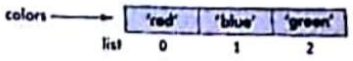
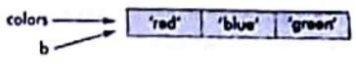
2.3.5 Making True Copy of a List

Assignment with an assignment operator (=) on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory (called *shallow copy*).

```
colors = ['red', 'blue', 'green']
```



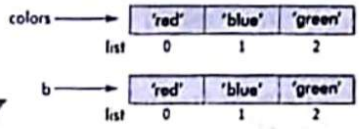

```
b = colors    ## Does not copy the list
```



So, if you make changes in one list, the other list will also report those changes because these two list-names are labels referring to same list because '=' copied the reference not the actual list.

To make b true copy of list **colors** *i.e.*, an independent list identical to list **colors** you should create copy of list as follows :

```
b = list(colors)
```



Now *colors* and *b* are separate lists (*deep copy*).

2.3.6 List Functions

Python also offers many built-in functions and methods for list manipulation. These can be applied to list as per following syntax :

```
<listObject>.<method name>()
```

1. The index method

This function returns the index of first matched item from the list.

```
List.index (<item>)
```

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

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

returns the index of first value 18, even if there is another value 18 at index 4.

However, if the given item is not in the list, it raises exception value Error

2. The append method

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']
```

See the item got added at the end of the list

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

3. The extend method

The *extend( )* method is also used for adding multiple elements (given in the form of a list) to a list.

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 the following example :

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

Extend the list t1, by adding all elements of t2

See the elements of list t2 are added at the end of list t1

Difference between append( ) and extend( ) methods

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

4. The insert method

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 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']
```

See element 'i' inserted at index 2

5. The pop method

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()
>>> ele2
'u'
>>> t1
['a', 'e', 'i', 'p', 'q']
```

Remove element at index 0 i.e., first element and store it in ele1

The removed element

List after removing first element

No index specified, it will remove the last element

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

6. 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', 'q', 'a', 'q', 'p']
>>> t1.remove('p')
>>> t1
['e', 'i', 'q', 'a', 'q', 'p']
```

First occurrence of 'a' is removed from the list

First occurrence of 'p' is removed from the 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 is used as per following format.

```
List.clear( )
```

For instance, if you have a list L1 as

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 is used as per following format.

```
List.clear( )
```

For instance, if you have a list L1 as

```
>>> L1 = [2, 3, 4, 5]
>>> L1.clear( )        <-- it will remove all the items from list L1.
>>> L1
[ ]                    <-- Now the L1 is an empty list.
```

Unlike **del <lstname>** 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 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    <-- returns 2 as there are two items with value 18 in the list.
>>> L1.count(28)
0    <-- No item with value 28 in the list, hence it returned 0 (zero)
```

9. The reverse method

The *reverse( )* reverses the items of the list. This is done **"in place"**, *i.e.*, it does not create a n 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          <-- The reversed list
['p', 'q', 'a', 'q', 'i', 'e']
```

10. The sort method

The *sort( )* function sorts the items of the list, by default in increasing order. This is done " **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          <-- Sorted list in default ascending order
['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)
```