Looping Through A List

Table of contents

Introduction

- Looping through a list in Python allows us to **access**, **modify**, and **perform operations** on each item in the list.
- Python provides **multiple** ways to iterate over list elements efficiently.
- Let's discuss them **one-by-one.**

01

Using a For Loop

Using a For Loop

- The most common way to iterate through a list is by using a **for** loop.
- The loop automatically **retrieves** each element from the list **one by one**.

Syntax
`for item in list_name:` `    # Code to execute`

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`for fruit in fruits:`
`    print(fruit)`
	Apple
	
	banana
	
	cherry

02

Using Index Numbers

Using Index Numbers

- You can loop through a list using index numbers by combining the **range**() and **len**() functions.
- This method gives **explicit control** over the **index values**.

Syntax
`for i in range(len(list_name)):` `  list_name[i]`

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`for i in range(len(fruits)):`
`  print(f"Index {i}: {fruits[i]}")`
	Index 0: apple
	
	Index 1: banana
	
	Index 2: cherry

03

Using a While Loop

Using a While Loop

A **while** loop can also be used to iterate through a list. This requires:

- Initializing an **index** variable.
- Using the **len**() function to determine the loop **condition**.
- **Incrementing the index** variable manually in each iteration.

Syntax

```
i = 0
while i < len(list_name):
    list_name[i]
    i += 1
```

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`i = 0`
`while i < len(fruits):`
`  print(fruits[i])`
`  i += 1`
	apple
	
	banana
	
	cherry

04

Nested List

Nested List

- To deal with **lists of lists** (**2D lists**), you can use **nested for loops** to access each element.
- The **outer loop** iterates through the **rows**, and the **inner loop** iterates through **each item** in those rows.

Syntax

```
for outer_item in list_of_lists:
    for inner_item in outer_item:
        # Code block
```

Code	Output

`matrix = [`
`  [1, 2, 3],`
`  [4, 5, 6],`
`  [7, 8, 9]`
`]`
`for row in matrix:`
`  for item in row:`
`  print(item, end=" ")`
	1 2 3 4 5 6 7 8 9

05

Using the enumerate() Function

Using the enumerate() Function

- The **enumerate()** function allows us to iterate over a list while also **retrieving** both the **index** and the **item**.
- Using **enumerate()** is **cleaner** and more **readable** than using range(len(list)).

Syntax
`for index, item in enumerate(iterable):` `    # Code to execute`

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`for index, fruit in enumerate(fruits):`
`    print(f"Index {index}: {fruit}")`
	Index 0: apple
	
	Index 1: banana
	
	Index 2: cherry

06

Using the zip() Function

Using the zip() Function

The **zip()** function pairs elements from **multiple lists** together, allowing you to iterate over them simultaneously.

Syntax
`for i1, i2, … in zip(iter1, iter2,…):` `    # Code to execute`

Code	Output

`list1 = ["apple", "banana", "cherry"]`
`list2 = [1, 2, 3]`
 	
`for fruit, number in zip(list1, list2):`
`    print(fruit, number)`
	apple 1
	
	banana 2
	
	cherry 3

07

Using the reversed() Function

Using the reversed() Function

You can iterate through a list in **reverse** order in two ways:

1. **Using the reversed() function**
2. Using **slicing**

Syntax
`for item in reversed(list):` `    # Code to execute`

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`for fruit in reversed(fruits):`
`    print(fruit)`
	cherry
	
	banana
	
	apple

Using Slicing

You can iterate through a list in **reverse** order in two ways:

1. Using the **reversed**() function
2. **Using slicing**

Syntax
`for item in list[::-1]:` `    # Code to execute`

Code	Output

`fruits = ["apple", "banana", "cherry"]`
`for fruit in fruits[::-1]:`
`    print(fruit)`
	cherry
	
	banana
	
	apple

Summary

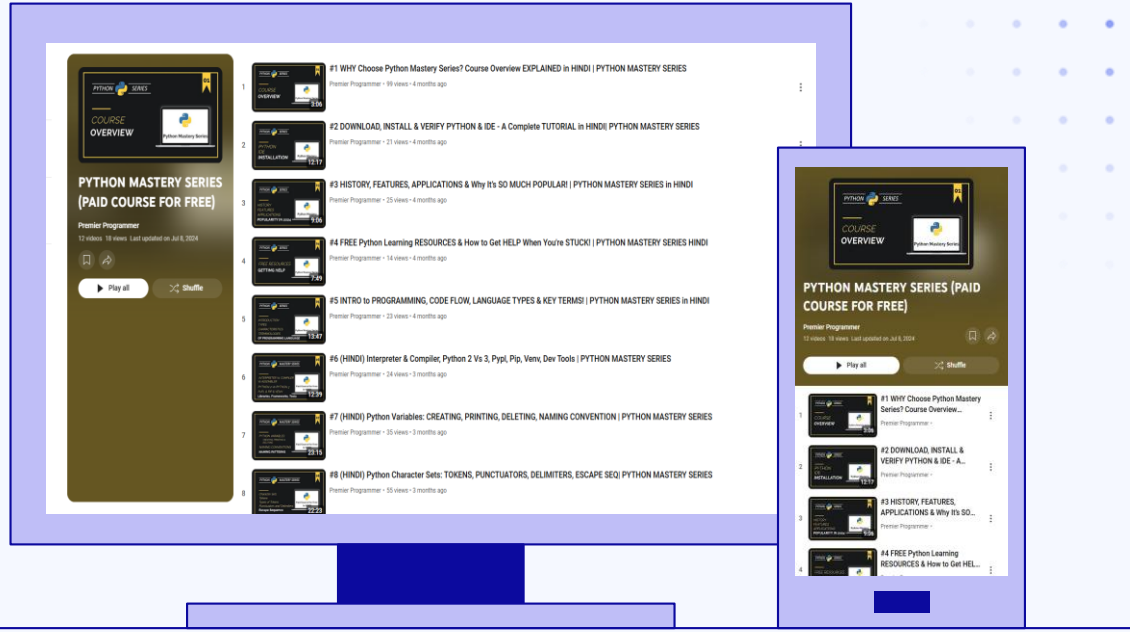
Methods	Use Cases
for loop	**Directly iterate** through list elements.
for loop with index(range(len(list)))	**Access** both **index and value**; useful for modifying elements.
while loop	Provides **manual index control**, useful for dynamic iterations.
enumerate()	Efficient way to **access both index and value** simultaneously.
Zip()	Efficient way to **access multiple lists** simultaneously.

PLEASE NOTE

“I did not include List Comprehension here because in the next video, I’ll cover it in detail.”

WATCH

Level up your coding with each episode in this focused Python series.



Next Video!

List Comprehensions