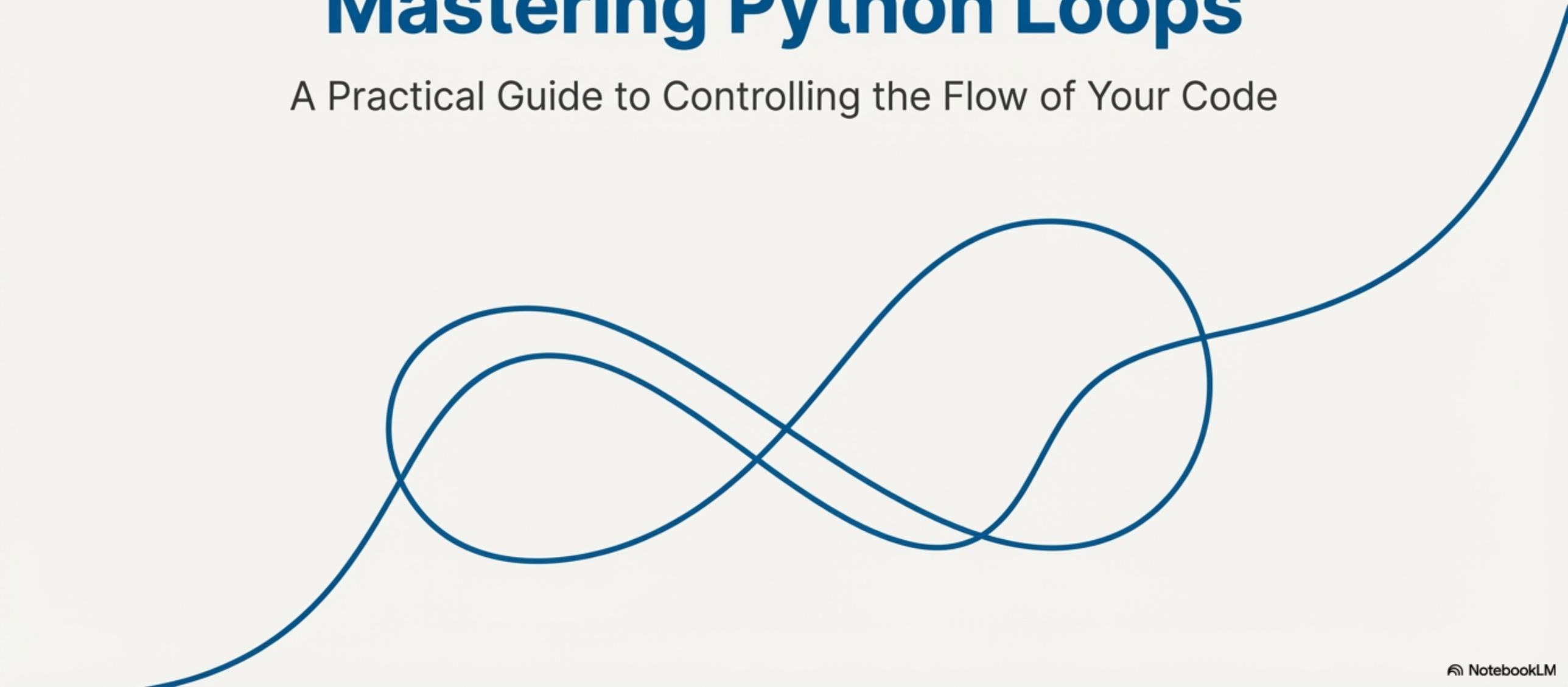
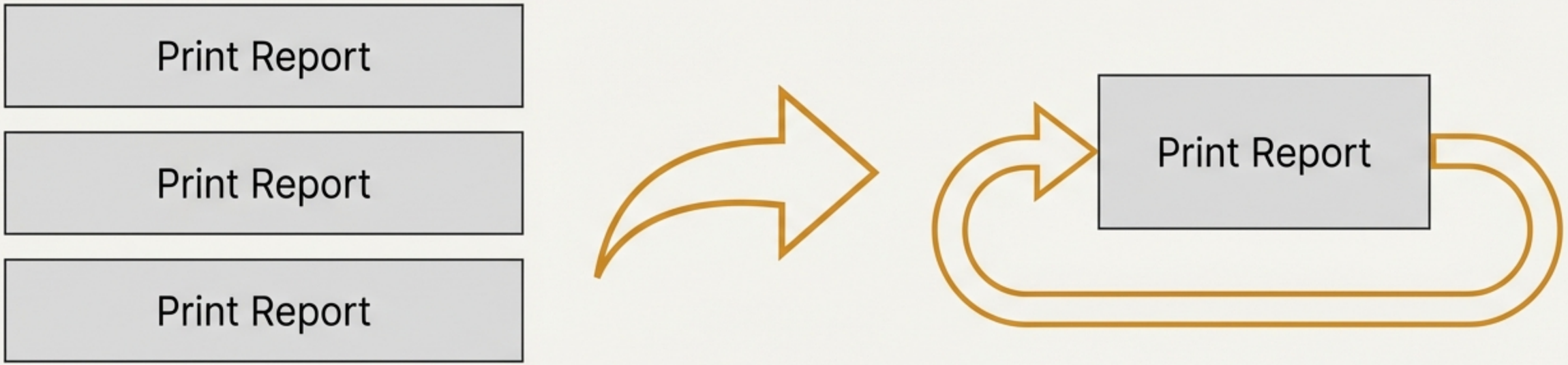
Mastering Python Loops

A Practical Guide to Controlling the Flow of Your Code



Why We Need Loops: The Principle of Automation

At its heart, a loop is a tool for automating repetitive tasks. When you need to execute the same block of code multiple times, a loop is the clean and efficient solution.



The **for loop**: Used when you want to iterate over a finite sequence of items.

The **while loop**: Used when you want to repeat a block of code as long as a certain condition remains true.

The for Loop: A Tour Through Any Sequence

Why

Use a for loop when you have a known collection of items (like a list, a string, or a range of numbers) and **you want** to perform an action for each item in that collection.

What (Syntax)

```
for variable in iterable:
    # statement(s) to execute
```

How (Example)

```
# Iterating over each character in a string
for character in "Python":
    print(character)
```

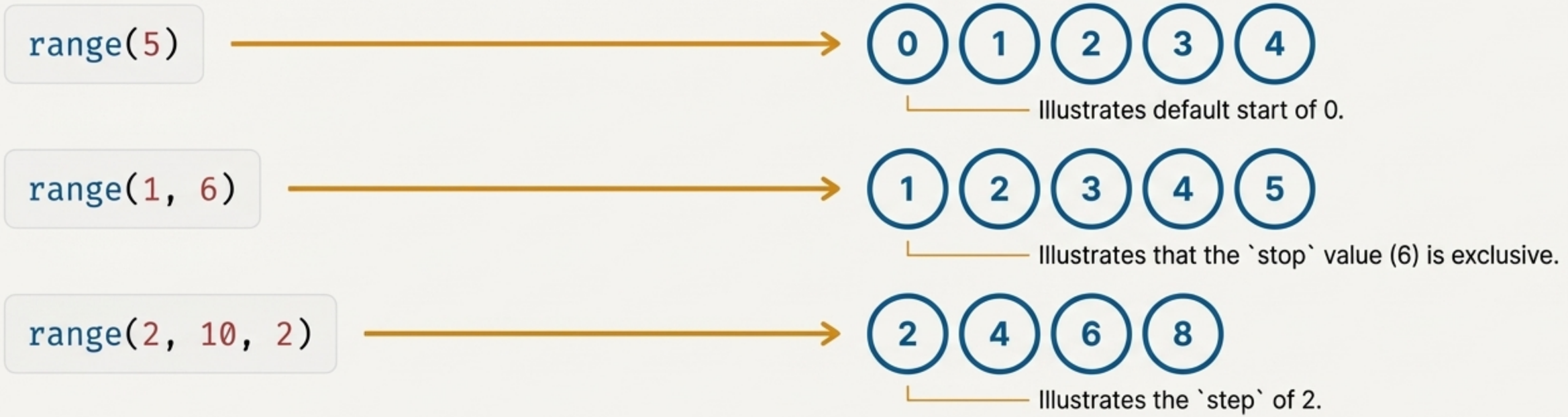
```
P
y
t
h
o
n
```

The `range()` Function: Your Guide to Generating Sequences

The `range()` function generates a sequence of numbers, which is perfect for running a `for` loop a specific number of times.

```
range(start, stop, step)
```

- `start` (Optional): The first number in the sequence. Defaults to 0.
- `stop` (**Required**): The sequence is generated up to, but not including, this number.
- `step` (Optional): The increment between each number. Defaults to 1.



The `while` Loop: Repeating Until a Condition Changes

Why

Use a `while` loop when you need to repeat a block of code for an unknown number of times, **continuing** as long as a specific condition is met.

What (Syntax)

```
while condition:
    # statement(s) to execute
```

How (Example)

```
count = 1  # 1. Initialise the counter
while count <= 5: # 2. Check the condition
    print(count)
    count += 1 # 3. IMPORTANT: Update the counter
```

```
1
2
3
4
5
```

⚠ Be mindful of infinite loops! If the condition never becomes false, the loop will run forever. For instance, `while True:`.

`for` vs. `while`: Choosing the Right Tool for the Job

The `for` Loop

Best for: Definite Iteration

Core idea: “Do this for every item in this collection.”

Iterating over a list, string, or a fixed number of times using `range()`.

```
# Iterate over a known list
for item in [10, 20, 30]:
    print(item)
```

The `while` Loop

Best for: Indefinite Iteration

Core idea: “Keep doing this as long as this condition is true.”

Waiting for user input, processing a data stream until an end signal is received.

```
# Repeat until condition is met
count = 1
while count <= 3:
    print(count)
    count += 1
```

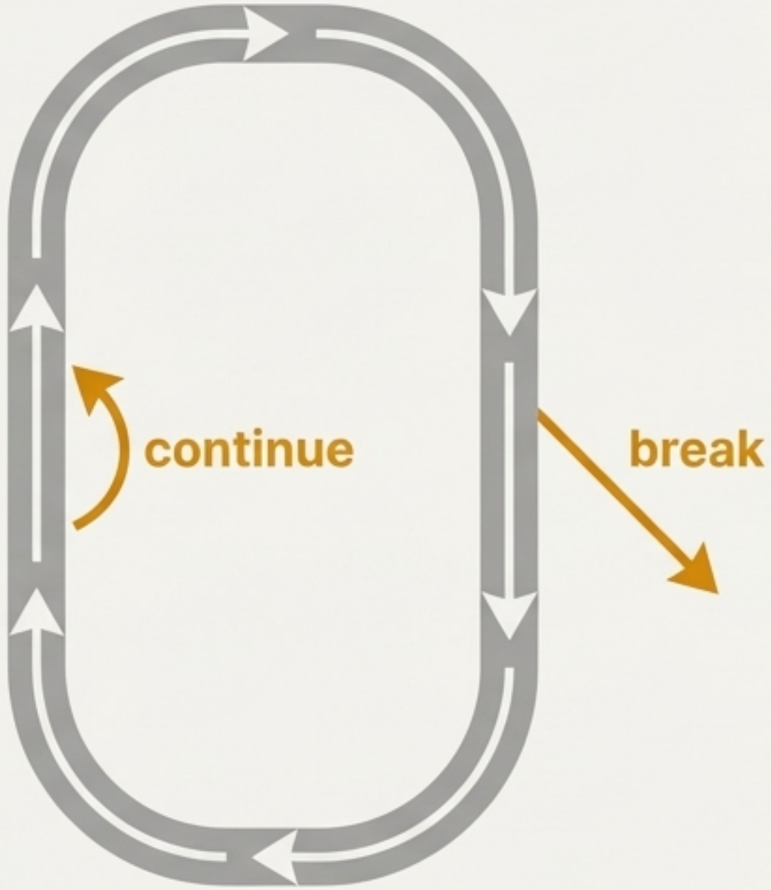
Taking Command of the Flow

Beyond Simple Repetition: An Introduction to Loop Control

Python provides powerful statements to alter the standard flow of a loop. These tools allow you to **exit early**, skip specific iterations, or run code only when a loop completes successfully.

What We'll Cover

- **break**: For an immediate exit.
- **continue**: To skip the current iteration.
- **else**: To execute code after a normal loop conclusion.



The `break` Statement: Making an Early Exit

Why

Use `break` to immediately terminate a loop from within, regardless of the loop's original condition or remaining items in the sequence. It's often used when a specific condition is met.

What

The `break` keyword is used inside a conditional statement (like `if`) within the loop.

How (Example)

```
for i in range(10):
    if i == 5:
        # When i is 5, exit the loop completely
        break
    print(i)
```

```
0
1
2
3
4
```

The loop stops at 5 and does not continue to 9.

The `continue` Statement: Skipping to the Next Iteration

Why

Use `continue` to stop the current iteration and immediately jump to the beginning of the next one. The loop itself does not terminate.

What

The `continue` keyword is used inside a conditional statement within the loop.

How (Example)

```
for i in range(5):
    if i == 2:
        # When i is 2, skip the print statement
        # and go to the next iteration (i=3)
        continue
    print(i)
```

```
0
1
3
4
```

The iteration for `i == 2` was skipped.

`break` vs. `continue`: A Side-by-Side Comparison

`break`

Action: Exits the Loop

Effect: The entire loop is terminated immediately. Execution continues with the first statement *after* the loop.

```
for i in range(5):
    if i == 2:
        break
    print(i)
```

Output:

```
0
1
```

`continue`

Action: Skips an Iteration

Effect: The current iteration is abandoned. Execution jumps to the top of the loop for the next iteration.

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Output:

```
0
1
3
4
```

The Loop `else` Clause: Code That Runs on Normal Completion

Why

The `else` block provides a way to execute a piece of code only if the loop completes its entire sequence without being terminated by a `break` statement.

How it Works

- If the loop finishes naturally (all items in the `for` loop are processed, or the `while` condition becomes false), the `else` block is executed.
- If the loop is exited via a `break` statement, the `else` block is skipped.

Example

```
for i in range(3):
    print(i)
else:
    # This runs because the loop was not 'broken'
    print("Loop completed normally.")
```

```
0
1
2
Loop completed normally.
```

Patterns and Applications

Nested Loops: Placing One Loop Inside Another

Why

Nested loops are essential for working with two-dimensional data structures (like grids or matrices) or when you need to perform a repetitive action for each item of another repetitive action.

How it Works

For each single iteration of the “outer” loop, the “inner” loop will complete its entire sequence of iterations.

Example

Outer loop: i = 0, 1, 2

```
# The outer loop runs 3 times
for i in range(3):
    # The inner loop runs 2 times for each
    # outer loop iteration
    for j in range(2):
        print(f"i = {i}, j = {j}")
```

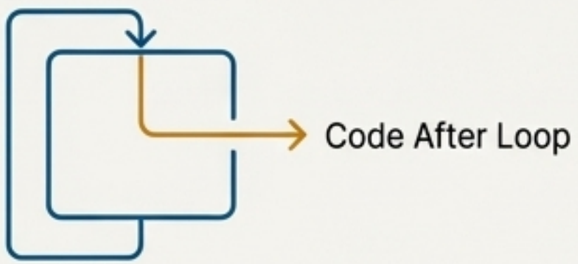
Inner loop: j = 0, 1 for each i

```
i = 0, j = 0
i = 0, j = 1

i = 1, j = 0
i = 1, j = 1

i = 2, j = 0
i = 2, j = 1
```

Loop Control Statements: A Quick Reference Guide

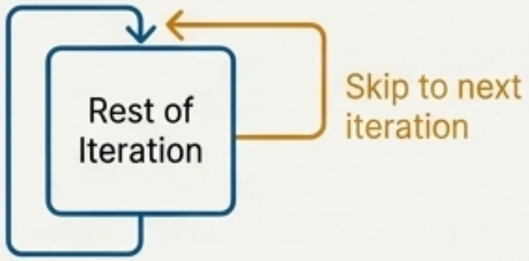
`break`

- **Purpose:** Exit the loop entirely.
- **When to Use:** When a search is successful or a critical error condition is met.
- **Flow:** Execution jumps to the code immediately following the loop.



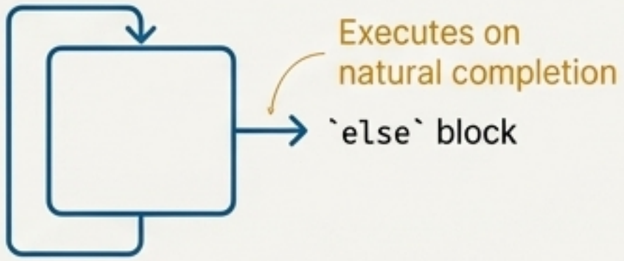
`continue`

- **Purpose:** Skip the rest of the current iteration.
- **When to Use:** When you want to ignore certain items in a sequence but continue processing the rest.
- **Flow:** Execution jumps back to the start of the loop for the next iteration.



`else`

- **Purpose:** Run a block of code after the loop finishes.
- **When to Use:** To perform a "clean-up" action or confirm that a loop ran to completion without being broken.
- **Flow:** Executes only if the loop completes naturally (no `break`).



Practical Pattern 1: Aggregating Data

A common task in data analysis is to calculate a sum or total from a list of numbers. A `for` loop is the perfect tool for this.

Code Logic

Step 1. Initialise an accumulator variable

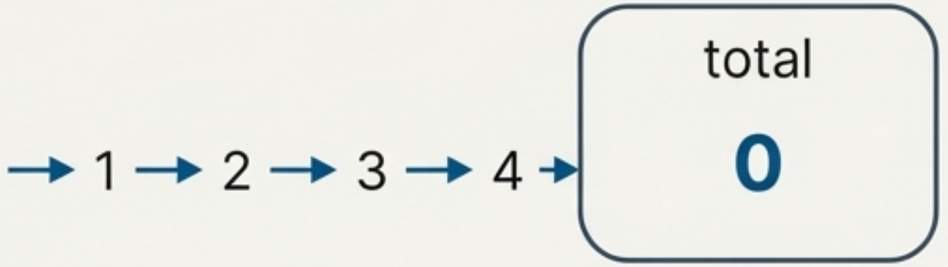
```
total = 0
numbers = [1, 2, 3, 4]
```

Step 2. Loop through the list, adding each number

```
for n in numbers:
    total += n
```

Step 3. Display the final result

```
print(f"The final total is: {total}")
```




```
numbers = [1, 2, 3, 4]
total = 0
for n in numbers:
    total += n
print(total)
```

10

Practical Pattern 2: Building Structures

Scenario: Loops are not just for processing data; they are fundamental for generating structured output, from formatted text reports to simple graphics.

Using a `for` loop and the `range()` function to print a star pattern. This demonstrates programmatic control over structure.

```
# Loop from 1 to 5
for i in range(1, 6):
    # Print the '*' character 'i' times
    print("*" * i)
```

This simple pattern showcases how iteration gives you precise control to build complex results from simple rules.