Lecture Notes: Nested Loops and Lists in Python

Introduction

In this lecture, we covered two powerful tools in Python: **nested loops** and **lists**. Both are fundamental concepts that help us handle repetitive tasks and store collections of data, respectively.

1. Nested Loops

What is a Nested Loop?

A **nested loop** is a loop inside another loop. The outer loop runs first, and for each iteration of the outer loop, the inner loop runs completely. Nested loops allow us to perform complex iterations, making them useful for tasks like processing multi-dimensional data or creating patterns.

Example: Printing Patterns

Imagine you are tasked with printing a triangular pattern like this:

```
*
* *
* * *
* * * *
```

To achieve this, you can use a nested loop:

```python
for i in range(1, 5):
    for j in range(1, i+1):
        print("*", end=" ")
    print()
```

Here, the outer loop controls the rows, and the inner loop controls how many stars are printed on each row.

Real-Life Analogy:

Think of nested loops like having two clocks. The outer clock (hours) ticks once, and for every tick of the outer clock, the inner clock (minutes) completes a full rotation. It's like how nested loops work — the inner loop completes fully for every iteration of the outer loop.

Trick Question #1:

What will be the output of this code?

```
for i in range(2):
    for j in range(2):
        print(i, j)
```

- Think: How many times will the inner loop run for each iteration of the outer loop?

2. Lists in Python

What is a List?

A **list** is a collection of items that can be stored in a single variable. In Python, lists are dynamic, which means you can add, remove, or change the items in the list. Lists are ordered, changeable, and allow duplicate values.

Example: Creating and Using Lists

You can create a list by enclosing items in square brackets:

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

You can access elements in a list by using their index:

```
print(my_list[0])  # Output: 1
```

List Operations:

- **Appending**: Adds an item to the end of the list.

```
my_list.append(6)
```

- **Removing**: Removes the specified item.

```
my_list.remove(2)
```

Real-Life Analogy:

Think of a list like a grocery shopping list. You write down items (which can be numbers, strings, etc.), and you can add more items, remove items, or modify them as needed.

Trick Question #2:

What is the output of this code?

```
my_list = [1, 2, 3, 4]
my_list[1] = 10
print(my_list)
```

- Think: What happens when you modify an element at a specific index?

3. List Comprehensions

List Comprehension is a shorter way to create a new list by using a loop in a single line of code. It's very efficient and makes your code cleaner.

Example:

```
squares = [x**2 for x in range(5)]
print(squares)
```

This creates a list of squares for the numbers 0 to 4.

Trick Question #3:

What will be the output of this list comprehension?

```
my_list = [x for x in range(6) if x % 2 == 0]
print(my_list)
```

- Think: How does the list comprehension filter the values?

4. Using Nested Loops with Lists

You can combine nested loops with lists to iterate over complex data structures like **2D lists** (lists of lists).

Example:

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

for row in matrix:
    for element in row:
        print(element, end=" ")
    print()
```

In this example, the outer loop iterates over each row, and the inner loop iterates over each element in that row.

Summary of Key Concepts

- **Nested Loops**: Loops within loops, used for multi-dimensional operations or repeated tasks.
- **Lists**: Ordered collections of items, allowing you to store, access, and modify data efficiently.
- **List Comprehensions**: A concise way to create and manipulate lists.

- **Combining Nested Loops with Lists**: Useful for working with multi-dimensional data like matrices.

Trick Question #4:

What happens if you try to access an index that doesn't exist in a list?

- **Hint**: Think about error handling in Python.

Preview of Next Lecture: Strings in Python

In the next class, we'll explore **strings** in Python — how to work with text, perform operations like slicing, and manipulate strings efficiently. Here's a sneak peek:

Example:

```
my_string = "Hello, World!"
print(my_string[0:5])  # Output: Hello
```

Strings are a crucial part of Python programming, and we'll dive deeper into their methods and uses.