

From Static Script to Intelligent Program

Your Toolkit for Control Flow in Python



Choice



Repetition



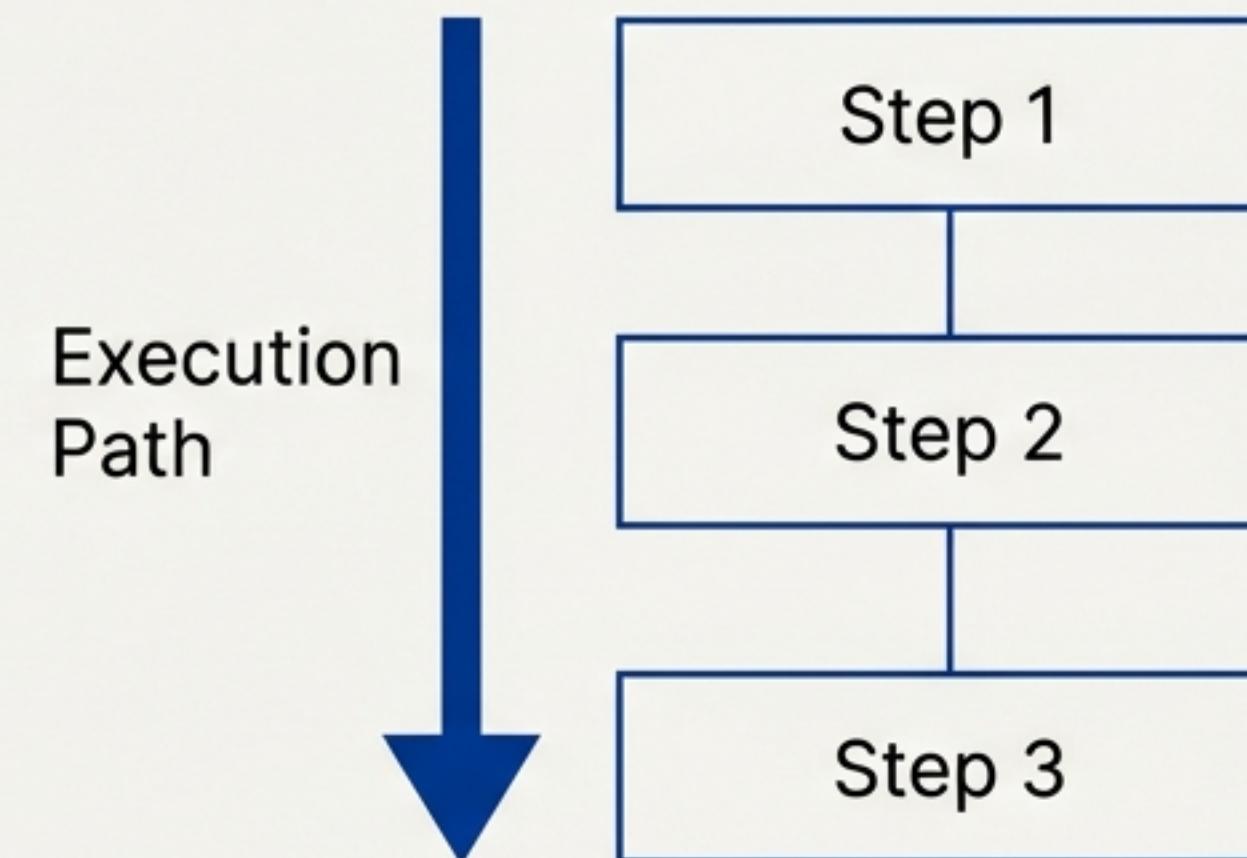
Precision

Programs need to make decisions and repeat actions. Control flow structures are the foundation of programming logic—they are the tools that transform simple scripts into intelligent, responsive applications. This guide will walk you through your essential toolkit.

Without Control, Code is Just a Checklist

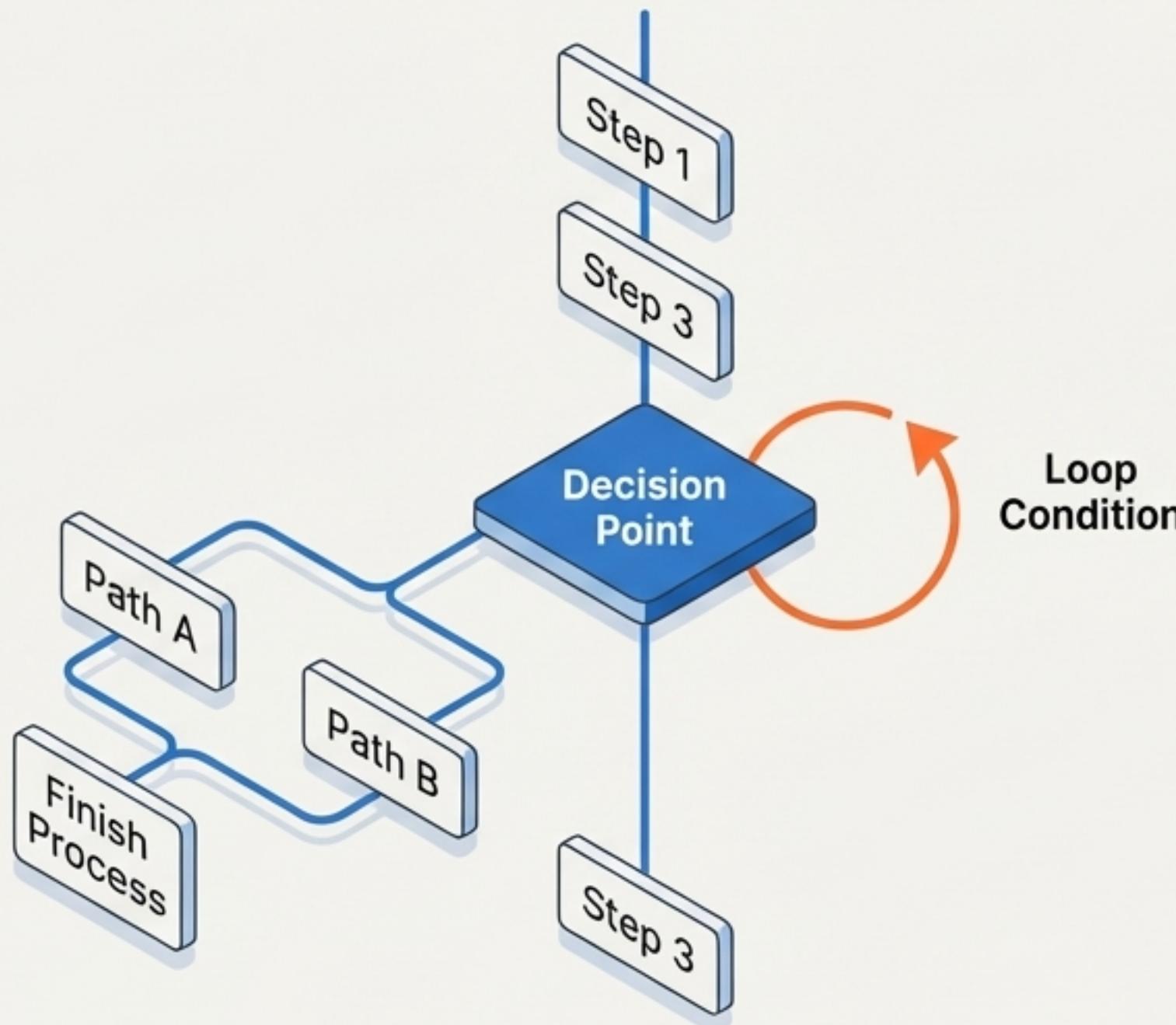
By default, a Python script executes from top to bottom, one line at a time. It's rigid and unthinking. It cannot react to user input, handle different data, or repeat a task without you manually copying the code.

```
# a_script.py
print("Step 1: Start process.")
print("Step 2: Load data.")
print("Step 3: Finish process.")
```



Control Flow is the Brain of Your Program

These structures give your code the ability to analyze situations and change its path. We can group these powerful tools into three main categories, each giving your program a new capability.



The Power of Choice.

Tools to make decisions and choose different paths.



The Power of Repetition.

Tools to automate tasks and repeat actions.



The Power of Precision.

Tools to refine loops and build complex, multi-layered logic.

The Power of Choice: Making Decisions with `if/elif/else`

Big Idea

Your program must respond intelligently to different situations. The `if/elif/else` structure is the fundamental tool for creating branching paths in your code based on conditions.

How it Works

Python evaluates conditions from top to bottom and executes the code block for the ***first*** condition that is `True`. All other branches are skipped.

```
score: int = 85
grade: str = ""

if score >= 90: # Is score 90 or higher?
    grade = "A"
elif score >= 80: # If not, is it 80 or higher? --> TRUE for 85
    grade = "B" # This block runs.
else:
    grade = "F" # This is skipped.

print(f"The grade is: {grade}") # Output: The grade is: B
```

- A colon ends the condition line.
- Indentation (4 spaces) defines the code block.
- Checks the next condition in the sequence.

A Modern Tool for Choice: Pattern Matching with `match/case`

Big Idea

When you find yourself checking the **same variable** against many **different specific values**, `match/case` (new in Python 3.10) offers a cleaner, more readable structure that looks like a decision table.

Repetitive Checks

```
status_code: int = 404
message: str = ""

if status_code == 200:
    message = "OK"
elif status_code == 404:
    message = "Not Found"
elif status_code == 500:
    message = "Server Error"
else:
    message = "Unknown"
```

Clean Decision Table

```
status_code: int = 404
message: str = ""

match status_code:
    case 200:
        message = "OK"
    case 404:
        message = "Not Found"
    case 500:
        message = "Server Error"
    case _: # The "wildcard" for everything else
        message = "Unknown"
```

Notice how `match` states the variable once, making the list of possible values highly scannable.

Choosing the Right Tool for the Job

if / elif / else

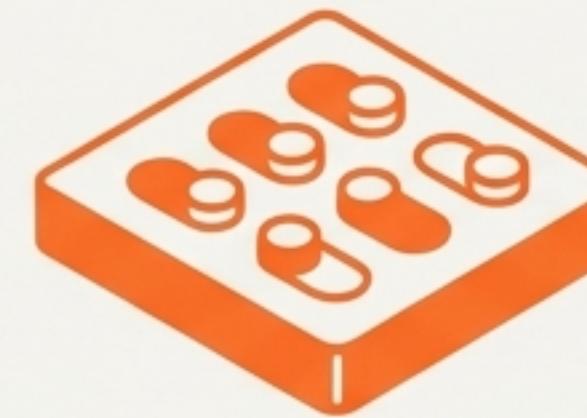


Complex Conditions & Ranges

Core Strength: Unmatched flexibility. Can handle any combination of variables, comparison operators (`>`, `<=`, `!=`), and logical operators (`and`, `or`).

```
# Checking ranges and multiple variables
if (is_member and purchase > 50) or (purchase > 100):
    apply_discount()
```

match / case



Specific Values & Patterns

Core Strength: Superior readability when comparing one variable against a list of exact, literal values. Clearly signals "here are all the possibilities."

```
# Routing based on a single command string
match command:
    case "save": save_file()
    case "load": load_file()
```

The Power of Repetition: Automating with `for` Loops

Big Idea

Use a `for` loop when you need to repeat an action a *known number of times* or for *every item in a sequence*. It automates the counting for you.

The `range()` Function

- `range(5)` -> 0, 1, 2, 3, 4 (Stops before 5)
- `range(2, 6)` -> 2, 3, 4, 5 (Starts at 2, stops before 6)
- `range(10, 0, -1)` -> 10, 9, ... 1 (Counts down by 1)

```
for i in range(10, 0, -1): # Counts down from 10 to 1
    print(f"{i}...")
print("Liftoff!") # Runs once after the loop finishes
```

Key Insight: `range()` *never* includes the `stop` value. This is a frequent source of 'off-by-one' errors. To count 1-to-10, you must use `range(1, 11)`.

Repeating Until a Condition Changes with `while` Loops

Big Idea

Use a `while` loop when you don't know how many times you need to repeat. The loop continues *as long as a condition* remains True.

Critical Responsibility

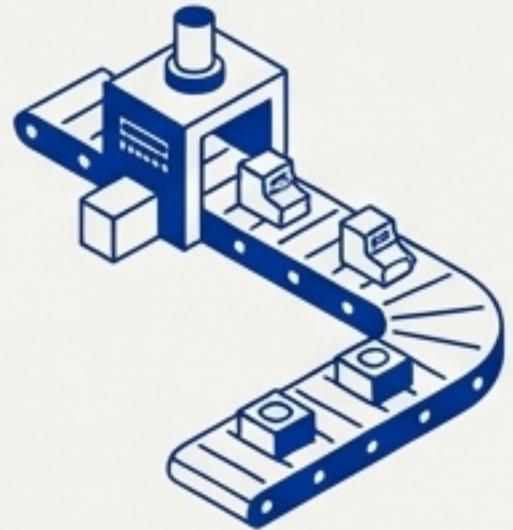
You must ensure the condition can eventually become False. If the variable you're checking never changes inside the loop, you create an infinite loop that freezes your program.

```
● ● ●  
age: int = 0  
  
# Loop as long as age is not in the valid range 1-120  
while age < 1 or age > 120:  
    user_input = input("Enter a valid age (1-120): ")  
    if user_input.isdigit():  
        age = int(user_input) # This line is key! It changes the 'age' variable.  
  
print(f"Age confirmed: {age}")
```



Without this update,
the loop would run
forever!

Choosing the Right Loop for the Task



`for` Loop

Use When

You have a definite number of iterations.

Core Question

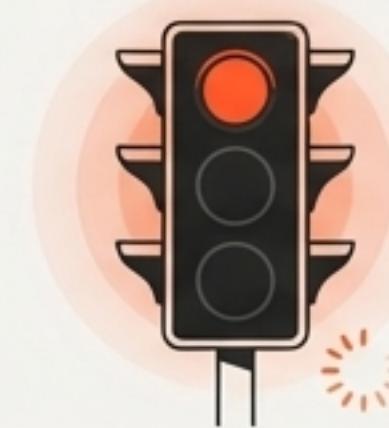
"How many times?" or "For each item?"

Keywords

For each, for N times, iterate over.

Scenarios

- Process every item in a list.
- Print the numbers from 1 to 100.
- Run a task exactly 5 times.



`while` Loop

Use When

Repetition depends on a changing condition.

Core Question

"Should I keep going?"

Keywords

Until, while, as long as.

Scenarios

- Keep asking for input until it's valid.
- Retry a network request until it succeeds.
- Run a game loop until the player quits.

Rule of Thumb: If you know the count, use `for`. If you're waiting for a condition, use `while`.

The Power of Precision: Fine-Tuning Your Loops

Sometimes you need more control than just letting a loop run its course.

‘break’ and ‘continue’ allow you to alter the flow from **inside** the loop.

‘break’ (The Emergency Exit)

Big Idea

Stop working the moment you have the answer.

‘break’ immediately terminates the **entire** loop.

Use Case

Searching.



```
numbers = [3, 7, 12, 5, 19, 8]
target = 12
for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break # Exit loop. 5, 19, 8 are never checked.
```

‘continue’ (The Skip Button)

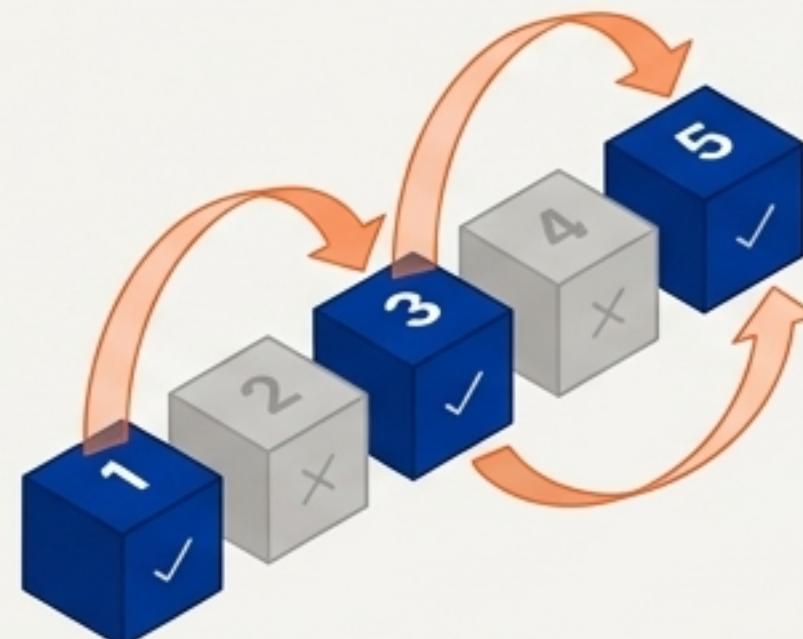
Big Idea

Ignore the current item and move to the next one.

‘continue’ skips the rest of the current iteration and jumps to the top of the next one.

Use Case

Filtering.



```
# Process only odd numbers
for num in range(1, 6):
    if num % 2 == 0: # If the number is even...
        continue # ...skip the print() and go to the next
    print(f"Processing odd number: {num}")
# Output: 1, 3, 5
```

The Master Technique: Building Logic with Nesting

Big Idea: You can place any control structure inside another to build layers of logic. This is how you solve problems that have multiple steps or dimensions.

Pattern 1: Conditionals Inside Loops (Selective Processing)

Use Case: Iterate through a collection, but only act on certain items.



```
# Sum only the positive numbers in a list
data = [10, -5, 20, 0, -15, 30]
total = 0

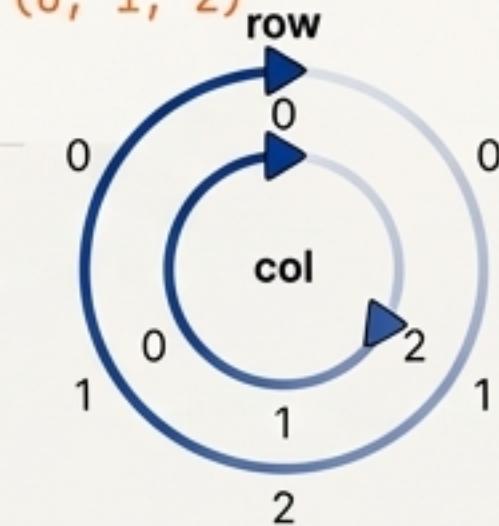
for number in data:
    if number > 0: # The decision is INSIDE the loop
        total += number

print(f"Sum of positive numbers: {total}") # Output: 60
```

Pattern 2: Nested Loops (Two-Dimensional Iteration)

Use Case: Process data in grids, tables, or matrices (rows and columns).

```
# Generate coordinates for a 3x3 grid
for row in range(3):      # Outer loop (0, 1, 2)
    for col in range(3): # Inner loop (0, 1, 2)
        print(f"({row}, {col})")
```



The inner loop runs completely for EACH iteration of the outer loop.

Advanced Precision: Did the Loop Find Anything?

Big Idea: Python's loops have a unique `else` clause that runs **only if the loop completes normally*** (i.e., it was not terminated by a `break` statement). This lets you handle a “not found” case elegantly.

The Problem

How do you know if a search loop finished because it found the item or because it checked everything and found nothing?

The Solution

‘for...else’

Scenario 1: Name is found

```
names = ["Alice", "Bob", "Charlie"]
for name in names:
    if name == "Bob":
        print("Found Bob!")
        break # The break skips the else block
    else:
        print("Bob was not in the list.")
# Output: Found Bob!
```

JetBrains Mono
The break skips the
else block

This is much cleaner than using a
separate `found = False` flag variable.

Scenario 2: Name is not found

```
for name in names:
    if name == "David":
        print("Found David!")
        break
    else: # The loop completes without a break
        print("David was not in the list.")
```

JetBrains Mono
The loop completes
without a break

Output: David was not in the list.

Debugging Clinic: Common Control Flow Errors

Conditional Logic Errors

🚩 Unreachable `elif`

Wrong: JetBrains Mono

```
if score >= 60:  
    ...  
elif score >= 90:  
    ...
```

Fix: Order from most specific to least specific (`>= 90` first).

Loop Errors

🚩 Infinite `while` Loop

Wrong: JetBrains Mono

```
count=0;  
while count < 5:  
    print(count) !
```

Fix: Always update the loop variable inside the loop (`count += 1`).

Pattern Matching Errors

🚩 Unreachable `case`

Wrong: JetBrains Mono

```
match status:  
    case _: ...  
    case 404: X
```

Fix: The wildcard `case _` must always be the *last* case.

Impossible Condition

Wrong: JetBrains Mono

```
if age >= 18 and age < 13:
```

Fix: Check your logic. You likely meant `or` or a different range.

Off-by-One Error

Wrong: JetBrains Mono

```
for i in range(1, 10):  
    # Tries to print 1 to 10
```

Fix: Remember `range()` excludes the stop value. Use `range(1, 11)`.

Type Mismatch

Wrong: JetBrains Mono

```
status_code = 200;  
match status_code:  
    case '200': ...
```

Fix: Ensure the `case` type (`'200'`) matches the variable's type (`200`). Use `case 200:`.

Your Complete Control Flow Workbench

Tools for
CHOICE



`'if/elif/else'`

For any decision,
especially ranges
and complex logic.



`'match/case'`

For clean, readable
checks against a list
of specific values.

Tools for
REPETITION



`'for' loop`

For repeating a
known number of
times.



`'while' loop`

For repeating as
long as a condition
holds true.

Tools for
PRECISION & COMPLEXITY



`'break/continue'`

For fine-tuning loop
flow: exiting early or
skipping iterations.



`Nesting`

The master technique
for combining all tools
to build multi-layered
logic.

Thinking with AI: Describe Your Intent, Not Just the Syntax

As you collaborate with AI, your most valuable skill isn't memorizing syntax—it's clearly describing the logical intent. The AI can handle the code, but you must provide the strategy. Frame your requests around the “why.”

Instead of describing the syntax...

‘Write me a `while` loop that...’

‘Generate a `for` loop with `range()`...’

‘Use `break` in an `if` statement...’

‘Create a nested `if` structure...’

Describe the logical goal...

“**Loop until** the user enters ‘quit.’” (**Signals a `while` loop**)

“**Process every item** in this list.” (**Signals a `for` loop**)

“Search this data and **stop as soon as you find** the first match.” (**Signals `break`**)

“**First**, check if the user is over 25. **If they are**, then check if they have a valid license.” (**Signals nesting**)

Master the logic, and you’ll be able to build anything, with or without an AI partner.