For Loop Class Theory

Class Theory: Python for Loops

1. Introduction

A `for` **loop** in Python is used to iterate over elements of a sequence (like a list, tuple, set, dictionary keys, or string). Unlike some other languages, Python's `for` loop acts more like an **iterator** you do not need to manage an indexing variable manually.

Key Benefits:

- Easy iteration over items in any iterable (lists, strings, etc.).
- Integrates seamlessly with Python's built-in functions like `range()` .
- Straightforward syntax reduces errors compared to index-based loops.

2. Basic Looping

2.1. Iterating Over a List

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

Explanation:

- The loop runs once for each item in `fruits` .
- The variable `x` is assigned each element in turn.

2.2. Looping Through a String

```
for char in "banana":
    print(char)
```

- Strings are sequences of characters, so Python lets you iterate through each character directly.

3. The `break` Statement

`break` **exits** the entire loop immediately, even if there are remaining elements.

3.1. Stop After Printing "banana"

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

- The loop stops right after encountering “banana”.

3.2. Skip Printing "banana"

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

- Here, “banana” isn’t printed at all because the loop breaks **before** `print(x)` .

4. The `continue` Statement

`continue` **skips the current iteration** and proceeds to the next one.

4.1. Avoid Printing "banana"

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

- When `x` is “banana,” `continue` bypasses the `print` , so “banana” is never printed.

5. Looping with `range()`

`range()` generates a sequence of integers.

5.1. Default Range

```python
for x in range(6):
    print(x)
```

- Iterates from `0` to `5` (6 is not included).

5.2. Specifying a Start and End

```python
for x in range(2, 6):
    print(x)
```

- Iterates from `2` to `5` .

5.3. Specifying a Step

```python
for x in range(2, 30, 3):
    print(x)
```

- Starts at `2` , goes up to (but not including) `30` , incrementing by `3` .

6. `else` Clause in a `for` Loop

- The `else` block runs **if** the loop completes normally (no `break`).

6.1. Simple Use Case

```python
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

- After printing `0` through `5` , it prints “Finally finished!” since no break occurred.

6.2. Breaking the Loop

```python
for x in range(6):
    if x == 3:
        break
    print(x)
else:
    print("Finally finished!")
```

- The `else` block **does not execute** because `break` triggered an early exit when `x` became `3`.

7. Nested Loops

A **nested loop** is one loop inside another. The **inner loop** completes its iterations for each iteration of the **outer loop**.

```python
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

- For each adjective in `adj`, the inner loop runs through every fruit in `fruits`.

8. The `pass` Statement

In Python, a loop **cannot be empty**. Use `pass` if you need a loop with no content to avoid an error.

```python
for x in [0, 1, 2]:
    pass
```

- Here, the loop does nothing but remains syntactically valid.

9. Summary and Best Practices

1. **Iterate Over Any Iterable**
   - Lists, strings, tuples, sets, dictionaries (by keys), or even custom iterables.
2. **Use `break` / `continue` Wisely**
   - `break`: Stop the loop entirely.
   - `continue`: Skip only the current iteration.
3. **`else` Block**

- Great for detecting a loop’s “normal completion” (no `break`).

4. **Nested Loops**
    - Powerful but can be less efficient if used excessively aim for clarity.

5. **`range()`**
    - Perfect for looping a specific number of times or creating arithmetic progressions.

With these foundations, you can handle tasks ranging from simple iterations to more advanced data-processing scenarios. The **`for` loop** in Python is both **flexible** and **reader-friendly**, making it a key construct for anyone coding in Python.