

Lists

Introduction

A list is a standard data type of Python that can store a sequence of values belonging to any type. The **Lists** are contained within square brackets (`[]`). Following are some examples of lists in Python:

```
[ ] #Empty list
[1, 2, 3] #List of integers
[1, 2, 5.6, 9.8] #List of numbers (Floating point and Integers)
['a', 'b', 'c'] #List of characters
['a', 1, 4.3, "Zero"] #List of mixed data types
["One", "Two", "Three"] #List of strings
```

Creating Lists

In Python programming, a list is created by placing all the items (elements) inside square brackets `[]`, separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
list1 = [ ] #Empty list
list2 = [1, 2, 3] #List of integers
list3 = [1, "One", 3.4] #List with mixed data types
```

A list can also have another list as an element. Such a list is called a **Nested List**.

```
list4 = ["One", [8, 4, 6], ['Three']] #Nested List
```

Operations On Lists

Accessing Elements in a List

List indices start at 0 and go on till 1 less than the length of the list. We can use the index operator `[]` to access a particular item in a list. Eg.

Index:	0	1	2	3	4
	1	10	34	23	90

Note: Trying to access indexes out of the range `(0 ,lengthOfList-1)`, will raise an **IndexError**. Also, the index must be an integer. We can't use float or other types, this will result in **TypeError**.

Let us take an example to understand how to access elements in a list:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
>> print(l1[0]) #Output: 'Mother'
>> print(l1[2]) #Output: 'Daughter'
>> print(l1[4]) #Output: 23
```

Negative Indexing

Python allows negative indexing for its sequences. The index of **-1** refers to the last item, **-2** to the second last item, and so on. The negative indexing starts from the last element in the list.

Positive Indexing: 0 1 2 3 4 →

1	10	34	23	90
---	----	----	----	----

Negative Indexing: -5 -4 -3 -2 -1 ←

Let us take an example to understand how to access elements using negative indexing in a list:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
>> print(l1[-1]) #Output: 23
>> print(l1[-2]) #Output: 10
>> print(l1[-6]) #Output: IndexError error
```

Changing Elements of a List

Once a list is created, we can even change the elements of the list. This is done by using the assignment operator (=) to change the value at a particular list index. This can be done as follows:

```
l1 = ['Mother', 'Father', 'Daughter', 10, 23]
l1[-1] = "Daughter" #Changing the last element to "Daughter"
l1[3] = 12 #Changing the element at index 3 to 12
print(l1)
```

Output:

```
['Mother', 'Father', 'Daughter', 12, "Daughter"]
```

Concatenation of Lists

Joining or concatenating two list in Python is very easy. The concatenation operator (+), can be used to join two lists. Consider the example given below:

```
l1= [1,2,3] #First List
l2= [3,4,5] #Second List
```

```
l3= l1+l2 #Concatenating both to get a new list  
print(l3)
```

Output:

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

Note: The + operator when used with lists requires that both the operands are of list types. You cannot add a number or any other value to a list.

Repeating/Replicating Lists

Like strings, you can use * operator to replicate a list specified number of times. Consider the example given below:

```
>>> l1 = [1,2, 10, 23]  
>>> print(l1*3)  
[1,2,10,23,1,2,10,23,1,2,10,23] #Output
```

Notice that the above output has the same list **l1** repeated **3** times within a single list.

List Slicing

List slicing refers to accessing a specific portion or a subset of a list while the original list remains unaffected. You can use indexes of list elements to create list slices as per the following syntax:

```
slice= <List Name>[StartIndex : StopIndex : Steps]
```

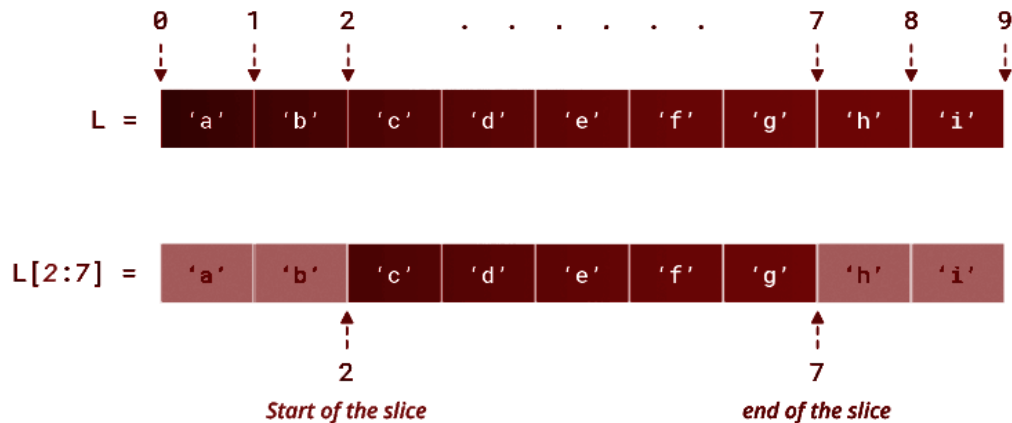
- The **StartIndex** represents the index from where the list slicing is supposed to begin. Its default value is 0, i.e. the list begins from index 0 if no **StartIndex** is specified.
- The **StopIndex** represents the last index up to which the list slicing will go on.

Its default value is `(length(list)-1)` or the index of the last element in the list.

- **steps** represent the number of steps. It is an optional parameter. **steps**, if defined, specifies the number of elements to jump over while counting from StartIndex to StopIndex. By default, it is 1.
- The list slices created, include elements falling between the indexes **StartIndex** and **StopIndex**, including **StartIndex** and not including **StopIndex**.

Here is a basic example of list slicing.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
print(L[2:7])
```



As, you can see from the figure given above, we get the output as:

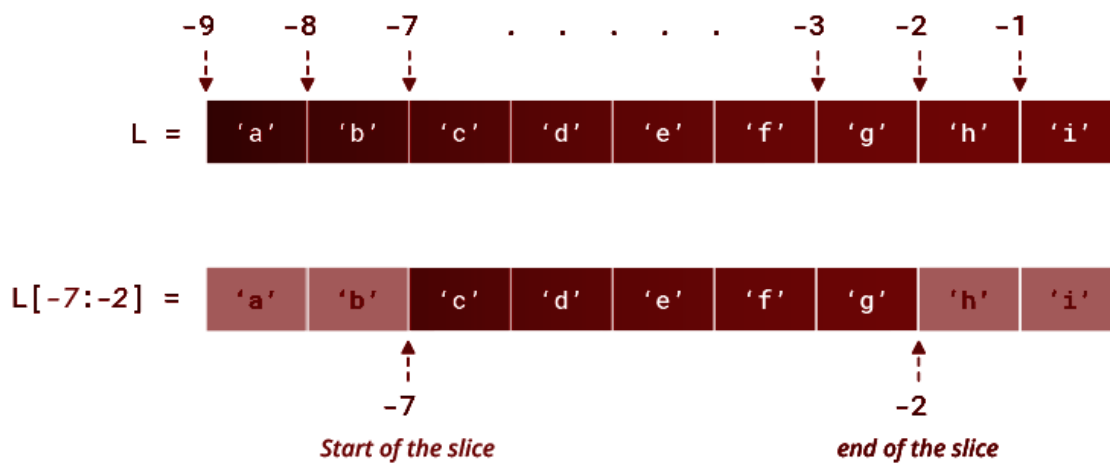
```
['c', 'd', 'e', 'f', 'g']
```

Slice Using Negative Indices

You can also specify negative indices while slicing a list. Consider the example given

below.

```
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
print(L[-7:-2])
```



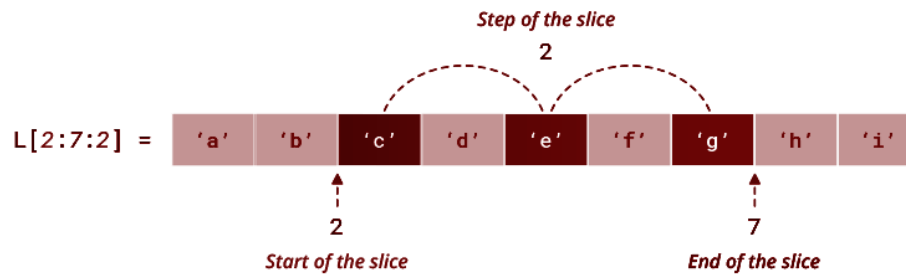
Thus, we get the output as:

```
['c', 'd', 'e', 'f', 'g']
```

Specify Step of the Slicing

You can specify the step of the slicing using the **steps** parameter. The **steps** parameter is optional and by default 1.

```
# Print every 2nd item between position 2 to 7  
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
print(L[2:7:2])
```



The output will be:

```
['c', 'e', 'g']
```

You can even specify a negative step size:

```
# Print every 2nd item between position 6 to 1
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[6:1:-2])
```

The output will be:

```
['g', 'e', 'c']
```

Slice at Beginning & End

Omitting the **StartIndex** starts the slice from the index 0. Meaning, `L[:stop]` is equivalent to `L[0:stop]`.

```
# Slice the first three items from the list
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[:3])
```

Output

```
['a', 'b', 'c']
```

Whereas, omitting the **StopIndex** extends the slice to the end of the list. Meaning, **L[start:]** is equivalent to **L[start:len(L)]**.

```
# Slice the last three items from the list
L = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
print(L[6:])
```

Output

```
['g', 'h', 'i']
```

Reversing a List

You can reverse a list by omitting both **StartIndex** and **StopIndex** and specifying **steps** as -1.

```
L = ['a', 'b', 'c', 'd', 'e']
print(L[::-1])
```

Output:

```
['e', 'd', 'c', 'b', 'a'] #Reversed List
```

List Methods

append(): Used for appending/adding elements at the end of a list.

Syntax: <ListName>.append(element)

Example:

```
>>> li=[1,2,3,4]
>>> li.append(5) #Append 5 to the end of the list
>>> li
[1,2,3,4,5]
```


extend(): Adds the contents of **List2** to the end of **List1**.

Syntax: <ListName1>.extend(<ListName2>)

Example:

```
>>> l1=[1,2,3,4]
>>> l2=[5,6,7,8]
>>> l1.extend(l2) #Adds contents of l2 to l1 at the end
>>> l1
[1,2,3,4,5,6,7,8]
```

**** .append() vs .extend():**

The only difference between **.append()** and **.extend()** is that, the **.append()** method adds an element at the back of a given list (*appends an element at the end of a list*). On the other hand, the **.extend()** method adds the contents of another list at the end of the given list i.e. it merges two lists (*extends the given list*). See the examples given above for better clarity.

insert(): Inserts an element at a specified position/index in a list.

Syntax: <ListName>(position, element)

Example:

```
>>> li=[1,2,3,4]
>>> li.insert(2,5) #Insert 5 at the index no. 2
>>> li
[1,2,5,3,4]
```

sum() : Returns the sum of all the elements of a List. *(Used only for lists containing numerical values)*

Syntax: `sum(<ListName>)`

Example:

```
>>> l1=[1,2,3,4]
>>> sum1= sum(l1) #Finds the sum of all elements in l1
>>> sum1
10
```

count(): Returns the total number of times a given element occurs in a List.

Syntax: `<ListName>.count(element)`

Example:

```
>>> l1=[1,2,3,4,4,3,5,4,4,2]
>>> c= l1.count(4) #Number of times 4 occurs in the list
>>> c
4
```

len(): Returns the total length of a List.

Syntax: `len(<ListName>)`

Example:

```
>>> l1=[1,2,3,4,5]
>>> len(l1)
5
```

index(): Returns the index of first occurrence of an element in a list. If element is not present, it returns -1.

Syntax: <ListName>.index(element)

Example:

```
>>> l1=[1,2,3,4]
>>> l1.index(3)
2
```

min(): Returns the minimum element in a List.

Syntax: min(<ListName>)

Example:

```
>>> l1=[1,2,3,4]
>>> min(l1)
1
```

max(): Returns the maximum element in a List.

Syntax: max(<ListName>)

Example:

```
>>> l1=[1,2,3,4]
>>> max(l1)
4
```

pop(): It deletes and returns the element at the specified index. If we don't mention the index, it by default pops the last element in the list.

Syntax: <ListName>.pop([index])

Example:

```
>>> l1=[1,2,3,4]
```

```
>>> poppedElement= l1.pop(2)
>>> poppedElement #Element popped
3
>>> l1 #List after popping the element
[1,2,4]
```

Note: Index must be in range of the List, otherwise **IndexError** occurs.

del() : Element to be deleted is mentioned using list name and index.

Syntax: del <ListName>[index]

Example:

```
>>> l1=[1,1,12,3]
>>> del l1[2]
>>> l1
[1,1,3]
```

remove() : Element to be deleted is mentioned using list name and element.

Syntax: <ListName>.remove(element)

Example:

```
>>> l1=[1,1,12,3]
>>> l1.remove(12)
>>> l1
[1,1,3]
```

Looping On Lists

There are multiple ways to iterate over a list in Python.

Using **for** loop

```
li = [1, 3, 5, 7, 9]
# Using for loop
for i in li:
    print(i) #Print the element in the list
```

Output:

```
1
3
5
7
9
```

Using **for** loop and **range()**

```
list = [1, 3, 5, 7, 9]
length = len(list) #Getting the length of the list
for i in range(length): #Iterations from 0 to (length-1)
    print(i)
```

Output:

```
1
3
5
7
9
```

You can even use **while()** loops. Try using the **while()** loops on your own.

Taking Lists as User Inputs

There are two common ways to take lists as user inputs. These are:

- Space Separated Input of Lists
- Line Separated Input of Lists

Line Separated Input Of List

The way to take line separated input of a list is described below:

- Create an empty list.
- Let the number of elements you wish to put in the list be N.
- Run a loop for N iterations and during these N iterations do the following:
 - Take an element as user input using the `input()` function.
 - Append this element to the list we created.
- At the end of N iterations, you would have appended N desired elements to your list.
- In this, different elements will have to be entered by the user in different lines.

Consider the given example:

```
li=[] #Create empty list
for i in range(5): #Run the loop 5 times
    a=int(input()) #Take user input
    li.append(a) #Append it to the list
```

```
print(li) #Print the list
```

The above code will prompt the user to input 5 integers in 5 lines. These 5 integers will be appended to a list and the list will be printed.

Space Separated Input Of List

in Python, a user can take multiple values or inputs in one line by two methods.

- Using **split()** method
- Using List comprehension

Using **split()** method

This function helps in taking multiple inputs from the user in a single line . It breaks the given input by the specified **separator**. If a **separator** is not provided then any white space is treated as a separator.

Note: The **split()** method is generally used to split a string.

Syntax :

```
input().split(<separator>) #<separator> is optional
```

Example :

```
In[]: a= input().split()
In[]: print(a)
User[]: 1 2 3 4 5 #User inputs the data (space separated input)
Out[]: ['1', '2', '3', '4', '5']
```

Now, say you want to take comma separated inputs, then you will use **,** as the separator. This will be done as follows:

```
In[]: a= input().split(",")
In[]: print(a)
```

```
User[]: 1,2,3,4,5 #User inputs the data (space separated input)
Out[]: ['1', '2', '3', '4', '5']
```

Note: Observe that the elements were considered as characters and not integers in the list created using the `split()` function. (What if you want a list of integers?- Think)

Using List Comprehension

List comprehension is an elegant way to define and create a list in Python. We can create lists just like mathematical statements in one line only.

A common syntax to take input using list comprehension is given below.

```
inputList= [int(x) for x in input().split()]
```

Example :

```
In[1]: li= [int(x) for x in input().split()]
In[2]: print(li)
User[]: 1 2 3 4 5 #User inputs the data (space separated input)
Out[]: [1,2,3,4,5] #List has integers
```

Here, `In[1]` typecasts `x` in the list `input().split()` to an integer and then makes a list out of all these `x`'s.

Linear Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order.

Linear Search Algorithm

We have been given a **list** and a **targetValue**. We aim to check whether the given **targetValue** is present in the given **list** or not. If the element is present we are required to print the index of the element in the list and if the element is not present we print **-1**.

(First, let us implement this using a simple loop, then later we will see how to implement linear search using functions.)

The following are the steps in the algorithm:

1. Traverse the given list using a loop.
2. In every iteration, compare the **targetValue** with the value of the element in the list in the current iteration.
 - If the values match, print the current index of the list.
 - If the values do not match, move on to the next list element.
3. If no match is found, print -1.

Pseudo-Code

```
for each element in the array: #For Loop to traverse the List
    if element == targetValue #Compare the targetValue to the element
        print(indexOf(item))
```

Python Code

```
li= [int(x) for x in input().split()] #Taking List as user input
targetValue= int(input()) #User input for targetValue
```

```

found = False #Boolean value to check if we found the targetValue
for i in li:
    if (i==targetValue): #If we found the targetValue
        print(li.index(i)) #Print the index
        found = True #Set found as True as we found the targetValue
        break #Since we found the targetValue, we break out of loop
if found is False:#If we did not find the targetValue
    print(-1)

```

We have:

```

User[1]: 1 2 4 67 23 12 #User input for list
User[2]: 4 #User input= targetValue
Out[]: 2 #Index of 4

```

Linear Search Through Functions

We will create a function `linearSearch`, which will have the following properties:

- Take `list` and `targetValue` as parameters.
- Run a loop to check the presence of `targetValue` in `list`.
- If it is present then it returns the index of the `targetValue`.
- If it is not present, then it returns `-1`.

We will call this function and then print the return value of the function.

Python Code

```

def linearSearch(li,targetValue):

```

```
for i in li:
    if (i==targetValue): #If we found the targetValue
        return li.index(i) #Return the index
return -1 #If not found, return -1
li= [int(x) for x in input().split()] #Taking list as user input
targetValue= int(input()) #User input for targetValue
print(linearSearch(li,targetValue)) #Print the return value
```

Mutable And Immutable Concept

The Python data types can be broadly categorized into *two* - **Mutable** and **Immutable** types. Let us now discuss these two types in detail.

- Since everything in Python is an Object, every variable holds an object instance.
- When an object is initiated, it is assigned a unique object id (Address in the memory).
- Its type is defined at runtime and once set it can never change.
- However, its state can be changed if it is **mutable**. In other words, the value of a **mutable** object can be changed after it is created, whereas the value of an **immutable** object can't be changed.

Note: Objects of built-in types like (int, float, bool, str, tuple, Unicode) are **immutable**. Objects of built-in types like (list, set, dict) are **mutable**. A list is a mutable as we can insert/delete/reassign values in a list.