Session on Python Lists

Introduction to Python Lists

Python lists are one of the most versatile and widely used data types. They allow you to store collections of items in a single variable. Lists are ordered, mutable (changeable), and allow duplicate elements.

Creating Lists

1. Empty List

```
my_list = []
```

2. List with Initial Values

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

3. List from an Iterable (e.g., range)

```
my_list = list(range(1, 6))  # [1, 2, 3, 4, 5]
```

4. List Comprehension

```
squares = [x**2 for x in range(1, 6)]  # [1, 4, 9, 16, 25]
```

5. Using the `list()` Constructor

```
my_list = list("hello")  # ['h', 'e', 'l', 'l', 'o']
```

List Functions and Methods

1. Appending Items

```
my_list.append(6)  # [1, 2, 3, 4, 5, 6]
```

Session on Python Lists

2. Inserting Items

```
my_list.insert(0, 0)  # [0, 1, 2, 3, 4, 5, 6]
```

3. Removing Items

```
my_list.remove(3)  # [0, 1, 2, 4, 5, 6]
```

4. Popping Items

```
last_item = my_list.pop()  # [0, 1, 2, 4, 5], last_item = 6
```

5. Finding Index

```
index_of_2 = my_list.index(2)  # 2
```

6. Counting Items

```
count_of_4 = my_list.count(4)  # 1
```

7. Reversing the List

```
my_list.reverse()  # [5, 4, 2, 1, 0]
```

8. Sorting the List

```
my_list.sort()  # [0, 1, 2, 4, 5]
```

9. Clearing the List

```
my_list.clear()  # []
```

Session on Python Lists

Uniqueness of Python Lists

- Ordered: Items have a defined order, and this order will not change unless you explicitly do so.

- Mutable: You can change the content of a list after it has been created.

- Allows Duplicate Elements: Lists can have items with the same value.

- Dynamic Size: Lists can grow and shrink as needed.

Key Concepts in Detail

Indexing

Indexing allows you to access individual elements of a list using their position. Python uses zero-based indexing, meaning the first element is at index 0.

Example:

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

# Accessing elements
first_element = my_list[0]  # 10
second_element = my_list[1]  # 20
last_element = my_list[-1]  # 50 (Negative indexing)
```

Slicing

Slicing allows you to access a subset of a list. You specify the start and end indices, and the slicing will return a new list that includes elements from the start index up to, but not including, the end index.

Example:

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

# Slicing
sub_list1 = my_list[1:3]  # [20, 30]
sub_list2 = my_list[:3]   # [10, 20, 30]
sub_list3 = my_list[2:]   # [30, 40, 50]
sub_list4 = my_list[::2]  # [10, 30, 50] (with step)
```

Iteration

Iteration involves looping through the elements of a list. This is commonly done using a `for` loop or a `while` loop.

Example:

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

# Using a for loop
for item in my_list:
    print(item)

# Using a while loop
i = 0
```

Session on Python Lists

```
while i < len(my_list):
    print(my_list[i])
    i += 1
```

List Comprehension

List comprehensions provide a concise way to create lists. They can also include conditions and nested loops.

Example:

```
# Simple list comprehension
squares = [x**2 for x in range(1, 6)]  # [1, 4, 9, 16, 25]

# List comprehension with condition
evens = [x for x in range(1, 11) if x % 2 == 0]  # [2, 4, 6, 8, 10]

# Nested list comprehension
nested_list = [[x for x in range(3)] for _ in range(3)]  # [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Memory Management

Understanding memory management is crucial, especially when working with large lists. Lists in Python are dynamic arrays, meaning they resize automatically. However, this comes with some overhead.

Session on Python Lists

Key Points:

- Dynamic Resizing: Lists resize by allocating more memory than needed. When the allocated memory is filled, Python allocates a new chunk, typically larger, to accommodate more elements.
- Copying Lists: Creating a copy of a list creates a new list with the same elements, which doubles the memory usage temporarily.

Example:

```
import sys

my_list = [10, 20, 30, 40, 50]
print(sys.getsizeof(my_list))  # Returns the memory size of the list

# Copying a list
my_list_copy = my_list.copy()  # Creates a new list with the same elements
print(sys.getsizeof(my_list_copy))  # Same memory size as the original list
```

Comprehensive Example Incorporating All Concepts

```
# Creating a list with initial values
my_list = [10, 20, 30, 40, 50]

# Indexing
first_element = my_list[0]  # 10
last_element = my_list[-1]  # 50
```

```
# Slicing

sub_list1 = my_list[1:3]  # [20, 30]

sub_list2 = my_list[:3]   # [10, 20, 30]

sub_list3 = my_list[2:]   # [30, 40, 50]


# Iteration

for item in my_list:

    print(item)


# List comprehension

squares = [x**2 for x in my_list]  # [100, 400, 900, 1600, 2500]


# Memory management

import sys

print("Original list memory size:", sys.getsizeof(my_list))

print("Squares list memory size:", sys.getsizeof(squares))
```

Examples of List Operations

Example 1: Finding the Maximum Product of Two Elements (LeetCode Style)

```
def max_product(nums):

    nums.sort()

    return (nums[-1] - 1) * (nums[-2] - 1)


# Test
```

```
print(max_product([3, 4, 5, 2]))  # Output: 12
```

Example 2: Remove Duplicates from Sorted List (LeetCode Style)

```
def remove_duplicates(nums):
    if not nums:
        return 0

    write_index = 1

    for i in range(1, len(nums)):
        if nums[i] != nums[i - 1]:
            nums[write_index] = nums[i]
            write_index += 1

    return write_index

# Test
nums = [0, 0, 1, 1, 2, 2, 3, 3, 4]
new_length = remove_duplicates(nums)
print(nums[:new_length])  # Output: [0, 1, 2, 3, 4]
```