

Contents

1. Introduction to Lists in Python:	2
2. Creating a list in Python.....	2
a) Creating a list in Python by using square brackets '['	3
b) Creating a list in Python by using the list() function	3
3. Difference between list() function and list class in python:.....	4
4. Accessing List in Python:	5
a) Accessing List By using indexing in Python:	5
b) Accessing List By using slicing operator in Python.....	6
c) Accessing List By using loops in Python:	7
5. Important Functions And Methods of List In Python:	8
5.1. Method or Function to get size of list.	8
d) len() function in python:	8
e) count() method in Python:.....	8
5.2. Method or Function to add items in list.	9
a) append() method in Python:.....	9
b) insert() method in Python:.....	9
c) extend(list[]) method in Python:	10
5.3. Function and Method to remove items from list.	11
a) remove(x) method in Python:.....	11
b) pop(i) method in Python:.....	12
c) Difference between remove() and pop()	13
5.4. Ordering the elements in list:	13
5.5. LIST ALIASING and CLONING in PYTHON:	15
a) Aliasing List in Python:	15
b) Cloning in List:	16
6. MATHEMATICAL OPERATORS on LIST	17
a) The concatenation operator (+):.....	17
b) Multiplication operator in Lists:.....	18
7. COMPARISON of LISTS in Python	18
8. Membership Operators In List:	20

9.	List Comprehensions In Python:.....	20
	Example 2: <i>List Comprehensions (demo37.py): Conditional Statements in List Comprehension ..</i>	22
	Why Use List Comprehension?.....	22
10.	Nested Lists In Python (2D List):	23

1. Introduction to Lists in Python:

- A **list** in Python is a collection of **ordered, mutable (changeable), and heterogeneous** elements (can contain numbers, strings, other lists, etc.).
- A list is a **data structure** that can store a group of elements or objects. It has the following **features/properties**:
 - 1) Lists can store the **same type as well as different types** of elements.
Hence, it is **heterogeneous**.
 - 2) Lists have **dynamic nature** which means we **don't need to mention the size of the list** while declaring as we do for arrays in other programming languages. The size will increase dynamically.
 - 3) **Insertion order is preserved**. The order, in which the items are added to a list, is the order in which output is displayed.
 - 4) The list **can store duplicates elements**.
 - 5) In lists, the **index plays** the main role. Using index we perform almost all the operations on the items in it.
 - 6) List **objects are mutable** which means we **can modify or change** the content of the list even after the creation.

2. Creating a list in Python

There are a couple of ways to create a list in Python

- a) **Creating a list in Python by using square brackets '['']**
- b) **Creating a list in Python by using the list() function**

a) Creating a list in Python by using square brackets '[]'

We can create a list in python by using **square brackets** '[]'. The elements in the list should be comma-separated.

Example: Creating an empty list (demo1.py)

```
l1 = []  
print(l1)  
print(type(l1))
```

Example: Creating a list with elements (demo2.py)

```
names = ["Mohan", "Prasad", "Ramesh", "Mohan", 10, 20, True, None]  
print(names)
```

Output: ['Mohan', 'Prasad', 'Ramesh', 'Mohan', 10, 20, True, None]

We can observe from the output that the **order is preserved** and also **duplicate items are allowed**. It can contain **mixed data** types (strings, integers, boolean, None) elements.

b) Creating a list in Python by using the list() function

We can create a list by using the **list() function** in python. Generally, we use this function for creating **lists from other data types** like range, tuple, etc. This is generally referred to as **type conversion**

Example: Creating a list using list() function (demo3.py)

```
r=range(0, 10)  
l=list(r)  
print(l)
```

Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

3. Difference between list() function and list class in python:

- list() is a **predefined function** in python. list() function is used to **convert other** sequences or data types **into list data** types.
- The list is a **predefined class** in python. Once we create a list then it is internally represented as a list class object type. We can check this by using a type function.

Example: Creating a list using list() function (demo3.py)

```
r=range(0, 10)
l=list(r)
print(l)
print(type(l))
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
<class 'list'>
```

Example: Mutability (demo4.py)

```
l = [1, 2, 3, 4, 5]
print(l)
print("Before modifying l[0] : ",l[0])
l[0]=20
print("After modifying l[0] : ",l[0])
print(l)
```

Output:

```
[1, 2, 3, 4, 5]
Before modifying l[0] : 1
After modifying l[0] : 20
[20, 2, 3, 4, 5]
```

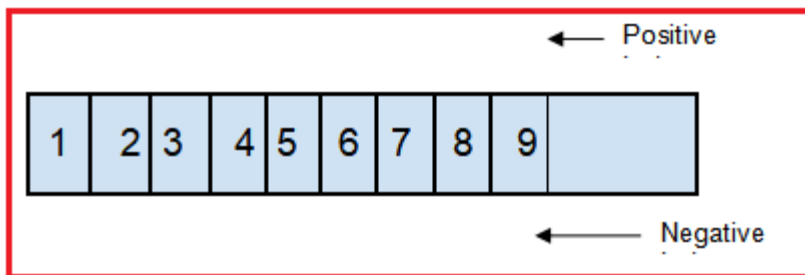
4. Accessing List in Python:

There are **three ways** in which you can access the elements in the list in python. They are:

- a) **By using indexing**
- b) **By using the slicing operator**
- c) **By using loops**

a) Accessing List By using indexing in Python:

Accessing the elements by index means, accessing by their position numbers in the list. Index starts from 0 onwards. List also, just as strings, supports both positive and negative indexes. The positive index represents from left to right direction and the negative index represents from right to left direction.



Note: IndexError: If we are trying to access beyond the range of list index, then we will get IndexError.

Example: List Indexing (demo5.py)

```
names = ["Prabhas", "Prashanth", "Prakash"]  
  
print(names)  
  
print(names[0])  
  
print(names[1])  
  
print(names[2])  
  
print(type(names))  
  
print(type(names[0]))  
  
print(type(names[1]))
```

```
print(type(names[2]))
```

Output:

```
['Prabhas', 'Prashanth', 'Prakash']
Prabhas
Prashanth
Prakash
<class 'list'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Example: List index out of range error (Demo6.py)

```
names = ["Prabhas", "Prashanth", "Prakash"]

print(names)

print(names[0])

print(names[1])

print(names[2])

print(names[10])
```

Output:

```
['Prabhas', 'Prashanth', 'Prakash']
Prabhas
Prashanth
Prakash
Traceback (most recent call last):
  File "/home/ruhan/python_course/7. Input and Output/demo12.py", line 6, in <module>
    print(names[10])
IndexError: list index out of range
```

b) Accessing List By using slicing operator in Python

Slicing is the **way of extracting a sublist** from the main list. The syntax is given below.

list_name[start: stop: stepsize]

- **start represents** the **index** from where we slice **should start**, the default value is **0**.

- **stop represents** the end **index** where the **slice should end**. The max value allowed is the length of the list. **If no value** is provided, then it takes the **last index** of the list as the default value.
- **stepsize** represents the **increment in the index** position of the two consecutive elements to be sliced.
- The result of the slicing operation on the list is **also a list** i.e it returns a list.

Example: Slicing (Demo7.py)

```
n = [1, 2, 3, 4, 5, 6]
print(n)
print(n[2:5:2])
print(n[4::2])
print(n[3:5])
```

Output:

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

c) Accessing List By using loops in Python:

We can also access the elements of a list **using for** and **while** loops

Example: Accessing list using loops (demo8.py)

```
a = [100, 200, 300, 400]
for x in a:
    print(x)
```

Output:

```
100
200
300
400
```

Example: Accessing list using loops (demo9.py)

```
a = [100, 200, 300, 400]
```

```
x = 0
while x < len(a):
    print(a[x])
    x = x + 1
```

Output:

```
100
200
300
400
```

5. Important Functions And Methods of List In Python:

In python, a **method** and **functions** are almost same. Only the difference is their use-case:

- **Function** → A **built-in global operation** that works on data structures (lists, tuples, strings, etc.). Eg. *function_name(list_object)*
- **Method** → A function that **belongs to an object** and is called using the dot . operator. Eg. *list_object.method_name(arguments)*

5.1. Method or Function to get size of list.

d) len() function in python:

This function when applied to a list **will return the length** of the elements in it.

Example: len() function (demo10.py)

```
n = [1, 2, 3, 4, 5]
print(len(n))
```

Output: 5

e) count() method in Python:

This method returns the **number of occurrences** of a specific item in the list

Example: count() method (demo11.py)

```
n = [1, 2, 3, 4, 5, 5, 5, 3]
```

```
print(n.count(5))  
print(n.count(3))  
print(n.count(2))
```

Output:

```
3  
2  
1
```

5.2. Method or Function to add items in list.

a) append() method in Python:

Using this method we **can add elements to the list**. The items or elements will be added at the end of the list.

Example: append() method (demo12.py)

```
l=[]  
l.append("Ramesh")  
l.append("Suresh")  
l.append("Naresh")  
print(l)
```

Output: ['Ramesh', 'Suresh', 'Naresh']

b) insert() method in Python:

We can add elements to the list object by using the insert() method. The insert() method takes two arguments as input: one is the index and the other is the element. It will add the elements to the list at the specified index.

Example: insert() method (demo13.py)

```
n=[10, 20, 30, 40, 50]  
n.insert(0, 76)  
print(n)
```

Output: [76, 10, 20, 30, 40, 50]

Difference between append and insert in Python

Both methods are used to add elements to the list. But the difference between them is that the **append** will add the elements to the last, whereas the insert will add the elements at the specified index mentioned.

While we are using the insert() method,

1. If the specified index is **greater than the max index**, then the element will be inserted **at the last** position.
2. If the specified index is **smaller than the min index**, then the element will be inserted **at the first position**.

Example: insert() method (demo14.py)

```
l=[10, 20, 30, 40]

print(l) # [10, 20, 30, 40]

l.insert(1, 111)

print(l) # [10, 111, 20, 30, 40]

l.insert(-1, 222)

print(l) # [10, 111, 20, 30, 222, 40]

l.insert(10, 333)

print(l) # [10, 111, 20, 30, 222, 40, 333]

l.insert(-10, 444)

print(l) # [444, 10, 111, 20, 30, 222, 40, 333]
```

Output:

```
[10, 20, 30, 40]
[10, 111, 20, 30, 40]
[10, 111, 20, 30, 222, 40]
[10, 111, 20, 30, 222, 40, 333]
[444, 10, 111, 20, 30, 222, 40, 333]
```

c) extend(list[]) method in Python:

Using this method the **items in one list** can be **added to the other**.

Example: extend() method (demo15.py)

```
l1 = [1,2,3]

l2 = ['Rahul', 'Rakesh', 'Regina']

print('Before extend l1 is:', l1)

print('Before extend l2 is:', l2)

l2.extend(l1)

print('After extend l1 is:', l1)

print('After extend l2 is:', l2)
```

Output:

```
Before extend l1 is: [1, 2, 3]
Before extend l2 is: ['Rahul', 'Rakesh', 'Regina']
After extend l1 is: [1, 2, 3]
After extend l2 is: ['Rahul', 'Rakesh', 'Regina', 1, 2, 3]
```

Example: extend() method (demo16.py)

```
august_txns = [100, 200, 500, 600, 400, 500, 900]

sept_txns = [111, 222, 333, 600, 790, 100, 200]

print("August month transactions are : ",august_txns)

print("September month transactions are : ",sept_txns)

sept_txns.extend(august_txns)

print("August and Sept total transactions amount: ",sum(sept_txns))
```

Output:

```
August month transactions are : [100, 200, 500, 600, 400, 500, 900]
September month transactions are : [111, 222, 333, 600, 790, 100, 200]
August and Sept total transactions amount: 5556
```

5.3. Function and Method to remove items from list.

a) remove(x) method in Python:

We can use this method to **remove specific items** from the list.

Example: remove() method (demo17.py)

```
n=[1, 2, 3]
n.remove(1)
print(n)
```

Output: [2, 3]

If the **item exists multiple times**, then **only the first occurrence** will be removed. If the specified item **not present in list**, then we will get **ValueError**. Before removing elements it's a good approach to check if the element exists or not to avoid errors.

Example: remove() method (demo18.py)

```
n=[1, 2, 3, 1]
n.remove(1)
print(n)
```

Output: [2, 3, 1]

Example: remove() method (demo19.py)

```
n=[1, 2, 3, 1]
n.remove(10)
print(n)
```

Output: ValueError: list.remove(x): x not in list

b) **pop(i) method in Python:**

- This method takes an **index as an argument** and **removes and returns** the element present at the given index.
- If **no index** is given, then the **last item** is removed and returned by default. If the **provided index is not in the range**, then **IndexError** is thrown.

Example: pop() method (demo20.py)

```
n=[1, 2, 3, 4, 5]
print(n.pop(1))
print(n)
print(n.pop())
print(n)
```

Output:

```
2
[1, 3, 4, 5]
5
[1, 3, 4]
```

Example: pop() method (demo21.py)

```
n=[1, 2, 3, 4, 5]
print(n.pop(10))
```

Output: `IndexError: pop index out of range`

c) Difference between remove() and pop()

remove()	pop()
To remove based on the element.	To remove based on index
It doesn't return any value	It returns the popped out element.
If the item is not present, then we get ValueError	If the index is not in the range, then we get IndexError

5.4. Ordering the elements in list:

a) reverse() method in python:

This method **reverses** the order of list elements.

Example: reverse() method (demo22.py)

```
n=[1, 2, 3, 4, 'two']
print(n)
n.reverse()
print(n)
```

Output:

```
[1, 2, 3, 4, 'two']  
['two', 4, 3, 2, 1]
```

b) **sort() method in Python:**

- In a list by default **insertion order** is preserved.
- If we want to sort the elements of the list according to the **default natural sorting** order then we should go for the sort() method.
- **For numbers**, the default **natural** sorting order is **ascending order**.
- **For strings**, the default **natural** sorting order is **alphabetical order**.
- To use the sort() method, the list should contain **only homogeneous** elements, otherwise, we will get **TypeError**.

Example: sort() method (demo23.py)

```
n=[1, 4, 5, 2, 3]  
  
n.sort()  
  
print(n)  
  
s=['Suresh', 'Ramesh', 'Arjun']  
  
s.sort()  
  
print(s)
```

Output:

```
[1, 2, 3, 4, 5]  
['Arjun', 'Ramesh', 'Suresh']
```

Example: sort() method (demo24.py)

```
n=[1, 4, 5, 2, 3, 'Suresh', 'Ramesh', 'Arjun']  
  
n.sort()  
  
print(n)
```

Output: `TypeError: '<' not supported between instances of 'str' and 'int'`

Shallow Copy vs Deep Copy in Python Lists

5.5. LIST ALIASING and CLONING in PYTHON:

a) Aliasing List in Python:

- The process of **giving a new name to an existing one** is called aliasing.
- The new name is called an alias name. **Both names will refer to the same memory location.**
- Since both the actual and alias name refer to the same location, **any modifications done** on an existing **object will be reflected** in the new one also.

Example: aliasing (demo25.py)

```
x=[10, 20, 30]
```

```
y=x
```

```
print(x)
```

```
print(y)
```

```
print(id(x))
```

```
print(id(y))
```

```
x[1] = 99
```

```
print(x)
```

```
print(y)
```

```
print(id(x))
```

```
print(id(y))
```

Output:

```
[10, 20, 30]
[10, 20, 30]
139943116769288
139943116769288
[10, 99, 30]
[10, 99, 30]
139943116769288
139943116769288
```

b) Cloning in List:

- The process of **creating duplicate independent objects** is called cloning.
- We can implement cloning by using the **slice operator** or by using the **copy() method**.
- These processes create a **duplicate** of the existing one at a **different memory location**.
- Therefore, both objects will be **independent**, and applying any modifications to one will not impact the other.

Example: Cloning using **slicing** operator (demo26.py)

```
x=[10, 20, 30]
y=x[:]
print(x)
print(y)
print(id(x))
print(id(y))
x[1] = 99
print(x)
print(y)
print(id(x))
print(id(y))
```

Output:

```
[10, 20, 30]
[10, 20, 30]
139678909347848
139678909362696
[10, 99, 30]
[10, 20, 30]
139678909347848
139678909362696
```

Example: Cloning by using **copy()** method (demo27.py)

```
x=[10, 20, 30]

y=x.copy()

print(x)

print(y)

print(id(x))

print(id(y))
```

Output:

```
[10, 20, 30]
[10, 20, 30]
139711152551944
139711152566792
```

6. MATHEMATICAL OPERATORS on LIST

a) The concatenation operator (+):

- “+” operator **concatenates two list objects** to join them and returns a single list.
- For this, both the **operands** should be **list type**, else we will get **TypeError**.

Example: Concatenation (demo28.py)

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

b= [4, 5, 6]

c = a + b
```

```
print(c)
```

Output: `[1, 2, 3, 4, 5, 6]`

Example: TypeError (demo29.py)

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

```
b= 'Ram'
```

```
c = a + b
```

```
print(c)
```

Output: `TypeError: can only concatenate list (not "str") to list`

b) Multiplication operator in Lists:

- The “*” **operator** works to **repeat elements** in the list by the said number of times.
- For this **one operand** should be **list** and the **other operand** should be an **integer**, else we get **TypeError**.

Example: Multiplication operator (demo30.py)

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

```
print(a)
```

```
print(2*a)
```

Output:

```
[1, 2, 3]  
[1, 2, 3, 1, 2, 3]
```

7. COMPARISON of LISTS in Python

- We can use **relational operators** (<, <=, >, >=) on the List objects for **comparing two lists**.

- Initially, the **first two items** are compared, and if they **are not the same** then the corresponding result is returned. If they **are equal**, the next two items are compared, and so on.

Example: Comparison Operators (demo31.py)

```
print([1, 2, 3] < [2, 2, 3])
print([1, 2, 3] < [1, 2, 3])
print([1, 2, 3] <= [1, 2, 3])
print([1, 2, 3] < [1, 2, 4])
print([1, 2, 3] < [0, 2, 3])
print([1, 2, 3] == [1, 2, 3])
```

Output:

```
True
False
True
True
False
True
```

While **comparing lists** that are loaded **with strings** the following things are considered for the comparison.

1. The **number of elements**
2. The **order of elements**
3. The **content of elements** (case sensitive)

Example: Comparison of lists (demo32.py)

```
x=["abc", "def", "ghi"]
y=["abc", "def", "ghi"]
z=["ABC", "DEF", "GHI"]
a=["abc", "def", "ghi", "jkl"]
print(x==y)
print(x==z)
```

```
print(x==a)
```

Output:

```
True  
False  
False
```

8. Membership Operators In List:

- We can check if the **element is a member of a list or not** by using membership operators. They are:
 1. **in operator**
 2. **not in operator**
- If the **element is a member of the list**, then the **in** operator returns **True** otherwise **False**.
- If the **element is not in the list**, then **not in** operator returns **True** otherwise **False**.

Example: Membership operators (demo33.py)

```
x=[10, 20, 30, 40, 50]  
  
print(20 in x) # True  
  
print(20 not in x) # False  
  
print(90 in x) # False  
  
print(90 not in x) # True
```

Output:

```
True  
False  
False  
True
```

9. List Comprehensions In Python:

- List comprehension is a **short and easy way to make new lists** in Python. It **takes each item** from another sequence (like a list, tuple, or range), **does something with it**, and **puts the result into a new list**.

- It is **cleaner, faster, and easier to read** than writing long `for` loops.

Syntax for list comprehension would be

```
list = [expression for item in iterable_obj if condition]
```

Parameters:

- **expression:** operation or value to include in the new list.
- **item:** current element from the iterable.
- **iterable:** sequence like a list, tuple or range.
- **if condition (optional):** filter to include only items that satisfy the condition.

Example:

- Generally, for **multiplying each element of a list with 2**, we take a `for` loop and taking one item at a time, multiply it with 2 and finally save the value in a new list.

Example: Multiplying with 2 (demo35.py)

```
x = [1, 2, 3, 4]
y = []
for i in x:
    y.append(i*2)
print(y)
```

Output: `[2, 4, 6, 8]`

- The above code can be written in a **single line using list comprehensions**.

Example 1: *List Comprehensions (demo36.py)*

```
x = [1, 2, 3, 4]
y = [i*2 for i in x]
print(y)
```

Output: [2, 4, 6, 8]

Example 2: *List Comprehensions (demo37.py): Conditional Statements in List Comprehension*

```
s=range(1, 20, 3)

for i in s: #This loop is for knowing what is in s

print(i)

m=[x for x in s if x%2==0] #List comprehension

print(m)
```

Output:

```
1
4
7
10
13
16
19
[4, 10, 16]
```

Why Use List Comprehension?

- **Cleaner code:** Combines looping, filtering and transformation in one line.
- **More readable:** Avoids verbose loops and temporary variables.
- **Faster execution:** Often faster than equivalent for-loops.

10. Nested Lists In Python (2D List):

- A **nested list** in Python is simply a list that contains other lists as its elements.
It is like a **list inside another list**.
- This concept is useful when we need to represent data in a **hierarchical structure** or in the form of a **matrix/table (2D data)**. Nested lists can also go deeper (3D, 4D, etc.), where each element itself can contain multiple lists.

Key Points:

- A normal list contains single elements (numbers, strings, etc.).
- A nested list contains **lists as elements**.
- We can use **multiple indexing** (`list[i][j]`) to access elements inside.
- Nested lists are very useful to store **tabular data, matrices, game boards, student marks, etc.**

Example: Nested Lists (demo34.py)

```
a = [80, 90]
b = [10, 20, 30, a]
print(b[0])
print(b[1])
print(b[2])
print(b[3])
```

Output:

```
10
20
30
[80, 90]
```

Example:

```
#initialize the 2D list
matrix = [
    [1, 2, 3],    # Row 0
    [4, 5, 6],    # Row 1
    [7, 8, 9]     # Row 2
]

#Accessing Elements
print(matrix[0])    # [1, 2, 3] (first row)
print(matrix[2][1]) # 8 (row 2, column 1)

#Accessing 2D Elements Through Looping
for row in matrix:
    for col in row:
        print(col, end=" ")
    print()

#Modifying Elements
matrix[1][1] = 50    # Change 5 → 50
print(matrix)
```

```
[1, 2, 3]
8
1 2 3
4 5 6
7 8 9
[[1, 2, 3], [4, 50, 6], [7, 8, 9]]
```