Day 10- Trainer FG
(List)

Index

For Trainer(1.50 hrs) ILT

I. Recap

In our last session we learned:

- **Built-in String Methods:**Python provides a wide range of built-in string functions (methods) that allow you to manipulate and work with strings effectively.
- **String Comparison:**String comparison in Python involves comparing two strings to determine their relative order in lexicographic (dictionary) order. Python provides various comparison operators and methods for comparing strings.
- **Format Operator:**In Python, the format operator % is used for string formatting. It allows you to create formatted strings by replacing placeholders (marked by %) with values. The format operator is often referred to as "old-style" string formatting, and while it is still available and functional, Python 3 introduced a more modern and preferred method using the .format() method and f-strings (formatted string literals).

Try this:
Predict the output of the below code

```
str1 = "Hello World 2345"
sum_num = 0
for x in str1:
     if x.isdigit() == True:
          z = int(x)
          sum_num = sum_num + z
print(sum_num)
```

By the end of this session, students will understand how to create and work with lists in Python, perform operations on lists, use list functions, and traverse lists efficiently.

II. What is a List?

In Python, a list is a built-in data structure that represents an ordered collection of items. Lists are one of the most commonly used data structures in Python and are very versatile. Here are some key characteristics and features of Python lists:

- **Ordered:** Lists maintain the order of elements, meaning that the elements are stored and retrieved in the same sequence they were added.

- **Mutable:** Lists are mutable, which means you can change their contents by adding, removing, or modifying elements.

- **Heterogeneous:** Python lists can contain elements of different data types, including numbers, strings, other lists, objects, and more. This makes them versatile for a wide range of applications.

- **Dynamic Size:** Lists can grow or shrink dynamically as you add or remove elements. There's no need to specify a fixed size when creating a list.

Here's an example of creating and manipulating a Python list:

```
# Creating a list
my_list = [1, 2, 3, 'four', 5.0]
print(my_list)
```

In the above example we created a list and then printed the values.

Python lists are very versatile and are used extensively for tasks like data storage, iteration, and manipulation in a wide range of Python programs.

III. List Operations and Slices
In Python, lists are versatile data structures that allow you to store and manipulate collections of items. Here are some common operations and techniques for working with lists, including slices:

Creating a List: You can create a list by enclosing a comma-separated sequence of items within square brackets [].

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

Accessing Elements: You can access individual elements of a list by using their index. List indices start from 0.

```
my_list = [1, 2, 3, 4, 5]
first_element = my_list[0]  # Access the first element (1)
third_element = my_list[2]  # Access the third element (3)
```

Slicing Lists: Slicing allows you to extract a portion of a list. The syntax for slicing is *list[start:stop:step]*, where start is the index to start from, stop is the index to stop before, and step is the step size.

```
my_list = [1, 2, 3, 4, 5]
sub_list = my_list[1:4]  # Get a sub-list from index 1 to 3 ([2, 3, 4])
every_other = my_list[::2]  # Get every other element ([1, 3, 5])
```

Modifying Lists: Lists are mutable, so you can change their elements by assignment.

```
my_list = [1, 2, 3, 4, 5]
my_list[2] = 99  # Change the third element to 99
```

These are some fundamental list operations and techniques in Python. Lists are versatile and widely used for various data manipulation tasks.

IV. List Functions

In Python, lists are versatile data structures, and there are many built-in functions that you can use to manipulate and work with lists effectively. Here are some common list functions:

len(): Returns the number of items in a list.

```
my_list = [1, 2, 3, 4, 5]
length = len(my_list)  # length is 5
```

append(): Adds an item to the end of the list.

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

```
my_list.append(4)  # Adds 4 to the end: [1, 2, 3, 4]
```

extend(): Extends the list by appending elements from another iterable.

```
my_list = [1, 2, 3]
my_list.extend([4, 5])  # Extends with [4, 5]: [1, 2, 3, 4, 5]
```

insert(): Inserts an item at a specified position in the list.

```
my_list = [1, 2, 4]
my_list.insert(2, 3)  # Inserts 3 at index 2: [1, 2, 3, 4]
```

remove(): Removes the first occurrence of a specific value from the list.

```
my_list = [1, 2, 3, 2]
my_list.remove(2)  # Removes the first 2: [1, 3, 2]
```

pop(): Removes and returns an item at a specified index. If no index is provided, it removes and returns the last item.

```
my_list = [1, 2, 3, 4]
popped_item = my_list.pop(2)  # Removes and returns 3, list becomes [1, 2, 4]
```

index(): Returns the index of the first occurrence of a specific value.

```
my_list = [10, 20, 30, 20]
index = my_list.index(20)  # index is 1
```

Finding Elements using index: You can check if an element exists in a list using the in keyword and find its index using the index() method.

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

```
exists = 5 in my_list  # Check if 5 is in the list
index = my_list.index(4)  # Find the index of 4 (returns 3)
```

count(): Returns the number of times a specific value appears in the list.

```
my_list = [1, 2, 2, 3, 2]
count = my_list.count(2)  # count is 3
```

sort(): Sorts the list in ascending order (in-place).

```
my_list = [3, 1, 2]
my_list.sort()  # Sorts the list: [1, 2, 3]
```

sorted(): Returns a new sorted list (leaves the original list unchanged).

```
my_list = [3, 1, 2]
sorted_list = sorted(my_list)  # sorted_list is [1, 2, 3], my_list remains [3, 1, 2]
```

reverse(): Reverses the order of elements in the list (in-place).

```
my_list = [1, 2, 3]
my_list.reverse()  # Reverses the list: [3, 2, 1]
```

Sort and Reverse function Example:

```
my_list = [1, 2, 3, 4, 5]
my_list.sort()             # Sort the list in ascending order
reversed_list = sorted(my_list, reverse=True)  # Create a sorted copy in descending order
my_list.reverse()          # Reverse the list in-place
```

copy(): Returns a shallow copy of the list.

```
my_list = [1, 2, 3]
copied_list = my_list.copy()  # Creates a copy of my_list
```

These are some of the most commonly used list functions in Python. They make it easier to manipulate and work with lists for various tasks.

V. Traversing a list
Traversing a list in Python means iterating through all the elements of the list, one by one, to perform some operation or gather information from each element. There are several ways to traverse a list in Python:

Using a for Loop: The most common and straightforward way to traverse a list is by using a for loop. Here's an example:

```
my_list = [1, 2, 3, 4, 5]
for item in my_list:
    print(item)
```

This code iterates through each element in my_list and prints it to the console.

Using List Index and range(): You can also use a for loop with the range() function to access elements by their index:

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

for i in range(len(my_list)):
    print(my_list[i])
```

This code uses the range() function to generate indices from 0 to the length of the list minus 1 and then accesses the elements using these indices.

Using List Comprehension: List comprehensions are a concise way to traverse and manipulate a list. You can use them to create a new list or perform an action on each element.

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Suppose based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside.

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:
Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

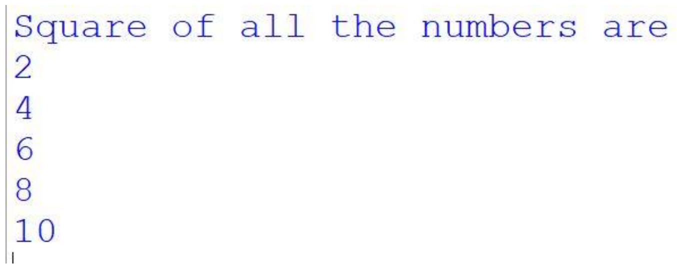
Let’s see another example of list comprehension to print the square of all numbers of a list.

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

# Create a new list with squared elements
squared_list = [x ** 2 for x in my_list]

# Print each element multiplied by 2
print("Square of all the numbers are")
[print(x * 2) for x in my_list]
```

Output:



```
Square of all the numbers are
2
4
6
8
10
```

List comprehensions are especially useful when you want to create a new list based on the existing one.

Using enumerate(): If you need both the element and its index while traversing the list, you can use the enumerate() function:

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

for index, item in enumerate(my_list):
    print(f"Index: {index}, Value: {item}")
```

The enumerate() function returns both the index and the corresponding element in each iteration.

Using a while Loop: Although less common, you can also traverse a list using a while loop by manually incrementing an index variable:

```
my_list = [1, 2, 3, 4, 5]
index = 0

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

Each of these methods allows you to traverse a list in Python, and you can choose the one that best suits your specific use case and coding style.

Exercise

Use GPT to write a program:

1.Hi! Can you create a program using a list which allows us to manage our to-do list interactively, including adding tasks, marking tasks as done, removing tasks, and viewing the current task. The tasks are stored in a list, where each task is represented as a list containing the task name and a boolean value indicating whether it's done or not.

After the ILT, the student has to do a 30 min live lab. Please refer to the Day 10 Lab doc in the LMS.