List Data Structure

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.

- List need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects.
- Lists are mutable, and hence, they can be altered even after their creation.
- Duplicate entries are allowed.
- Insertion order is important to be preserved.
- Growable

Creating A list

```
# Creating a List
List = []
print("Blank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["aditya", "lokhande", "hehe"]
print("\nList Items: ")
print(List[0])
print(List[2])

# Creating a List with
# mixed type of values
List = [1, 2, 'aditya', 4, 'neha', 6, 'swati']
print("\nList with the use of Mixed Values: ")
print(List)

#Creating list using range function
List = [item for item in range(10,23,1)]
print("\nList using range")
print(List)

#Taking Input From User
List = input("\nEnter Elements Of List : ").split()
List = list(List)
print("List Is : ",List)
```

```
Blank List:
[]

List of numbers:
[10, 20, 14]

List Items:
aditya
hehe

List with the use of Mixed Values:
[1, 2, 'aditya', 4, 'neha', 6, 'swati']

List using range
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]

Enter Elements Of List : 12 23 5
List Is :  ['12', '23', '5']
```

Accessing List Elements

- Using Indexing
- Using Slice Operator

[IMAGE]

Using Indexing

> +ve index : forward direction
> -ve index : backward direction

```
# Sample list
my_list = [10, 20, 30, 40, 50]

# Accessing individual elements using indexing
element_0 = my_list[0]
element_1 = my_list[1]
element_2 = my_list[2]
element_3 = my_list[3]
element_4 = my_list[4]

print(element_0)  # Output: 10
print(element_1)  # Output: 20
print(element_2)  # Output: 30
print(element_3)  # Output: 40
print(element_4)  # Output: 50

# Accessing elements using negative indexing
last_element = my_list[-1]
second_last_element = my_list[-2]
third_last_element = my_list[-3]

print(last_element)          # Output: 50
print(second_last_element)   # Output: 40
print(third_last_element)    # Output: 30
```

```
10
20
30
40
50
50
40
30
```

Using Slice Operator

The slice operator in Python allows you to access a portion of a list, rather than just a single element. It uses the syntax list[start:stop], where start is the index of the first element you want to include, and stop is the index of the first element you want to exclude.

```
# Sample list
my_list = [10, 20, 30, 40, 50]

# Using slice operator to access a portion of the list
# It includes elements from index 1 up to (but not including) index 4
sliced_list = my_list[1:4]
print(sliced_list)  # Output: [20, 30, 40]

# Using negative indexing with slice operator
# It includes elements from the second last up to (but not including) the last element
negative_sliced_list = my_list[-2:]
print(negative_sliced_list)  # Output: [40, 50]

# Slicing with step size (taking every other element)
step_sliced_list = my_list[::2]
print(step_sliced_list)  # Output: [10, 30, 50]

# Slicing with negative step size (reversing the list)
reverse_list = my_list[::-1]
print(reverse_list)  # Output: [50, 40, 30, 20, 10]
```

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

Nested List

A nested list is a list that contains other lists as elements. In Python, you can create a list where each element is itself a list, forming a nested structure. This allows you to create multi-dimensional data structures.
Nested lists are useful when you need to work with multi-dimensional data, such as matrices or tables, or when you want to group related data together in a hierarchical manner. They allow you to organize data efficiently and access individual elements easily using indexing.

```
# Nested list example
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
print(nested_list)

# Accessing elements of the nested list
print(nested_list[0])     # Output: [1, 2, 3]
print(nested_list[1])     # Output: [4, 5, 6]
print(nested_list[2])     # Output: [7, 8, 9]

# Accessing specific elements within the nested list
print(nested_list[0][0])  # Output: 1
print(nested_list[1][1])  # Output: 5
print(nested_list[2][2])  # Output: 9
```

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

Traversing List

List traversal in Python refers to the process of visiting or accessing each element in a list one by one. It involves iterating over all the elements in the list to perform some operation or check some condition on each element.

Methods To Traverse

- while loop
- for loop
- range and len
- enumerate

```
# Sample list
my_list = [10, 20, 30, 40, 50]

# Using while loop
print("Traversing using while loop:")
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1

# Using for loop
print("\nTraversing using for loop:")
for element in my_list:
    print(element)

# Using range() and len()
print("\nTraversing using range() and len():")
for i in range(len(my_list)):
    print(my_list[i])

# Using enumerate()
print("\nTraversing using enumerate():")
for index, element in enumerate(my_list):
    print(index, element)
```

```
Traversing using while loop:
10
20
30
40
50

Traversing using for loop:
10
20
30
40
50

Traversing using range() and len():
10
20
30
40
50

Traversing using enumerate():
0 10
1 20
2 30
```

```
3 40
4 50
```

Important Functions Of List

- len() : return the size of list, string, tuple, set, dict
- count() : counts the occurence of a particular elements
- index() : return the index of first occurence of an elements or raises value error otherwise.

```
# len()
my_list = [10, 20, 30, 40, 50]
list_length = len(my_list)
print("Length of the list:", list_length)  # Output: 5

# count()
my_list = [10, 20, 30, 20, 40, 20, 50]
count_of_20 = my_list.count(20)
print("Count of 20 in the list:", count_of_20)  # Output: 3

# index()
my_list = [10, 20, 30, 40, 50]
index_of_30 = my_list.index(30)
print("Index of 30 in the list:", index_of_30)  # Output: 2

# Trying to find the index of an element that is not present in the list
try:
    index_of_100 = my_list.index(100)
    print("Index of 100 in the list:", index_of_100)
except ValueError as e:
    print("Element not found in the list.")
```

```
Length of the list: 5
Count of 20 in the list: 3
Index of 30 in the list: 2
Element not found in the list.
```

Adding Elements In List

- Using append() : Adds a single element to the end of the list.
- Using extend() : Adds multiple elements from another iterable (list, tuple, etc.) to the end of the list.
- Using insert() : Adds an element at a specific index in the list.
- Using += operator : Adds elements from another list to the end of the list.
- Using list concatenation : Concatenates two lists and creates a new list.

Remember that these methods modify the original list in-place (except for list concatenation, which creates a new list). If you want to preserve the original list, make a copy before adding elements.

```
# Original list
my_list = [1, 2, 3]

# Using append() to add a single element
my_list.append(4)
print("After appending 4:", my_list)  # Output: [1, 2, 3, 4]

# Using extend() to add multiple elements
additional_elements = [5, 6, 7]
my_list.extend(additional_elements)
print("After extending with [5, 6, 7]:", my_list)  # Output: [1, 2, 3, 4, 5, 6, 7]

# Using insert() to add an element at a specific index
my_list.insert(1, 10)  # Insert 10 at index 1
print("After inserting 10 at index 1:", my_list)  # Output: [1, 10, 2, 3, 4, 5, 6, 7]

# Using += operator to add elements from another list
additional_elements = [8, 9]
my_list += additional_elements
print("After using += operator with [8, 9]:", my_list)  # Output: [1, 10, 2, 3, 4, 5, 6, 7, 8, 9]

# Using list concatenation to create a new list
more_elements = [11, 12]
new_list = my_list + more_elements
print("New list after concatenation:", new_list)  # Output: [1, 10, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12]
```

```
After appending 4: [1, 2, 3, 4]
After extending with [5, 6, 7]: [1, 2, 3, 4, 5, 6, 7]
After inserting 10 at index 1: [1, 10, 2, 3, 4, 5, 6, 7]
After using += operator with [8, 9]: [1, 10, 2, 3, 4, 5, 6, 7, 8, 9]
New list after concatenation: [1, 10, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12]
```

Deleting Elements From List

- Using del : The del statement is used to remove an element from a list using its index.
- Using remove() : The remove() method is used to remove the first occurrence of a specific element.
- Using pop() : The pop() method is used to remove an element from a specific index and return its value.

```
# Original list
my_list = [1, 2, 3, 4, 5, 6, 7, 8]

# Using del to remove an element by index
del my_list[2]  # Remove element at index 2 (value 3)
print("After using del:", my_list)  # Output: [1, 2, 4, 5, 6, 7, 8]

# Using remove to remove an element by value
my_list.remove(5)  # Remove the first occurrence of 5
print("After using remove:", my_list)  # Output: [1, 2, 4, 6, 7, 8]

# Using pop to remove an element by index and return its value
popped_element = my_list.pop(1)  # Remove element at index 1 (value 2) and store in popped_element
print("After using pop:", my_list)  # Output: [1, 4, 6, 7, 8]
print("Popped element:", popped_element)  # Output: 2


my_list = [1, 2, 3, 4, 5]

try:
    # Using del to remove an element by index
    del my_list[2]  # Remove element at index 2 (value 3)
    print("After using del:", my_list)  # Output: [1, 2, 4, 5]

    # Using remove to remove an element by value
    my_list.remove(4)  # Remove the first occurrence of 4
    print("After using remove:", my_list)  # Output: [1, 2, 5]

    # Trying to delete an element with an invalid index
    del my_list[5]  # Trying to delete index 5, which doesn't exist
except IndexError as e:
    print("Error:", e)  # Output: Error: list assignment index out of range

    # Trying to remove an element that does not exist
    try:
        my_list.remove(4)  # Trying to remove value 4, which doesn't exist in the list
    except ValueError as e:
        print("Error:", e)  # Output: Error: list.remove(x): x not in list
```

```
After using del: [1, 2, 4, 5, 6, 7, 8]
After using remove: [1, 2, 4, 6, 7, 8]
After using pop: [1, 4, 6, 7, 8]
Popped element: 2
After using del: [1, 2, 4, 5]
After using remove: [1, 2, 5]
Error: list assignment index out of range
Error: list.remove(x): x not in list
```

Ordering Elememts Of List

Ordering Methods:

- Using sorted() : The sorted() function returns a new list containing all the elements of the original list in ascending order. It does not modify the original list.
- Using sort() : The sort() method sorts the elements of the list in-place, meaning it modifies the original list.
- Using reverse=True with sorted() or sort() : You can sort the elements in descending order by using the reverse=True argument with sorted() or sort(). Using key parameter with sorted() or sort():

You can provide a custom function using the key parameter to specify the sorting order based on some specific criteria.

Reversing Method:

- Using reverse() : The reverse() method is used to reverse the elements of the list in-place, modifying the original list.
- Using slicing with [::-1] : You can use slicing with a step of -1 to create a reversed copy of the original list without modifying the original list.

```
# Original list
my_list = [5, 2, 8, 1, 3]

# Using sorted() to get a new sorted list (ascending order)
```

```
sorted_list = sorted(my_list)
print("Sorted list:", sorted_list)  # Output: [1, 2, 3, 5, 8]

# Using sort() to sort the original list in-place (ascending order)
my_list.sort()
print("Sorted list in-place:", my_list)  # Output: [1, 2, 3, 5, 8]

# Using sorted() with reverse=True to get a new sorted list (descending order)
descending_sorted_list = sorted(my_list, reverse=True)
print("Descending sorted list:", descending_sorted_list)  # Output: [8, 5, 3, 2, 1]

# Using sort() with reverse=True to sort the original list in-place (descending order)
my_list.sort(reverse=True)
print("Descending sorted list in-place:", my_list)  # Output: [8, 5, 3, 2, 1]

# Reversing the original list in-place using reverse()
my_list.reverse()
print("Reversed list:", my_list)  # Output: [1, 2, 3, 5, 8]

# Creating a reversed copy of the original list using slicing
reversed_list = my_list[::-1]
print("Reversed copy of the list:", reversed_list)  # Output: [8, 5, 3, 2, 1]
```

```
Sorted list: [1, 2, 3, 5, 8]
Sorted list in-place: [1, 2, 3, 5, 8]
Descending sorted list: [8, 5, 3, 2, 1]
Descending sorted list in-place: [8, 5, 3, 2, 1]
Reversed list: [1, 2, 3, 5, 8]
Reversed copy of the list: [8, 5, 3, 2, 1]
```

Aliasing

In Python, aliasing refers to the situation when two or more variables refer to the same object in memory. When you create a new variable and assign it the value of an existing variable (e.g., a list), you are creating an alias. Changes made to one variable (alias) will affect the other because they both point to the same object in memory.

```
original_list = [1, 2, 3, 4]

# Creating an alias
alias_list = original_list

# Modifying the alias list
alias_list[0] = 99

# Both lists are modified because they are aliases
print("Original list:", original_list)  # Output: [99, 2, 3, 4]
print("Alias list:", alias_list)        # Output: [99, 2, 3, 4]
```

To avoid aliasing, you can create a true copy of the list. There are two ways to create a copy of a list: a shallow copy or a deep copy.

Shallow Copy

A shallow copy creates a new list, but it only copies the references to the elements in the original list, not the elements themselves. Therefore, changes made to the elements in the shallow copy will also affect the original list, and vice versa.

You can create a shallow copy using the slicing technique [:] or the copy() method from the copy module.

```
import copy

original_list = [1, 2, [3, 4]]
shallow_copy_list = original_list[:]
shallow_copy_list[2][0] = 99

print("Original list:", original_list)        # Output: [1, 2, [99, 4]]
print("Shallow copy list:", shallow_copy_list)  # Output: [1, 2, [99, 4]]
```

```
Original list: [1, 2, [99, 4]]
Shallow copy list: [1, 2, [99, 4]]
```

Deep Copy

A deep copy creates a completely independent copy of the original list, including all the nested elements. Changes made to the deep copy will not affect the original list, and vice versa.

You can create a deep copy using the deepcopy() function from the copy module.

```
import copy

original_list = [1, 2, [3, 4]]
deep_copy_list = copy.deepcopy(original_list)
deep_copy_list[2][0] = 99

print("Original list:", original_list)      # Output: [1, 2, [3, 4]]
print("Deep copy list:", deep_copy_list)    # Output: [1, 2, [99, 4]]
```

```
Original list: [1, 2, [3, 4]]
Deep copy list: [1, 2, [99, 4]]
```

Aliasing and cloning (copying) of a list are two different concepts in Python:

Aliasing:

- Aliasing refers to the situation when two or more variables refer to the same object in memory.
- When you create an alias, you are creating an additional reference to the original list, not a new copy of the list.
- Changes made to one variable (alias) will affect the other because they both point to the same object in memory.
- Aliasing occurs when you assign one variable to another without using any specific copy functions or techniques.

```
# Aliasing example
original_list = [1, 2, 3, 4]
alias_list = original_list
alias_list[0] = 99

print("Original list:", original_list)  # Output: [99, 2, 3, 4]
print("Alias list:", alias_list)        # Output: [99, 2, 3, 4]
```

Cloning (Copying):

- Cloning or copying a list means creating a new list with the same elements as the original list, but the two lists are independent and not related in memory.
- Changes made to the cloned list will not affect the original list, and vice versa.
- There are two types of cloning: shallow copy and deep copy.
  - Shallow copy creates a new list, but it only copies the references to the elements, not the elements themselves. Nested elements are still shared between the original and copied lists.
  - Deep copy creates a completely independent copy of the original list, including all nested elements.

```
import copy

# Shallow copy example
original_list = [1, [2, 3], 4]
shallow_copy_list = original_list[:]
shallow_copy_list[1][0] = 99

print("Original list:", original_list)  # Output: [1, [99, 3], 4]
print("Shallow copy list:", shallow_copy_list)  # Output: [1, [99, 3], 4]

# Deep copy example
original_list = [1, [2, 3], 4]
deep_copy_list = copy.deepcopy(original_list)
deep_copy_list[1][0] = 99

print("Original list:", original_list)  # Output: [1, [2, 3], 4]
print("Deep copy list:", deep_copy_list)  # Output: [1, [99, 3], 4]
```

In summary, aliasing creates a new reference to the same object, whereas cloning creates a new, independent object with the same values. Cloning can be achieved using either a shallow copy or a deep copy, depending on whether you need nested elements to be shared or completely independent between the original and cloned lists.

Matrix Representation

In Python, you can represent a matrix using a list of lists. Each inner list represents a row of the matrix, and the elements within the inner list represent the values in that row. Here's an example of how to represent a 3x3 matrix using a list of lists:

```
# Example of a 3x3 matrix
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

In this example, matrix is a list of three lists, each representing a row of the 3x3 matrix. To access elements of the matrix, you use two indices: one for the row and one for the column. For example, to access the element in the second row and third column (indexing is 0-based), you would use matrix[1][2], which gives you the value 6.

Here's how you can access and modify elements of the matrix:

```
# Accessing elements of the matrix
print(matrix[1][2])  # Output: 6

# Modifying an element of the matrix
matrix[0][1] = 10
print(matrix)
# Output:
# [[1, 10, 3],
#  [4, 5, 6],
#  [7, 8, 9]]
```

You can create matrices of different sizes using the same list of lists approach. For example, a 2x4 matrix can be represented as follows:

```
# Example of a 2x4 matrix
matrix_2x4 = [
    [1, 2, 3, 4],
    [5, 6, 7, 8]
]
```

This approach allows you to work with matrices efficiently using standard list operations, and you can easily perform matrix calculations and manipulations using loops and list comprehension in Python.

List Comprehension

List comprehension is a concise and powerful technique in Python for creating new lists by applying an expression to each element of an existing list (or other iterable). It allows you to create a new list in a single line of code, making the code more readable and efficient.

The general syntax for list comprehension is:

```
new_list = [expression for item in iterable if condition]
```

where:

- `expression`: The operation or transformation to be applied to each element of the iterable to create a new element in the new list.
- `item`: A variable representing each element in the iterable.
- `iterable`: The original list or any other iterable (e.g., another list, tuple, set, etc.).
- `condition` (optional): An optional filter that can be applied to include only specific elements that satisfy the condition.

Here are some examples of list comprehension:

Example 1: Squaring each element of a list:

```
original_list = [1, 2, 3, 4, 5]
squared_list = [x**2 for x in original_list]
print(squared_list)  # Output: [1, 4, 9, 16, 25]
```

Example 2: Filtering even numbers from a list:

```
original_list = [1, 2, 3, 4, 5]
even_numbers = [x for x in original_list if x % 2 == 0]
print(even_numbers)  # Output: [2, 4]
```

Example 3: Creating a list of tuples:

```
names = ["Alice", "Bob", "Charlie"]
name_lengths = [(name, len(name)) for name in names]
print(name_lengths)  # Output: [('Alice', 5), ('Bob', 3), ('Charlie', 7)]
```

Example 4: Nested list comprehension:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_matrix = [element for row in matrix for element in row]
print(flattened_matrix)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehension is a versatile and elegant technique that can greatly simplify your code, especially when you need to create new lists based on existing ones or apply certain transformations and filters to the elements.