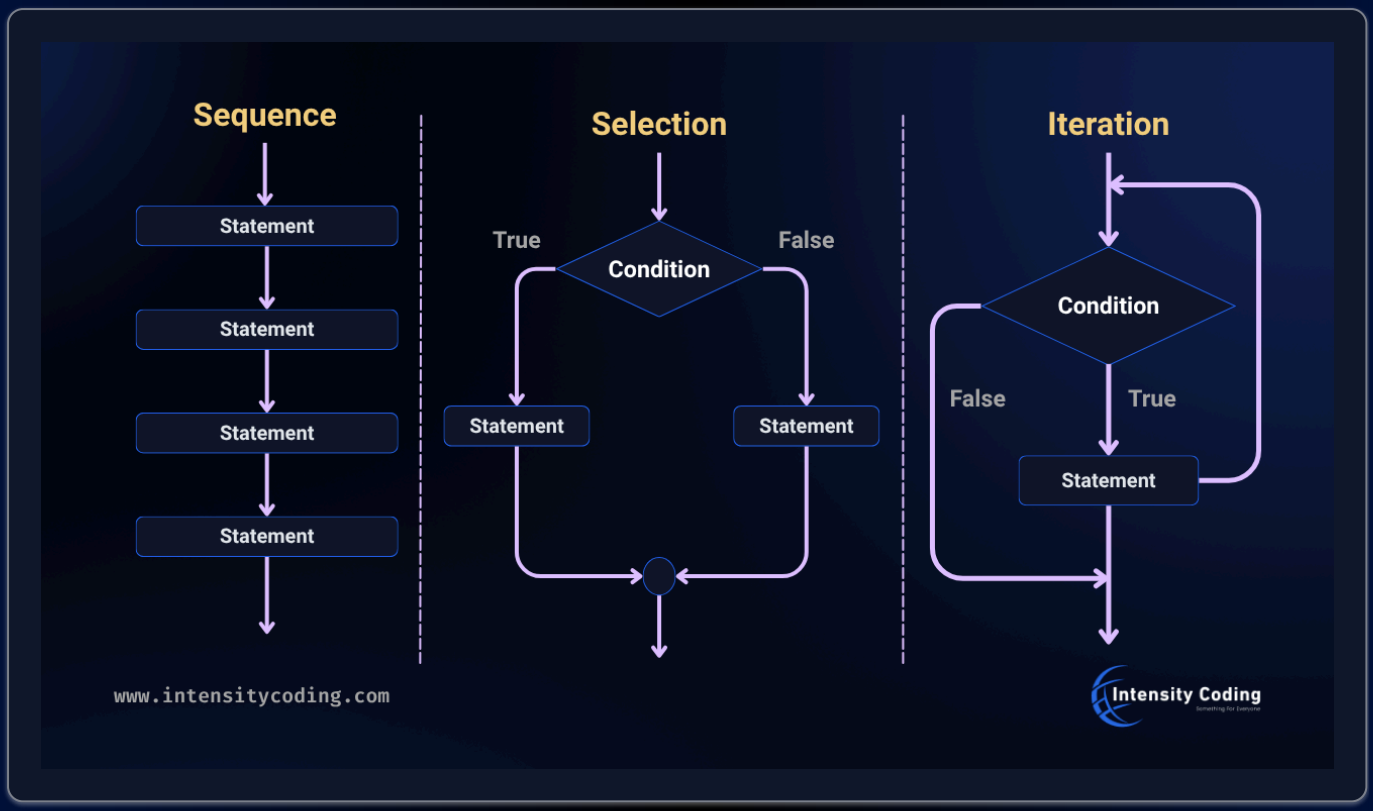
PYTHON

Python Control Statements

Python

Control Statements in Python

- Control statements determine the flow of execution in a program. They allow us to **manage the order in which statements are executed**, enabling decision-making, repetition, and sequential execution.
- There are three primary types of control flow:
    1. **Sequential Execution** – Executes statements in the order they are written.
    2. **Selection Control (Decision Making)** – Executes different statements based on conditions.
    3. **Iteration (Looping)** – Repeats a block of code multiple times.



1. Sequential Control

- **Definition**: Statements execute **one after another** in a linear fashion.
- **Behavior**: No branching or repetition occurs.

```
# Printing steps in sequential order
print("Step 1")  # Executes first
print("Step 2")  # Executes second
print("Step 3")  # Executes third
```

```
Step 1
Step 2
Step 3
```

2. Selection Control (Decision Making)

- **Definition**: The program **chooses a path** based on conditions.

- **Behavior**: Executes different blocks of code depending on conditions (e.g., `if-else`, `Match`).

```
# Define an age variable
age = 18

# Check if age is 18 or above
if age >= 18:
    print("You are eligible to vote")  # Executes if condition is True
else:
    print("You are not eligible to vote")  # Executes if condition is
False
```

```
You are eligible to vote
```

3. Iteration Control (Looping)

- **Definition**: Repeats a block of code **until a condition is met**.
- **Behavior**: Executes loops (`for`, `while`).

```
# Initialize counter variable
i = 1

while i <= 5:
    print("Iteration:", i)  # Print the current iteration number
    i += 1  # Increment i by 1 to avoid infinite loop
```

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
```

Python If...Else Statements

- Python provides conditional statements to control the flow of execution based on specific conditions.
- Python supports the following logical conditions for comparisons:
  - **Equals**: `a == b`
  - **Not Equals**: `a != b`
  - **Less than**: `a < b`
  - **Less than or equal to**: `a <= b`
  - **Greater than**: `a > b`
  - **Greater than or equal to**: `a >= b`

If Statements

- In Python, the `if` statement is the most basic form. It evaluates a given condition and checks whether it is True or False.
  - If the condition evaluates to True, the block of code under the `if` statement executes.
  - If the condition evaluates to False, that block is skipped, and the program continues with the next line of code following the `if` block.

Syntax:

```
if condition:
    statement_1
    statement_2
    ...
    statement_n
```

```
views = 5000
threshold = 10000

# If views are greater than the threshold, print a message
if views > threshold:
```

```
    print("Intensity Coding has reached a milestone!")

# Output: (No output because condition is False)
```

Indentation in If Statements

- Python uses indentation to define code blocks. Incorrect indentation leads to errors.

```
if views > threshold:
print("Intensity Coding has reached a milestone!")  # IndentationError
```

The `elif` Statement

- The `elif` keyword is used to check additional conditions when the previous `if` statement is false.

Syntax:

```
if condition:
    statement_1
elif another_condition:
    statement_2
```

```
subscribers = 10000
goal = 10000

if subscribers > goal:
    print("Intensity Coding has exceeded its subscriber goal!")
elif subscribers == goal:
    print("Intensity Coding has reached its subscriber goal!")

# Output:
# Intensity Coding has reached its subscriber goal!
```

```
Intensity Coding has reached its subscriber goal!
```

The `else` Statement

- The `else` clause is used to define a block of code that executes only when all previous conditions evaluate to `False`. It acts as a fallback option when none of the specified conditions are satisfied.

Syntax:

```
if condition:
    statement_1
elif another_condition:
    statement_2
else:
    statement_3
```

```
likes = 150
required_likes = 200

if likes > required_likes:
    print("Intensity Coding video is trending!")
elif likes == required_likes:
    print("Intensity Coding video is gaining attention!")
else:
    print("More likes are needed for trending!")

# Output:
# More likes are needed for trending!
```

```
More likes are needed for trending!
```

Syntax of the if-else statement

- If there is no need for an `elif`, you can use only `else`:

```
if condition:
    statement 1
```

```
else:
    statement 2
```

```
views = 3000
required_views = 5000

if views > required_views:
    print("Intensity Coding tutorial is getting popular!")
else:
    print("Keep sharing the tutorial to reach more people!")

# Output:
# Keep sharing the tutorial to reach more people!
```

```
Keep sharing the tutorial to reach more people!
```

Short-Hand If Statement

- When an `if` statement has only one line, it can be written in a single line.

```
if views > required_views: print("Tutorial is going viral!")
# Output: (No output because condition is False)
```

Short-Hand If...Else (Ternary Operator)

- For concise conditional expressions, Python allows a short-hand format.

```
subscribers = 5000
goal = 10000

print("Goal Achieved!") if subscribers ≥ goal else print("Keep
Growing!")
```

```
# Output:
# Keep Growing!
```

```
Keep Growing!
```

```
# You can also use multiple conditions in one line:
subscribers = 10000
goal = 10000

print("Above Goal!") if subscribers > goal else print("On Target!") if
subscribers == goal else print("Keep Pushing!")
# Output:
# On Target!
```

```
On Target!
```

Logical Operators (`and`, `or`, `not`)

Using `and` Operator

- Both conditions must be `True` for the block to execute.

```
likes = 500
comments = 50
shares = 20

if likes > 400 and comments > 30:
    print("Intensity Coding tutorial is getting engagement!")

# Output:
# Intensity Coding tutorial is getting engagement!
```

```
Intensity Coding tutorial is getting engagement!
```

Using `or` Operator

- At least one condition must be `True` for execution.

```python
if likes > 400 or shares > 50:
    print("Tutorial is gaining visibility!")

# Output:
# Tutorial is gaining visibility!
```

```
Tutorial is gaining visibility!
```

Using `not` Operator

- Reverses the result of a condition.

```python
subscribers = 5000
goal = 10000

if not subscribers > goal:
    print("Subscribers goal not reached yet.")

# Output:
# Subscribers goal not reached yet.
```

```
Subscribers goal not reached yet.
```

Nested If Statements

- An `if` statement inside another `if` statement is called a **nested if**. It allows the program to check multiple levels of conditions sequentially.

Syntax:

```
if condition_outer:
    if condition_inner:
        # Executed when both outer and inner conditions are True
        statement_of_inner_if
    else:
        # Executed when outer condition is True but inner condition is False
        statement_of_inner_else
    # Executed when the outer condition is True (after inner block completes)
    statement_of_outer_if
else:
    # Executed when the outer condition is False
    statement_of_outer_else

# Statement outside all if-else blocks
statement_outside_if_block
```

```
views = 12000

if views > 1000:
    print("Intensity Coding tutorial is getting views!")  # Output:
Intensity Coding tutorial is getting views!

    if views > 10000:
        print("Tutorial is becoming popular!")  # Output: Tutorial is
becoming popular!
    else:
        print("Keep sharing the tutorial.")

# Output:
# Intensity Coding tutorial is getting views!
# Tutorial is becoming popular!
```

```
Intensity Coding tutorial is getting views!
Tutorial is becoming popular!
```

The `pass` Statement

- Python does not allow empty `if` statements. However, you can use the `pass` keyword if an `if` block is required but should not execute any code.

```
subscribers = 10000

if subscribers > 5000:
    pass  # Placeholder for future logic
```

Python match Statement

- Python introduced the powerful match statement in version 3.10, bringing pattern matching capabilities similar to switch-case in languages like C, Java, or JavaScript — but far more expressive and Pythonic.
- The match statement allows you to compare a value (called the subject) against various patterns, executing code based on the first successful match. This provides a more readable and scalable alternative to lengthy if...elif...else chains.

Syntax

```
match subject:
    case pattern1:
        # Executes if subject matches pattern1
    case pattern2:
        # Executes if subject matches pattern2
    case _:
        # Default fallback if no other pattern matches
```

How `match` Works

- The subject is evaluated once.
- Patterns are checked top-down.

- The first matching case block is executed.
- The _ is a wildcard pattern, similar to else.

Pattern Matching Use Cases

Use Case	Description
Literal patterns	Matches exact values like numbers or strings
OR patterns	Combines multiple literals using |
Variable binding	Captures parts of the subject into variables
Sequence patterns	Matches list or tuple structures
Mapping patterns	Matches dictionaries and extracts values
Class patterns	Matches class instances and extracts attributes
Guards	Adds if conditions to refine a match

Basic Matching Example

```
model_name = "gpt4"

match model_name:
    case "bert":
        print("Selected Model: BERT Base")
    case "gpt3":
        print("Selected Model: GPT-3")
    case "gpt4":
        print("Selected Model: GPT-4 by OpenAI")
    case _:
        print("Unknown model code")
# Output: Selected Model: GPT-4 by OpenAI
```

```
Selected Model: GPT-4 by OpenAI
```

Using _ as a Default Fallback

```
# Example: Match task type with default fallback
task_type = "summarization"

match task_type:
    case "classification":
        print("Running a classification task")
    case "regression":
        print("Running a regression task")
    case _:
        print("Task not supported yet")

# Output: Task not supported yet
```

```
Task not supported yet
```

Match Multiple Conditions Using |

```
# Example: Handle multiple text preprocessing tasks
task = "tokenization"

match task:
    case "cleaning" | "normalization" | "tokenization":
        print("Executing basic NLP preprocessing")
    case "translation" | "summarization":
        print("Executing high-level NLP task")

# Output: Executing basic NLP preprocessing
```

```
Executing basic NLP preprocessing
```

Use Guards (`if`) to Add Conditions

```python
# Example: Handling datasets based on name and version
dataset = "MNIST"
version = 2

match dataset:
    case "MNIST" if version == 1:
        print("Loading MNIST - Original version")
    case "MNIST" if version == 2:
        print("Loading MNIST - Augmented version")
    case "CIFAR":
        print("Loading CIFAR dataset")
    case _:
        print("Unknown dataset")

# Output: Loading MNIST - Augmented version
```

```
Loading MNIST - Augmented version
```

Matching Tuples (Sequence Pattern)

```python
point = (0, 5)

match point:
    case (0, 0):
        print("Point at origin")
    case (0, y):
        print(f"Point on Y-axis at y={y}")
    case (x, 0):
        print(f"Point on X-axis at x={x}")
    case (x, y):
        print(f"Point at coordinates ({x}, {y})")
    case _:
        print("Unknown structure")

# Output: Point on Y-axis at y=5
```

```
Point on Y-axis at y=5
```

Matching Class Instances

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def locate_point(p):
    match p:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"On Y-axis at y={y}")
        case Point(x=x, y=0):
            print(f"On X-axis at x={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a Point instance")
```

```
locate_point(Point(0, 7))
# Output: On Y-axis at y=7
```

```
On Y-axis at y=7
```

Sequence Matching with Extended Unpacking

```
data = [1, 2, 3, 4]

match data:
    case [x, y, *rest]:
        print(f"First: {x}, Second: {y}, Rest: {rest}")

# Output: First: 1, Second: 2, Rest: [3, 4]
```

```
First: 1, Second: 2, Rest: [3, 4]
```

Matching Dictionaries (Mapping Pattern)

```
config = {"bandwidth": 100, "latency": 50}

match config:
    case {"bandwidth": bw, "latency": lt}:
        print(f"Bandwidth: {bw}, Latency: {lt}")

# Output: Bandwidth: 100, Latency: 50
```

```
Bandwidth: 100, Latency: 50
```

Nested Patterns with `as`

```python
class Point:
    __match_args__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y

points = [Point(0, 0), Point(1, 1)]

match points:
    case [Point(0, 0), Point(x, y) as second_point]:
        print(f"Second point is at ({x}, {y})")
        print(f"Captured as object: {second_point}")

# Output:
# Second point is at (1, 1)
# Captured as object: <__main__.Point object at ...>
```

```
Second point is at (1, 1)
Captured as object: <__main__.Point object at 0×7ecba94ce910>
```

Python Loops – while and for

- Loops in Python allow executing a block of code multiple times. Python provides two primary loop types:
  - `while` **loop** – Executes as long as a specified condition is `True`.
  - `for` **loop** – Iterates over a sequence such as a list, tuple, dictionary, or string.

Python While Loops

while Loop

- The `while` loop executes a block of code as long as a specified condition evaluates to `True`.

Syntax:

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

- In a `while` **loop**, the condition is evaluated **before** each iteration begins. If the condition evaluates to **True**, the block of code inside the loop executes. Once the condition becomes **False**, the loop terminates, and the program continues with the next statement following the loop.

```
# Using a while loop to print numbers from 1 to 5
i = 1
while i <= 5:
    print(i)  # Output: 1, 2, 3, 4, 5
    i += 1  # Increment to prevent an infinite loop
```

```
1
2
3
4
5
```

Infinite Loop

- A `while` loop without a stopping condition runs indefinitely.

```
# WARNING: This is an infinite loop! Uncomment with caution.
# while True:
#     print("This will run forever!")
```

break Statement

- The `break` statement stops the loop even if the condition remains `True`.

```
# Stop the loop when i equals 3
i = 1
while i ⩽ 5:
    print(i)  # Output: 1, 2, 3
    if i == 3:
        break  # Exit the loop
    i += 1
```

```
1
2
3
```

`continue` Statement

- The `continue` statement skips the current iteration and moves to the next.

```
# Skip printing 3
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue  # Skip when i equals 3
    print(i)  # Output: 1, 2, 4, 5
```

```
1
2
4
5
```

`else` Statement

- The `else` block runs after the `while` loop finishes execution.

- If the loop is terminated prematurely by either a break or return statement, the else clause won't execute at all.

```
# Print numbers and display a message after completion
i = 1
while i <= 5:
    print(i)  # Output: 1, 2, 3, 4, 5
    i += 1
else:
    print("Loop completed!")  # Output: Loop completed!
```

```
1
2
3
4
5
Loop completed!
```

Nested while Loops

- A nested while loop is a loop inside another loop. The outer loop runs first, and for each iteration of the outer loop, the inner loop executes completely.

Key Points

- The outer loop controls the main iteration.
- The inner loop runs completely for each iteration of the outer loop.
- Both loops require proper exit conditions to avoid infinite loops.

Syntax

```
while condition_outer:
    # Outer loop body
    while condition_inner:
        # Inner loop body
```

```python
# Nested while loop example : Multiplication Table
i = 1  # Outer loop counter

while i <= 5:  # Outer loop for numbers 1 to 5
    j = 1  # Inner loop counter
    while j <= 5:  # Inner loop for multiplication
        print(f"{i} x {j} = {i*j}", end="\t")
        j += 1
    print()  # New line after inner loop
    i += 1
```

```
1 x 1 = 1       1 x 2 = 2       1 x 3 = 3       1 x 4 = 4       1 x 5
= 5
2 x 1 = 2       2 x 2 = 4       2 x 3 = 6       2 x 4 = 8       2 x 5
= 10
3 x 1 = 3       3 x 2 = 6       3 x 3 = 9       3 x 4 = 12      3 x 5
= 15
4 x 1 = 4       4 x 2 = 8       4 x 3 = 12      4 x 4 = 16      4 x 5
= 20
5 x 1 = 5       5 x 2 = 10      5 x 3 = 15      5 x 4 = 20      5 x 5
= 25
```

Nested while Loops with break

- Break : Immediately exits the current loop (inner loop in nested cases).
- Using break in the inner loop does not affect the outer loop except in logic flow.

Syntax with break

```python
while condition_outer:
    # Outer loop body
    while condition_inner:
        if some_condition:
            break  # Exit inner loop immediately
```

```python
# Nested while loop with break
i = 1  # Outer loop counter

while i <= 5:  # Outer loop
    j = 1  # Inner loop counter
    while j <= 5:  # Inner loop
        if j == 4:  # Break inner loop when j is 4
            break
        print(f"{i} x {j} = {i*j}", end="\t")
        j += 1
    print()
    i += 1
# Here, the inner loop stops at j = 4 because of break.
```

```
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
4 x 1 = 4       4 x 2 = 8       4 x 3 = 12
5 x 1 = 5       5 x 2 = 10      5 x 3 = 15
```

Nested while Loops with continue

- continue : Skips the rest of the current iteration and moves to the next iteration of the loop.
- Using continue in the inner loop does not affect the outer loop except in logic flow.

Syntax with continue

```python
while condition_outer:
    # Outer loop body
    while condition_inner:
        if some_condition:
            continue  # Skip current iteration of inner loop
```

```python
# Nested while loop with continue
i = 1  # Outer loop counter
```

```
while i ⩽ 5:  # Outer loop
    j = 1  # Inner loop counter
    while j ⩽ 5:  # Inner loop
        if j == 3:  # Skip multiplication when j is 3
            j += 1  # Important: increment before continue
            continue
        print(f"{i} x {j} = {i*j}", end="\t")
        j += 1
    print()
    i += 1
# Here, the inner loop skips printing when j = 3 because of continue.
```

```
1 x 1 = 1       1 x 2 = 2       1 x 4 = 4       1 x 5 = 5
2 x 1 = 2       2 x 2 = 4       2 x 4 = 8       2 x 5 = 10
3 x 1 = 3       3 x 2 = 6       3 x 4 = 12      3 x 5 = 15
4 x 1 = 4       4 x 2 = 8       4 x 4 = 16      4 x 5 = 20
5 x 1 = 5       5 x 2 = 10      5 x 4 = 20      5 x 5 = 25
```

NOTE Always update loop counters before continue to avoid infinite loops.

Python For Loops

- A `for` loop in Python is used to iterate over a sequence, such as a list, tuple, dictionary, set, or string. Unlike traditional loops in other languages that use indexing, Python's `for` loop works more like an iterator.
- Unlike a while loop, which continues execution as long as a certain condition remains true, a for loop is typically used when the number of iterations is predetermined—you already know how many times the loop should run.

Syntax:

```
for element in sequence:
    # Code block to be executed repeatedly
    body_of_for_loop
```

Looping Through a List

- A `for` loop allows us to iterate through each item in a list and perform actions on them.

```
# Define a list of programming languages
languages = ["Python", "Java", "C++", "JavaScript"]

# Iterate through the list
for lang in languages:
    print(lang)  # Output each language

# Output:
# Python
# Java
# C++
# JavaScript
```

```
Python
Java
C++
JavaScript
```

Looping Through a String

- Strings in Python are sequences of characters, which means we can iterate through them like a list.

```
text = "AI"

# Iterate through each character
for char in text:
    print(char)

# Output:
# A
# I
```

```
A
I
```

Using break in a For Loop

- The `break` statement stops the loop when a certain condition is met.

```python
languages = ["Python", "Java", "C++", "JavaScript"]

for lang in languages:
    if lang == "C++":
        break  # Stop when "C++" is found
    print(lang)

# Output:
# Python
# Java
```

```
Python
Java
```

Using continue in a For Loop

- The `continue` statement skips the current iteration and moves to the next item.

```python
languages = ["Python", "Java", "C++", "JavaScript"]

for lang in languages:
    if lang == "C++":
        continue  # Skip "C++"
    print(lang)

# Output:
# Python
```

```
# Java
# JavaScript
```

```
Python
Java
JavaScript
```

The `range()` Function in For Loops

- The `range()` function generates a sequence of numbers, which can be useful for loops.

```
# Loop from 0 to 4
for i in range(5):
    print(i)

# Output:
# 0
# 1
# 2
# 3
# 4
```

```
0
1
2
3
4
```

```
### Specifying Start and End
for i in range(2, 6):
    print(i)

# Output:
# 2
# 3
```

```
# 4
# 5
```

```
2
3
4
5
```

```
### Using a Step Value = 2
for i in range(1, 10, 2):
    print(i)

# Output:
# 1
# 3
# 5
# 7
# 9
```

```
1
3
5
7
9
```

Else in a For Loop

- The `else` block in a `for` loop runs when the loop finishes normally (i.e., no `break`).
- If the loop is stopped with `break`, the `else` block does not execute.

```
for i in range(3):
    print(i)
else:
    print("Loop completed!")
```

```
# Output:
# 0
# 1
# 2
# Loop completed!
```

```
0
1
2
Loop completed!
```

```
#If the loop is stopped with `break`, the `else` block does not execute.
for i in range(3):
    if i == 1:
        break
    print(i)
else:
    print("Loop completed!")

# Output:
# 0
```

```
0
```

The pass Statement in a For Loop

- The `pass` statement is used as a placeholder when a loop cannot be empty.

```
for i in range(3):
    pass  # Placeholder, no action

# No output
```

Nested for Loops

- A nested for loop is a loop inside another loop. The outer loop runs first, and for each iteration of the outer loop, the inner loop executes completely.

Syntax

```
for outer_variable in iterable_outer:
    # Outer loop body
    for inner_variable in iterable_inner:
        # Inner loop body
```

```
# Nested for loop example : Multiplication Table

for i in range(1, 6):  # Outer loop for numbers 1 to 5
    for j in range(1, 6):  # Inner loop for numbers 1 to 5
        print(f"{i} x {j} = {i*j}", end="\t")
    print()  # New line after inner loop
```

```
1 x 1 = 1       1 x 2 = 2       1 x 3 = 3       1 x 4 = 4       1 x 5
= 5
2 x 1 = 2       2 x 2 = 4       2 x 3 = 6       2 x 4 = 8       2 x 5
= 10
3 x 1 = 3       3 x 2 = 6       3 x 3 = 9       3 x 4 = 12      3 x 5
= 15
4 x 1 = 4       4 x 2 = 8       4 x 3 = 12      4 x 4 = 16      4 x 5
= 20
5 x 1 = 5       5 x 2 = 10      5 x 3 = 15      5 x 4 = 20      5 x 5
= 25
```

Nested for Loops with break

- Break : Immediately exits the current loop (inner loop in nested cases).
- Using break in the inner loop does not affect the outer loop except in logic flow.

Syntax with break

```
for outer_variable in iterable_outer:
    # Outer loop body
    for inner_variable in iterable_inner:
        # Inner loop body
        if some_condition:
            break  # Exit inner loop immediately
```

```
#  Nested for loop with break

for i in range(1, 6):  # Outer loop
    for j in range(1, 6):  # Inner loop
        if j == 4:  # Break inner loop when j is 4
            break
        print(f"{i} x {j} = {i*j}", end="\t")
    print()
# Inner loop stops at j = 4 due to break.
```

```
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
4 x 1 = 4       4 x 2 = 8       4 x 3 = 12
5 x 1 = 5       5 x 2 = 10      5 x 3 = 15
```

Nested for Loops with continue

- continue : Skips the rest of the current iteration and moves to the next iteration of the loop.
- Using continue in the inner loop does not affect the outer loop except in logic flow.

Syntax with continue

```
for outer_variable in iterable_outer:
    # Outer loop body
    for inner_variable in iterable_inner:
        # Inner loop body
```

```
            if some_condition:
                continue  # Skip current iteration of inner loop
```

```
# Nested for loop with continue

for i in range(1, 6):  # Outer loop
    for j in range(1, 6):  # Inner loop
        if j == 3:  # Skip when j is 3
            continue
        print(f"{i} x {j} = {i*j}", end="\t")
    print()
# Inner loop skips printing when j = 3 due to continue.
```

```
1 x 1 = 1       1 x 2 = 2       1 x 4 = 4       1 x 5 = 5
2 x 1 = 2       2 x 2 = 4       2 x 4 = 8       2 x 5 = 10
3 x 1 = 3       3 x 2 = 6       3 x 4 = 12      3 x 5 = 15
4 x 1 = 4       4 x 2 = 8       4 x 4 = 16      4 x 5 = 20
5 x 1 = 5       5 x 2 = 10      5 x 4 = 20      5 x 5 = 25
```

Python range() Function

- The `range()` function in Python generates a sequence of numbers. It is commonly used in **loops** to iterate over a specified range without storing all values in memory, making it highly **memory efficient**.

Syntax

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

Parameter Details

Parameter	Description	Default Value	Required?
`start`	The first number in the sequence	`0`	Optional

Parameter	Description	Default Value	Required?
`stop`	The number **before which** the sequence stops	**N/A**	Required
`step`	The difference between each number in the sequence	`1`	Optional

Useful Information

Behavior	Explanation
`range(n)`	Generates numbers from `0` to `n-1`
`range(a, b)`	Generates numbers from `a` to `b-1`
`range(a, b, c)`	Generates numbers from `a` to `b-1` with step `c`
Negative step	Used for **reverse iteration**

Key Points About the `range()` Function

- The `range()` function operates **only with integers** — all its parameters (`start`, `stop`, and `step`) must be integer values. You cannot use floating-point numbers or other data types as arguments.
- Each of the three parameters can be **either positive or negative**.
- The `step` **value cannot be zero**; if you set `step = 0`, Python will raise a `ValueError`.

1. Using `range(stop)` – Default Start (0) and Step (1)

```
# range(5) generates numbers from 0 to 4 (default start: 0, step: 1)

for index in range(5):
    print(index)

# Output:
# 0
# 1
# 2
```

```
# 3
# 4
```

```
0
1
2
3
4
```

2. Using `range(start, stop)` – Specified Start and Default Step (1)

```
# Training model over epochs from 5 to 10 (inclusive of 5 but exclusive
of 11)

for index in range(5, 11):
    print(index)

# Output:
# 5
# 6
# 7
# 8
# 9
# 10
```

```
5
6
7
8
9
10
```

3. Using `range(start, stop, step)` – Specified Step Value

```python
for index in range(1, 11, 2):
    print(index)

# Output:
# 1
# 3
# 5
# 7
# 9
```

```
1
3
5
7
9
```

4. Using `range()` with a Negative Step (Counting Backwards)

```python
for index in range(10, 0, -2):
    print(index)

# Output:
# 10
# 8
# 6
# 4
# 2
```

```
10
8
6
4
2
```

www.intensitycoding.com

Found this helpful ?

Follow on LinkedIn
Master AI/ML with Intensity Coding

@bhavdippatel2020

Like | Comment | Share | Save

Each Article Includes Everything You Need

THEORY MADE SIMPLE
Complex ideas explained in an easy way

PYTHON CODE
Hands-on coding for practice

VISUAL LEARNING
Visual diagrams for clarity

MATH BEHIND AI/ML
Step-by-step explanation of core concepts

Explore more tutorials at Intensity Coding

www.intensitycoding.com

Intensity Coding
Something For Everyone