Chapter 2

Lists

After studying this lesson, students will be able to:

- *Understand the concept of mutable sequence types in Python.*
- *Appreciate the use of list to conveniently store a large amount of data in memory.*
- *Create, access & manipulate list objects*
- *Use various functions & methods to work with list*
- *Appreciate the use of index for accessing an element from a sequence.*

Introduction

Like a String, list also is sequence data type. It is an ordered set of values enclosed in square brackets []. Values in the list can be modified, i.e. it is mutable. As it is set of values, we can use index in square brackets [] to identify a value belonging to it. The values that make up a list are called its elements, and they can be of any type.

We can also say that list data type is a container that holds a number of elements in a given order. For accessing an element of the list, indexing is used.

Its syntax is:

Variable name [index] (variable name is name of the list).

It will provide the value at 'index+1' in the list. Index here, has to be an integer value-which can be positive or negative. Positive value of index means counting forward from beginning of the list and negative value means counting backward from end of the list. Remember the result of indexing a list is the value of type accessed from the list.

Index value	**Element of the list**
0, -size	1st
1, -size +1	2nd

2, -size +2	3rd
. . .	
size -2, -2	2nd last
size -1, -1	last

Please note that in the above example size is the total number of elements in the list.

Let's look at some example of simple list:

```
i)    >>>L1 = [1, 2, 3, 4]                # list of 4 integer elements.
ii)   >>>L2 = ["Delhi", "Chennai", "Mumbai"] #list of 3 string elements.
iii)  >>>L3 = [ ]                         # empty list i.e. list with no element
iv)   >>>L4 = ["abc", 10, 20]             # list with different types of elements
v)    >>>L5 = [1, 2, [6, 7, 8], 3]        # A list containing another list known as
      nested list
```

You will study about Nested lists in later parts of the chapter.

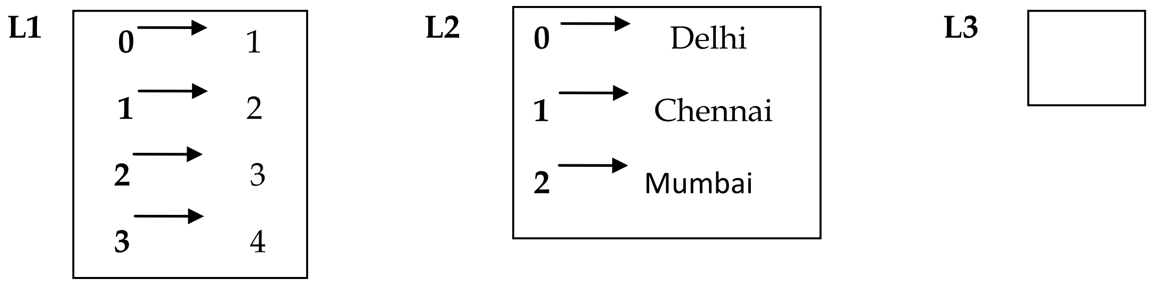
To change the value of element of list, we access the element & assign the new value.

Example

```
>>>print L1          # let's get the values of list before change
>>> L1 [2] = 5
>>> print L1         # modified list
 [1, 2, 5, 4]
```

Here, 3rd element of the list (accessed using index value **2**) is given a new value, so instead of **3** it will be **5.**

State diagram for the list looks like:



Note: List index works the same way as String index, which is:

- ✡ An integer value/expression can be used as index.
- ✡ An Index Error appears, if you try and access element that does not exist in the list.
- ✡ An index can have a negative value, in that case counting happens from the end of the list.

Creating a list

List can be created in many ways:

i) By enclosing elements in [ ], as we have done in above examples.

ii) Using other Lists

Example

L5=L1 [:]

Here L5 is created as a copy of L1.

```
>>>print L5
L6 = L1 [0:2]
>>>print L6
```

will create L6 having first two elements of L1.

iii) List comprehension

Example

```
>>>n = 5
```

```
>>>l = range(n)

>>>print l

[0, 1, 2, 3, 4]
```

Example

```
>>> S= [x**2 for x in range (10)]

>>> print S

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In mathematical terms, S can be defined as S = {x^2 for: x in (0.....9)}. So, we can say that list comprehension is short-hand for creating list.

Example

```
>>> A = [3, 4, 5]

>>> B = [value *3 for value in A]
```

Here B will be created with the help of A and its each element will be thrice of element of A.

```
>>> print B

[9, 12, 15]
```

Comprehensions are functionally equivalent to wrting as:

```
>>>B = [ ]

>>>for i in A

B. append (i*3)
```

Similarly, other comprehensions can be expended.

Example

```
>>> print B

[9, 12, 15]
```

Let's create a list of even numbers belonging to '**S**' list:

```
>>>C = [i for i in S if i % 2 = = 0]
```

```
>>>print C
[0, 4, 16, 36, 64]
```

iv) Using built-in object

L = list ( ) will create an empty list

Example

```
>>>l = list ( )
>>>print l
[ ] # empty list
```

Or

L = list (sequence)

Example

```
>>>L = list [(1, 2, 3, 4)]
>>>print L
[1, 2, 3, 4]
```

A single new list is created every time, you execute [ ]. We have created many different lists each using [ ]. But if a list is assigned to another variable, a new list is not created.

i) A=B=[ ]

Creates one list mapped to both A & B

Example

```
>>>A = B = [10, 20, 30]
>>> print A, B
[10, 20, 30] [10, 20, 30]
```

ii) A = [ ]

B = A

Will also create one list mapped to both

Example

```
>>> A = [1, 2, 3]
>>> B = A
>>> print A, B
[1, 2, 3] [1, 2, 3]
```

Accessing an element of list

For accessing an element, we use index and we have already seen example doing so. To access an element of list containing another list, we use pair of index. Lets access elements of L5 list. Also a sub-list of list can be accessed using list slice.

List Slices

Slice operator works on list also. We know that a slice of a list is its sub-list. For creating a list slice, we use

[n:m] operator.

```
>>>print L5 [0]
1
>>>print L5 [2]
[6, 7, 8]
```

as the 3rd element of this list is a list. To access a value from this sub-list, we will use

```
>>>print L5 [2] [0]
6
>>>print L5 [2] [2]
8
```

This will return the part of the list from nth element to mth element, including the first element but excluding the last element. So the resultant list will have m-n elements in it.

```
>>> L1 [1:2]
```

will give

```
[2]
```

Slices are treated as boundaries, and the result will contain all the elements between boundaries.

Its Syntax is:

seq = L [start: stop: step]

Where start, stop & step- all three are optional. If you omit first index, slice starts from '0' and omitting of stop will take it to end. Default value of step is 1.

Example

For list L2 containing ["Delhi", "Chennai", "Mumbai"]

```
>>>L2 [0:2]

["Delhi", "Chennai"]
```

Example

```
>>>list = [10, 20, 30, 40, 50, 60]

>>> list [::2]    # produce a list with every alternate element

[10, 30, 50]

>>>list [4:]      # will produce a list containing all the elements from 5th position
                  till end

[50, 60]
```

Example

```
>>>list [:3]

[10, 20, 30]

>>>list [:]

[10, 20, 30, 40, 50, 60]
```

Example

```
>>> list [-1]    # '-1' refers to last elements of list

60
```

will produce a list with every other element

Note: Since lists are mutable, it is often recommended to make a copy of it before performing operation that change a list.

Traversing a List

Let us visit each element (traverse the list) of the list to display them on screen. This can be done in many ways:

(i)
```
i = 0
while i < 4:
print L1 [i],
i + = 1
```
will produce following output
```
1 2 5 4
```

(ii)
```
for i in L1:
print i,
```
will also produce the same output

(iii)
```
i=0
while i < len [L1]:
print L1 [i],
i + = 1
```
OR
```
i= 0
L = len (L1)
while i < L :
print L1 [i],
i + = 1
```
will also produce the same output.

Here len( ) function is used to get the length of list L1. As length of L1 is 4, i will take value from 0 to 3.

(iv) for i in range (len (L1)):

print L1 [i],

Using 2nd way for transversal will only allow us to print the list, but other ways can also be used to write or update the element of the list.

In 4th way, range ( ) function is used to generate, indices from 0 to len -1; with each iteration i gets the index of next element and values of list are printed.

Note: for loop in empty list is never executed:

Example

```
for i in [ ]:
    print i
    Accessing list with negative index
    i = 1
    while i < len (L1):
    print L1 [-i],
    i += 1
```

In this case, Python will add the length of the list to index and then return the index value and accesses the desired element. In this loop execution for a positive value of 'i' L1 [-i] will result into L1 [len (L1)-i] for i=1, L1 [4-1] will be printed. So resultant of the loop will be 4 5 2.

Appending in the list

Appending a list is adding more element(s) at the end of the list. To add new elements at the end of the list, Python provides a method append ( ).

Its Syntax is:

List. append (item)

L1. append (70)

This will add 70 to the list at the end, so now 70 will be the 5th element of the list, as it already have 4 elements.

```
>>> print L1
```

will produce following on screen

```
[1, 2, 5, 4, 70]
```

Example

```
>>>L4.append (30)          # will add 30 at the end of the list
>>>print L4
['abc', 10, 20, 30]
```

Using append ( ), only one element at a time can be added. For adding more than one element, extend ( ) method can be used, this can also be used to add elements of another list to the existing one.

Example

```
>>>A = [100, 90, 80, 50]
>>> L1. extend (A)
>>> print L1
```

will add all the elements of list 'A' at the end of the list 'L1'.

```
[1, 2, 5, 4, 70, 100, 90, 80, 50]
>>>print A
[100, 90, 80, 50]
```

Example

```
>>>B=[2009, 2011, 'abc']
>>>C=['xyz', 'pqr', 'mn']
>>>B.extend (c)
>>>print B
```

```
[2009, 2011, 'abc', 'xyz', 'pqr', 'mn']
```

Remember: 'A' remains unchanged

Updating array elements

Updating an element of list is, accomplished by accessing the element & modifying its value in place. It is possible to modify a single element or a part of list. For first type, we use index to access single element and for second type, list slice is used. We have seen examples of updations of an element of list. Lets update a slice.

Example

```
>>> L1 [1:2] = [10, 20]

>>> print L1

will produce

[1, 10, 20, 4, 70, 100, 90, 80, 50]
```

Example

```
>>>A=[10, 20, 30, 40]

>>>A [1:4] = [100]

>>>print A

will produce

[10, 100]
```

As lists are sequences, they support many operations of strings. For example, operator **+** & * results in concatenation & repetition of lists. Use of these operators generate a new list.

Example

```
>>> a= L1+L2
```

will produce a 3rd list **a** containing elements from L1 & then L2. **a** will contain

```
[1, 10, 20, 4, 70, 100, 90, 80, 50, "Delhi", "Chennai", "Mumbai"]
```

Example

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

Example

```
>>> b = L1*2
>>> print b
[[1, 10, 20, 4, 70, 100, 90, 80, 50, 1, 10, 20, 4, 70, 100, 90, 80, 50]
```

Example

```
>>> ['Hi!']* 3
['Hi!', 'Hi!', 'Hi!']
```

It is important to know that **'+'** operator in lists expects the same type of sequence on both the sides otherwise you get a type error.

If you want to concatenate a list and string, either you have to convert the list to string or string to list**.**

Example

```
>>> str([11, 12]) + "34"    or  >>>"[11,12]" + "34"
'[11, 12] 34'
>>> [11, 12] + list ("34")    or >>>[11, 12] + ["3", "4"]
  [11, 12, '3', '4']
```

Deleting Elements

It is possible to delete/remove element(s) from the list. There are many ways of doing so:

(i) If index is known, we can use pop ( ) or del

(ii) If the element is known, not the index, remove ( ) can be used.

(iii) To remove more than one element, del ( ) with list slice can be used.

(iv) Using assignment operator

Let us study all the above methods in details:

Pop ( )

It removes the element from the specified index, and also return the element which was removed.

Its syntax is:

List.pop ([index])

Example

```
>>> L1 = [1, 2, 5, 4, 70, 10, 90, 80, 50]
>>> a= L1.pop (1)          # here the element deleted will be returned to 'a'
>>> print L1
[1, 5, 4, 70, 10, 90, 80, 50]
>>> print a
2
```

If no index value is provided in pop ( ), then last element is deleted.

```
>>>L1.pop ( )
50
```

del removes the specified element from the list, but does not return the deleted value.

```
>>> del L1 [4]
>>> print L1
[1, 5, 4, 70, 90, 80]
```

remove ( )

In case, we know the element to be deleted not the index, of the element, then remove ( ) can be used.

```
>>> L1. remove (90)
```

will remove the value 90 from the list

```
>>> print L1
[1, 5, 4, 70, 80]
```

del () with slicing

Consider the following example:

Examples

```
>>> del L1 [2:4]
>>>print L1
[1, 5, 80]
```

will remove 2nd and 3rd element from the list. As we know that slice selects all the elements up to 2nd index but not the 2nd index element. So **4th element** will remain in the list.

```
>>> L5 [1:2] = [ ]
```

Will delete the slice

```
>>>print L5
[1, [6, 7, 8], 3]
```

Note:

(i) All the methods, modify the list, after deletions.

(ii) If an out of range index is provided with del ( ) and pop ( ), the code will result in to run-time error.

(iii) del can be used with negative index value also.

Other functions & methods

insert ( )

This method allows us to insert an element, at the given position specified by its index, and the remaining elements are shifted to accommodate the new element. Insert (

() requires two arguments-**index value** and **item value**.

Its syntax is

list. insert (index, item)

Index specifies the position (starting from 0) where the element is to be inserted. Item is the element to be inserted in the list. Length of list changes after insert operation.

Example

```
>>> L1.insert (3,100)
>>>print L1
will produce
[1, 5, 80, 100]
```

Note: If the index specified is greater then len (list) the object is inserted in the last and if index is less than zero, the object is inserted at the beginning.

```
>>> print len(L1)
4
>>> L1.insert (6, 29)
>>> L1.insert (-2, 46)
>>>print L1
will produce
[46, 1, 5, 80, 100, 29]
```

reverse ( )

This method can be used to reverse the elements of the list in place

Its syntax is:

list.reverse ( )

Method does not return anything as the reversed list is stored in the same variable.

Example

```
>>> L1.reverse ( )
>>> print L1
will produce
[29, 100, 80, 5, 1, 46]
```

Following will also result into reversed list.

```
>>>L1 [: : -1]
```

As this slices the whole sequence with the step of -1 i.e. in reverse order.

sort ( )

For arranging elements in an order Python provides a method **sort ( )** and a function **sorted ( )**. sort ( ) modifies the list in place and sorted ( ) returns a new sorted list.

Its Syntax are:

sort ([cmp [, key [, reverse]]])

sorted (list [, cmp [, key [, reverse]]])

Parameters mentioned in [ ] are optional in both the cases. These parameters allow us to customize the function/method.

cmp, argument allow us to override the default way of comparing elements of list. By default, sort determines the order of elements by comparing the elements in the list against each other. To overside this, we can use a user defined function which should take two values and return -1 for **'less than'**, 0 for **'equal to'** and 1 for 'greater than'.

'**Key'** argument is preferred over **'cmp'** as it produces list faster.

Example

The parameter **'key'** is for specifying a function that transforms each element of list before comparison. We can use predefined functions or a user defined function here. If its user defined then, the function should take a single argument and return a key which can be used for sorting purpose.

Reverse parameter can have a boolean value which is used to specify the order of arranging the elements of list. Value **'True'** for reverse will arrange the elements of list in descending order and value **'False'** for reverse will arrange the elements in ascending order. Default value of this parameter is False.

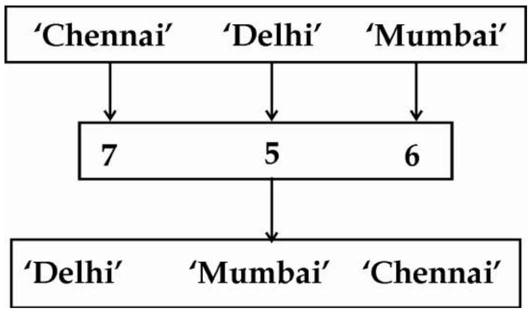
sorted ( ) function also behaves in similar manner except for it produce a new sorted list, so original is not changed. This function can also be used to sort any iterable collection. As sort ( ) method does not create a new list so it can be little faster.

Example

```
>>> L1.sort ( )

>>> print L1

will produce

[1, 5, 29, 46, 80, 100]

>>> L2.sort ( )

>>> print L2

will produce

['Chennai', 'Delhi', 'Mumbai']

>>> L2.sort (key=len)

will produce

['Delhi', 'Mumbai', 'Chennai']
```

Here we have specified len ( ) built in function, as key for sorting. So the list will get sorted by the length of the strings, i.e., from shorted to longest.

sort will call len ( ) function for each element of list and then these lengths will be used for arranging elements.



```
>>> L4.sort ( )

>>> print L4

will produce
```

```
[10, 20, 30, 'abc']
>>>L4.sort (reverse = True)
['abc', 30, 20, 10]
>>> def compare (str):
...         return len (str)
>>> L2.sort (key=compare)
>>> L2
['Delhi', 'Mumbai', 'Chennai']
```

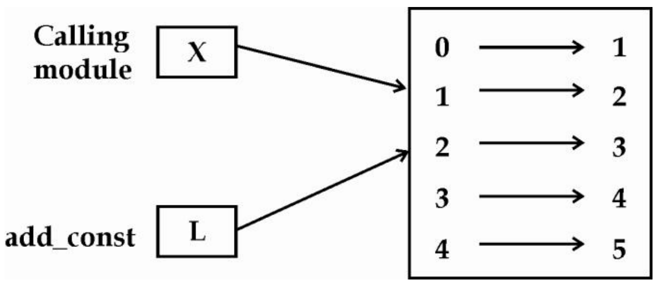
List as arguments

When a list is passed to the function, the function gets a reference to the list. So if the function makes any changes in the list, they will be reflected back in the list.

Example

```
def add_Const (L):
for i in range (len (l)):
L [i] += 10
>>> X = [1, 2, 3, 4, 5]
>>> add_Const (X)
>>> print X
[11, 12, 13, 14, 15]
```

Here parameter 'L' and argument 'X' are alias for same object. Its state diagram will look like

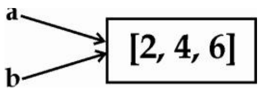


So any changes made in L will be reflected to X as lists as mutable.

Note: Here, it becomes important to distinguish between the operations which modifies a list and operation which creates a new list. Operations which create a new list will not affect the original (argument) list.

Let's look at some examples to see when we have different lists and when an alias is created.

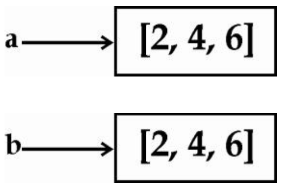
```
>>> a = [2, 4, 6]

>>> b = a
```



will map b to a. To check whether two variables refer to same object (i.e. having same value), we can use 'is' operator. So in our example:

```
>>> a is b

will return 'True'

>>> a = [2, 4, 6]

>>> b = [2, 4, 6]

>>> a is b
```

False



In first example, Python created one list, reference by a & b. So there are two references to the same object b. We can say that object [2, 4, 6] is aliased as it has more than one name, and since lists are mutable. So changes made using 'a' will affect 'b'.

```
>>> a [1] = 10

>>> print b

will print

[2, 10, 6]
```