Python Loops – `while` and `for`

Python provides two main types of loops:

1. **`while` Loop** – Executes as long as a condition is `True`.
2. **`for` Loop** – Iterates over a sequence (list, tuple, string, range, etc.).

Both loops allow you to execute a block of code multiple times, but they are used in different scenarios.

1. `while` Loop in Python

A `while` loop runs as long as a given condition remains `True`. It is typically used when the number of iterations is **unknown beforehand**.

Syntax:

```
while condition:
    # Code block to execute
```

- The `condition` is evaluated before each iteration.
- If the condition is `False` at the start, the loop will not execute at all.
- **Important:** If the condition never becomes `False`, an **infinite loop** occurs.

Example 1: Basic `while` Loop

In [1]:
```
count = 1
while count <= 5:
    print("Count:", count)
    count += 1  # Incrementing count to avoid infinite loop
```

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Example 2: User Input Validation Using `while`

A `while` loop is useful when we want to repeatedly ask the user for input until they provide a valid response.

In [2]:
```
password = ""
while password != "Python123":
    password = input("Enter the correct password: ")

print("Access Granted!")
```

```
Access Granted!
```

- The loop runs **until** the user enters `"Python123"`.
- If the correct password is entered on the first attempt, the loop runs only once.

Example 3: Using `while` Loop for Counting Down

In [3]:
```python
num = 5
while num > 0:
    print("Countdown:", num)
    num -= 1
print("Liftoff!")
```

```
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Liftoff!
```

- The loop decreases `num` until it reaches 0.
- When `num` becomes 0, the loop stops.

Special Cases in `while` Loops

1. Infinite Loop

A `while` loop that **never ends** due to a missing update in the condition.

```python
while True:
    print("This will run forever!")
```

- Press `Ctrl + C` to manually stop it.
- To **avoid infinite loops**, always ensure there is a way for the condition to become `False`.

2. Using `break` to Exit the Loop

In [4]:
```python
count = 1
while count <= 5:
    if count == 3:
        print("Breaking at count =", count)
        break  # Exits the loop
    print("Count:", count)
    count += 1
```

```
Count: 1
Count: 2
Breaking at count = 3
```

- The `break` statement **immediately terminates** the loop when `count == 3`.

3. Using `continue` to Skip an Iteration

In [5]:
```
count = 0
while count < 5:
    count += 1
    if count == 3:
        print("Skipping count =", count)
        continue  # Skips the rest of the loop body
    print("Count:", count)
```

```
Count: 1
Count: 2
Skipping count = 3
Count: 4
Count: 5
```

- When `count == 3`, `continue` **skips the rest of the loop body** and moves to the next iteration.

4. Using `pass` in a `while` Loop

In [6]:
```
x = 5
while x > 0:
    if x == 3:
        pass  # Placeholder for future logic
    else:
        print("Value of x:", x)
    x -= 1
```

```
Value of x: 5
Value of x: 4
Value of x: 2
Value of x: 1
```

- When `x == 3`, the `pass` statement **does nothing**, so nothing prints.

5. Using `else` with `while`

In [7]:
```
count = 1
while count < 4:
    print("Count:", count)
    count += 1
else:
    print("Loop ended naturally")
```

```
Count: 1
Count: 2
Count: 3
Loop ended naturally
```

- The `else` block executes **only if** the loop **completes without** `break`.

2. `for` Loop in Python

A `for` loop is **used to iterate over a sequence** (list, tuple, string, range, dictionary, etc.).
It is generally used when the number of iterations is **known beforehand**.

Syntax:

```
for variable in sequence:
    # Code block to execute
```

- The `variable` takes values from the `sequence`, **one at a time**.
- The loop **automatically stops** when all elements have been processed.

Example 1: Iterating Over a List

In [8]:
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

- The loop runs **once for each element** in the list.

Example 2: Using `range()` in `for` Loop

In [9]:
```
for num in range(1, 6):
    print(num)
```

```
1
2
3
4
5
```

- The `range(1, 6)` generates numbers from **1 to 5** (`stop` is exclusive).

Example 3: Iterating Over a String

In [10]:
```
for char in "Python":
    print(char)
```

```
P
y
t
h
o
n
```

- The loop prints each character of the string.

Special Cases in `for` Loops

1. Using `break` to Exit the Loop

In [11]:
```
for num in range(1, 6):
    if num == 3:
        print("Breaking at:", num)
        break
    print("Number:", num)
```

```
Number: 1
Number: 2
Breaking at: 3
```

- The loop **stops immediately** when `num == 3`.

2. Using `continue` to Skip an Iteration

In [12]:
```
for num in range(1, 6):
    if num == 3:
        print("Skipping", num)
        continue
    print("Number:", num)
```

```
Number: 1
Number: 2
Skipping 3
Number: 4
Number: 5
```

- The `continue` statement **skips** printing when `num == 3`.

3. Using `pass` in a `for` Loop

In [13]:
```
for i in range(5):
    if i == 2:
        pass  # This does nothing but avoids syntax errors
    else:
        print("Number:", i)
```

```
Number: 0
Number: 1
Number: 3
Number: 4
```

- At `i == 2`, the `pass` statement **executes but does nothing**, so `2` is skipped.

4. Using `else` with `for`

In [14]:
```
for num in range(1, 4):
    print("Number:", num)
```

```
else:
    print("Loop finished without break")
```

```
Number: 1
Number: 2
Number: 3
Loop finished without break
```

- The `else` block runs **only if** the loop **completes without** `break`.

3. Key Differences Between `while` and `for` Loops

Feature	`while` Loop	`for` Loop
Condition-based	Yes	No
Iterates over sequences	No	Yes
Uses `break` and `continue`	Yes	Yes
Suitable when number of iterations is unknown	Yes	No
Suitable when number of iterations is known	No	Yes

Conclusion

- `while` loops are **condition-based** and are useful for **unknown iteration counts**.
- `for` loops are **sequence-based** and are useful when **iterating over known sequences**.
- Both loops can use `break`, `continue`, and `else` for better control.

Nested Loops in Python

A **nested loop** is a loop inside another loop. The **inner loop** executes **completely** for each iteration of the **outer loop**.

Syntax of a Nested Loop:

```
for outer_variable in outer_sequence:
    for inner_variable in inner_sequence:
        # Code block executed in the inner loop
```

or

```
while condition1:
    while condition2:
        # Code block executed in the inner loop
```

- The **outer loop** runs first.
- The **inner loop** completes all its iterations for each **single iteration** of the outer loop.

Example 1: Nested `for` Loop (Multiplication Table)

In [15]:
```python
for i in range(1, 4):  # Outer loop
    for j in range(1, 4):  # Inner loop
        print(f"{i} x {j} = {i * j}")
    print("------")  # Separates tables
```

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
------
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
------
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
------
```

- The inner loop (`j`) runs **completely** for each **iteration** of the outer loop (`i`).
- This creates multiplication tables for `1, 2, and 3` .

Example 2: Nested `while` Loop

In [16]:
```python
i = 1
while i <= 3:  # Outer loop
    j = 1
    while j <= 3:  # Inner loop
        print(f"i={i}, j={j}")
        j += 1
    i += 1
    print("------")  # Separates iterations
```

```
i=1, j=1
i=1, j=2
i=1, j=3
------
i=2, j=1
i=2, j=2
i=2, j=3
------
i=3, j=1
i=3, j=2
i=3, j=3
------
```

- The **inner** `while` **loop** completes before the **outer** `while` **loop** moves to the next value.

Example 3: Printing Patterns Using Nested Loops

In [17]:
```
for i in range(1, 6):  # Outer loop controls rows
    for j in range(i):  # Inner loop controls columns
        print("*", end=" ")  # Print * on the same line
    print()  # Move to the next line
```

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

- The number of `*` in each row **increases** with `i`.
- The **inner loop prints stars**, and the **outer loop moves to the next row**.

Special Cases in Nested Loops

1. Using `break` in a Nested Loop

In [18]:
```
for i in range(1, 4):
    for j in range(1, 4):
        if j == 2:
            break  # Exits only the inner loop
        print(f"i={i}, j={j}")
```

```
i=1, j=1
i=2, j=1
i=3, j=1
```

- The `break` **only exits** the inner loop (`j`), but the outer loop (`i`) continues.

2. Using `continue` in a Nested Loop

In [19]:
```
for i in range(1, 4):
    for j in range(1, 4):
        if j == 2:
            continue  # Skips the rest of this iteration
        print(f"i={i}, j={j}")
```

```
i=1, j=1
i=1, j=3
i=2, j=1
i=2, j=3
i=3, j=1
i=3, j=3
```

- The **inner loop skips** `j = 2` but continues for other values.

3. Using `pass` in a Nested Loop

In [20]:
```
for i in range(1, 4):
    for j in range(1, 4):
```

```
        if i == j:
            pass  # Placeholder
        else:
            print(f"i={i}, j={j}")
```

```
i=1, j=2
i=1, j=3
i=2, j=1
i=2, j=3
i=3, j=1
i=3, j=2
```

- The `pass` **does nothing** when `i == j`, effectively skipping it.

Key Takeaways for Nested Loops

✅ The **inner loop runs completely** for each iteration of the outer loop.
✅ **Break** exits only the inner loop unless used in both loops.
✅ **Continue** skips the current iteration of the inner loop.
✅ **Pass** is used as a placeholder for future logic.