3, 4, -2, 6, -7, 8, 11, -9, 11]
The minimum element of the given list is:
-9 at index 8
```

Program:- Program to calculate mean of a given list of numbers.

```
1st = eval (input ("Enter list : "))
length = len (1st)
mean = sum = 0
for i in range (0, length) :
    sum + 1st [i]
mean = sum / length
print ("Given list is : ", 1st)
print ("The mean of the given list is :", mean)
```

Output:

```
Enter list : [7, 23, -11, 55, 13, 5, 20.05, -5.5]
Given list is : [7, 23, -11, 55, 13, 5, 20.05, -5.5]
The mean of the given list is : 14.578571428571427
```

Program:- Program to search for an element in a given list of numbers.

```
1st = eval (input ("Enter list:"))
length = len (1st)
element = int (input ("Enter element to be searched for :"))
for i in range (0, length) :
if element == 1st [i] :
    print (element, "found at index",i)
    break
else :                  # else of for loop
    print (element, "not found in given list")
```

Two sample runs of above program are being given below:

```
Enter list : [2, 8, 9, 11, -55, -11, 22, 78, 67]
Enter element to be searched for : -11
```

```
-11 found at index 5
Enter list : [2, 8, 9, 11, -55, -11, 22, 78, 67]
Enter element to be searched for : -22
-22 not found in given list
```

Program:- Program to count frequency of a given element in a list of numbers.

```
1st = eval (input ("Enter list : "))
length = len (1st)
element = int (input ("Enter element :"))
count = 0
for i in range (0, length) :
    if element == 1st [i] :
            count + =1
if count == 0
    print (element, "not found in given list")
else:
    print (element, "has frequency as", count, "in given list")
```

Sample run of above program is given below:

```
Enter list: [1, 1, 1, 2, 2, 3, 4, 2, 2, 5, 5, 2, 2, 5]
Enter element: 2
2 has frequency as 6 in given list
```

Program: - Program to find frequencies of all elements of a list. Also, print the lsit of unique elements in the list and duplicate elements in the given list.

```
1st eval (input ("Enter list : "))
length = len (1st)
uniq = []              # list to hold unique elements
dupl = []              # list to hold duplicate elements
count = i = 0
while i < length :
    element = 1st [i]
    count = 1          # count as 1 for the element at 1st [i]
if element not in uniq and element not in dupl :
    i + 1
```

```
    for j in range (i, length) :
    if element = = 1st [j] :
            count + = 1
else :              # when inner loop - for loop ends
    print ("Element", element, "frequency:", count)
    if count == 1:
    uniq . append (element)
else:
    dupl. append (element)
else :              # when element is found in uniq or dupl lists
    i + 1
print ("original list", list)
print ("Unique elements list", uniq)
print ("Duplicates elements list", dupl)
```

Output:

```
Enter list : [2, 3, 4, 5, 3, 6, 7, 3, 5, 2, 7, 1, 9, 2]
Element 2 frequency : 3
Element 3 frequency : 3
Element 4 frequency : 1
Element 5 frequency : 2
Element 6 frequency : 1
Element 7 frequency : 2
Element 1 frequency : 1
element 9 frequency : 1
original list [2, 3, 4, 5, 3, 6, 7, 3, 5, 2, 7, 1, 9, 2]
Unique element list [4, 6, 1, 9]
Duplicates elements list [2, 3, 5, 7]
```

LIST ERROR DETECTION

01. What is the difference between following two expressions, if 1st is given as [1, 3, 5]

    (a) 1st * 3 (b0 1st * = 3

02. Given two lists

L1 = ["this", 'is', 'a', 'List'], L2 = ["this", ["is", "another"], "List"]

Which of the following expressions will cause an error and why?

(a) L1 = = L2 (b) L1 . upper () (c) L1[3] . upper()

(c) L2 . upper () (d) L2 [1] . upper () (f) L2 [1] [1] . upper ()

03. From the previous question, give output of expressions that do not result in error.

04. Give a list L1 = [3, 45, 12, 25.7, [2, 1, 0, 5], 88]

(a) Which list slice will return [12, 25.7, [2, 1, 0, 5]]

(b) Which expression will return [2, 1, 0, 5]

(c) Which list slice will return [[2, 1, 0, 5]]

(d) Which list slice will return [4.5, 25.7, 88]

05. Give a list L1 = [3, 4.5, 12, 25.7,] 2, 1, 0, 5], 88] which function can change the list to:

(a) [3, 4.5, 12, 25.7, 88] (b) [3, 4.5, 12, 25.7] (c) [[2, 1, 0, 5], 88]

06. What will the following code result in?

```
L1 = [1, 3, 5, 7, 9]
print (L1 ==L1. reverse())
print (L1)
```

07. Predict the output

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
my_list [2:3] = [ ]
print (my _ list)
my_list [2:5] = []
print (my_list]
```

8. Predict the output

```
List1 = [13, 18, 11, 16, 13, 18, 13]
print (List1.index(18))
print(List1.count(18))
List1.append(List1.count(13))
print(List1)
```

9. Predict the output

```
odd = [1, 3, 5]
print ( (odd + [2, 4, 6]) [4])
print ( (odd + [12, 14, 16]) [4] - (odd + [2, 4, 6]) [4] )
```

10. Predict the output

```
a, b, c = [1, 2], [1, 2], [1, 2]
print (a = = b)
print (a is b)
```

11. Predict the output of following two parts. Are the outputs same? Are the outputs different? Why?

(a)
```
L1, L2 = [2, 4], [2, 4]
L3 = L2
L2[1] = 5
print (L3)
```

(b)
```
L1, L2 = [2, 4], [2, 4]
L3 = list [L2]
L2[1] = 5
print (L3)
```

12. Find the errors

```
1. L1 = [1, 11, 21, 31]
2. L2 = L1 + 2
3. L3 = L1 * 2
4. Idx = L1.index (45)
```

13. Find the errors

(a)
```
L1 = [1, 11, 21, 31]
An = L1.remove (41)
```

(b)
```
L1 = [1, 11, 21, 31]
An = L1.remove (31)
print (An + 2)
```

14. Find the errors

(a)
```
L1 = [3, 4, 5]
L1 = L1 * 3
print (L1 * 3.0)
print (L2)
```

(b)
```
L1 = [3, 3, 8, 1, 3, 0, '1', '0', '2', 'e', 'w', 'e', 'r']
print (L1 [: : -1])
print ( L1 [-1 : -2 : -3] )
```

```
print ( L1 [-1 : -2 : -3 : -4] )
```

15. What will be the output of following code?

```
x = ['3', '2', '5']
y = ''
while x :
    y = y + x [-1]
    x = x [: len (x) -1]
print (y)
print (x)
print (type (x), type (y))
```

Quick Interesting Facts:

- A list is an ordered and mutable Python container, being one of the most common data structures in Python.
- To create a list, the elements are placed inside square brackets ([]), separated by commas.
- lists can contain elements of different types as well as duplicated elements

****