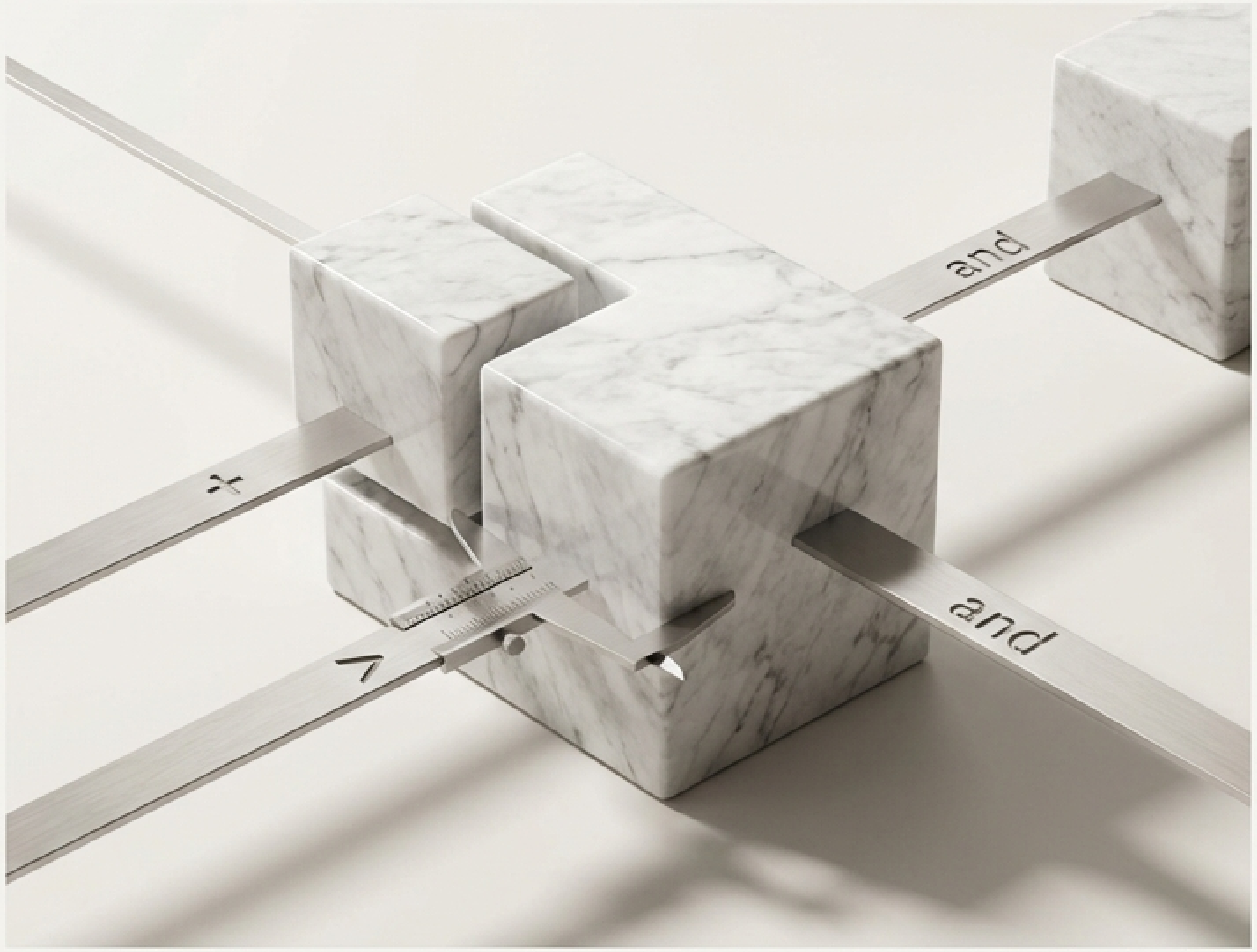
The Verbs of Python

Mastering Operators to Bring Data to Life



You’ve learned the ‘nouns’ of Python (data types). Now, you’ll learn the ‘verbs’ (operators) to make your data do things.

From Nouns to Sentences: Your Programming Journey

Data Types & Variables
(The Nouns)

Storing information (`int`, `float`, `str`).

→

Operators
(The Verbs)

Performing actions and asking questions. **(This is our focus)**

→

Control Flow
(The Sentences)

Making decisions (`if`, `for`, `while`).

Mastering operators is how you move from simply storing data to creating logic. We will cover the four fundamental types of operators that make this possible.

Block 1: The Verbs of Calculation

Arithmetic Operators

These seven operators (+`, `-`, `*`, `/`, `//`, `%`, `**`) perform the math you already know. The key is understanding how Python handles data types.

Design Insight

Why does `10 / 5` give `2.0`?

Python's standard division (/`) **always returns a float**. This is a deliberate design choice for consistency, because division often results in decimals. If you need an integer result (e.g., counting whole groups), you must use floor (//`).

```
# Standard division always yields a float
result_float: float = 10 / 5  # Result is 2.0
print(type(result_float))     # <class 'float'>

# Floor division yields an integer (with int inputs)
result_int: int = 10 // 5 # Result is 2
print(type(result_int))   # <class 'int'>
```

NotebookLM

The Arithmetic Operator Toolkit

Operator	Name	Example (a=10, b=3)	Result	Result Type	Common Use Case
`+`	Addition	`a + b`	13	`int`	Combining totals.
`-`	Subtraction	`a - b`	7	`int`	Finding difference.
`*`	Multiplication	`a * b`	30	`int`	Scaling values.
`/`	Division	`a / b`	3.333....	`float`	Precise calculation.
`//`	Floor Division	`a // b`	3	`int`	Counting whole groups.
`%`	Modulus	`a % b`	1	`int`	Finding remainders (e.g., `num % 2` for even/odd).
`**`	Exponentiation	`a ** b`	1000	`int`	Raising to a power.

Common Pitfall

For exponentiation, always use `**` (e.g., `2 ** 3`). The caret ^ is for a different operation (bitwise XOR) and will produce unexpected results.

NotebookLM

Block 2: The Verbs of Questioning

Comparison Operators

Comparison operators don’t return numbers; they evaluate a condition and return a boolean value: `True` or `False`. They are the foundation of decision-making.

Assignment ('=') vs. Comparison ('==')

A single equals (=) **stores** a value. A double equals (==) **asks** if two values are the same. Confusing them is a common error that leads to bugs.

```
# Assignment: "Make the value of score 100"
score = 100 ←—— STORING ——
                         ? ASKING

# Comparison: "Is the value of score 100?"
is_perfect_score = (score == 100) # Result is True
```

Value vs. Type Equality

Python’s `==` compares values, not necessarily types. This is why `5 == 5.0` == 5.0` returns `True` (same value), but `5 == "5"` returns `False` (different types and values).

Block 3: The Verbs of Reasoning

Logical Operators

Logical operators (`and`, `or`, `not`) combine boolean values to create complex conditions for decision-making.

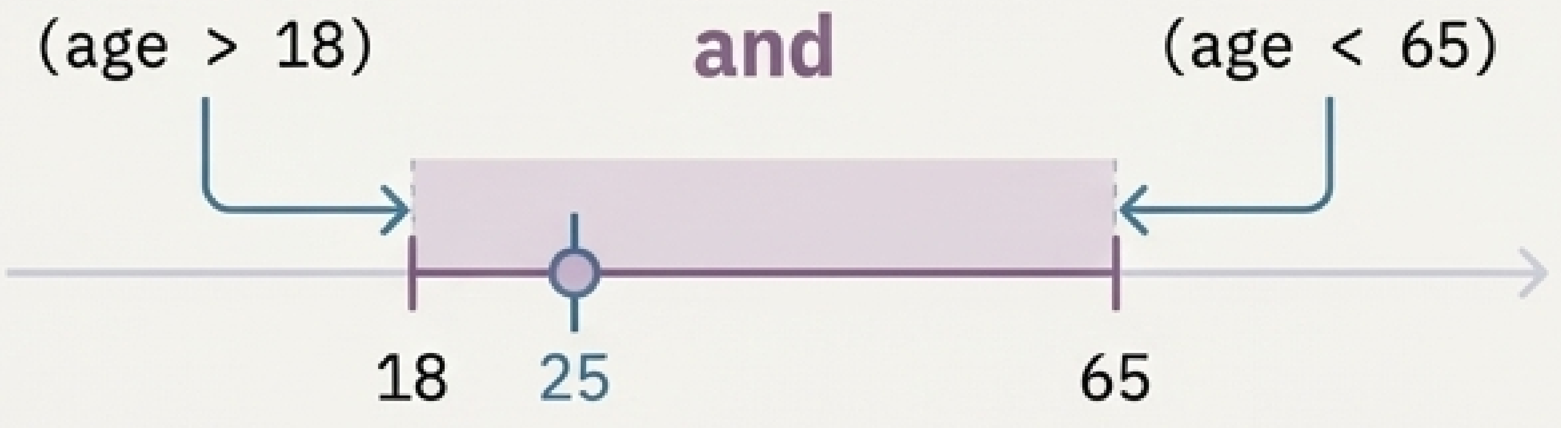
Operator breakdown:

- `and`: **Both** conditions must be `True`. Use for checking multiple requirements.
- `or`: **At least one** condition must be `True`. Use for checking alternative requirements.
- `not: **Reverses** the boolean value (True becomes 'False, 'False becomes 'True).

Practical Example: Checking a Range

Goal: Check if `age` is between 18 and 65.

Logic: This requires two comparisons joined by `and`.



```
age: int = 25

# Is age greater than 18 AND less than 65?
is_working_age = (age > 18) and (age < 65)
# Result: True and True -> True
```

A Rule for Clarity: Operator Precedence and Parentheses

Python evaluates operators in a specific order (PEMDAS: Parentheses, Exponents, Multiplication/Division, Addition/Subtraction). However, relying on memory is fragile.

Expert Insight

Don’t Memorize. Clarify.

The best practice is to always **use parentheses** to make your intent explicit. Code is read more often than it is written. Prioritize clarity for the human reader.

Ambiguous

```
result = 100 - 5 * 10
```

Clear

```
result = 100 - (5 * 10)
```

✓

Block 4: The Verbs of Updating

Assignment Operators

Assignment operators provide a concise shorthand for updating the value of a variable. They combine an arithmetic operation with assignment.

Shorthand	Equivalent To
`x += 5`	`x = x + 5`
`x -= 5`	`x = x - 5`
`x *= 5`	`x = x * 5`
`x /= 5`	`x = x / 5`

Common Patterns for Control Flow (Chapter 19)

The Counter Pattern

Used constantly in loops to track iterations.

```
item_count = 0
item_count += 1 # Increment the count
```

The Accumulator Pattern

Used to sum up values in a collection.

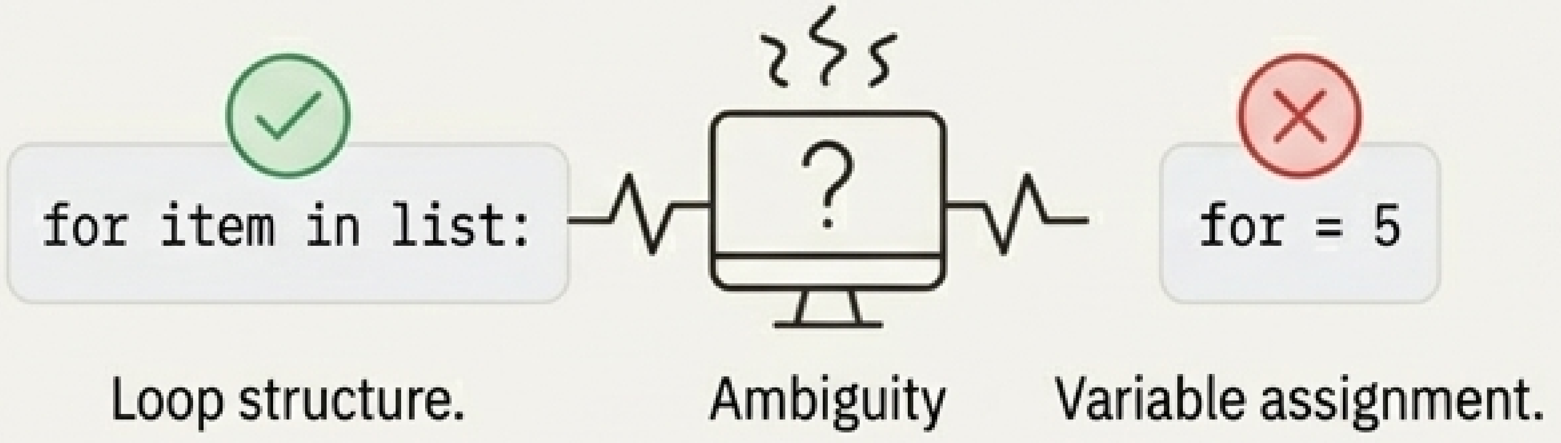
```
total_price = 0.0
total_price += item_price # Add to
the running total
```

Python's Protected Language: Keywords

Keywords are the 35 reserved words that form the structure of the Python language. Because they have special meaning, you cannot use them as variable names.

Why are they reserved?

If you could write `for = 5`, the Python interpreter wouldn't know if you were starting a loop or assigning a variable. Reservation prevents this ambiguity.



Example of Error

```
# This will raise a SyntaxError
for = 5 # Error: invalid syntax
```

SyntaxError

How to Check for Keywords

You don't need to memorize them. You can check programmatically.

```
import keyword

print(keyword.iskeyword('if'))
print(keyword.iskeyword('my_var'))
print(keyword.iskeyword('my_var'))
```

→ True

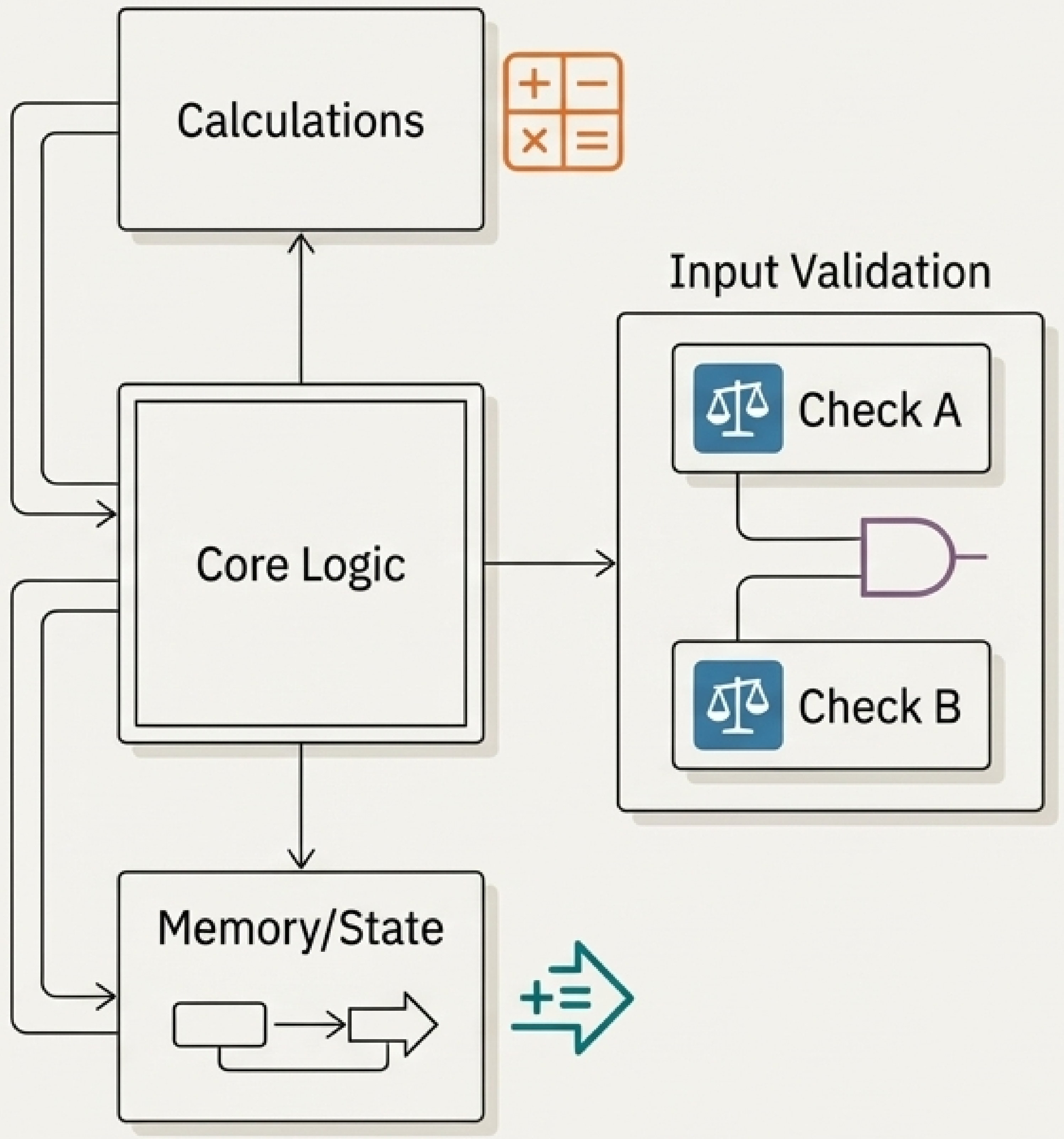
— False

Capstone: Bringing It All Together in a Type-Safe Calculator

Objective: Let's build a simple calculator to see all four operator types working together in a practical application. This project will integrate everything we've learned.

Checklist of Concepts Used

- ✓ **Arithmetic Operators:** For the core calculations (+, -, *, /).
- ✓ **Comparison** Operators: To validate inputs (e.g., check for division by zero).
- ✓ **Logical Operators:** To combine validation checks (e.g., num1 > 0 and num2 > 0).
- ✓ **Assignment Operators:** To track a running total (+=).
- ✓ **Type Validation:** Using type() to verify our results.



Anatomy of the Calculator Code

```python
# Simplified Calculator
num1: int = 15
num2: int = 4
running_total: float = 0.0

# 1. Arithmetic Operations
result: float = num1 / num2
print(f"{num1} / {num2} = {result}")
print(f"Result type: {type(result)}")

# 2. Comparison & Logical Checks
is_valid_op = (num1 > 0) and (num2 != 0)
print(f"Operation is valid: {is_valid_op}")

# 3. Assignment Operation
running_total += result
print(f"Running total: {running_total}")
```

Arithmetic: Performing the core calculation. Note the result is a `float`.

Comparison & Logical: Combining questions to validate our inputs before they cause errors.

Type Validation: A crucial habit to verify that our code is behaving exactly as we expect.

Assignment: Using the concise shorthand to update the program's state.

Knowledge Check: Four Core Operator Concepts

Before moving on, can you answer these key questions?

Why does `10 / 2` result in `2.0` (a `float`) and not `2` (an `int`)?

For type consistency. Python's `/` operator is designed to always return a `float` because division frequently produces decimals.

In one sentence, what is the difference between `x = 5` and `x == 5`?

`x = 5` is an assignment that stores the value 5 in x, while `x == 5` is a comparison that asks if `x` is equal to 5, returning `True` or `False`.

What is the most common use for `count += 1`?

It represents the 'Counter Pattern,' used universally inside loops to track the number of iterations.

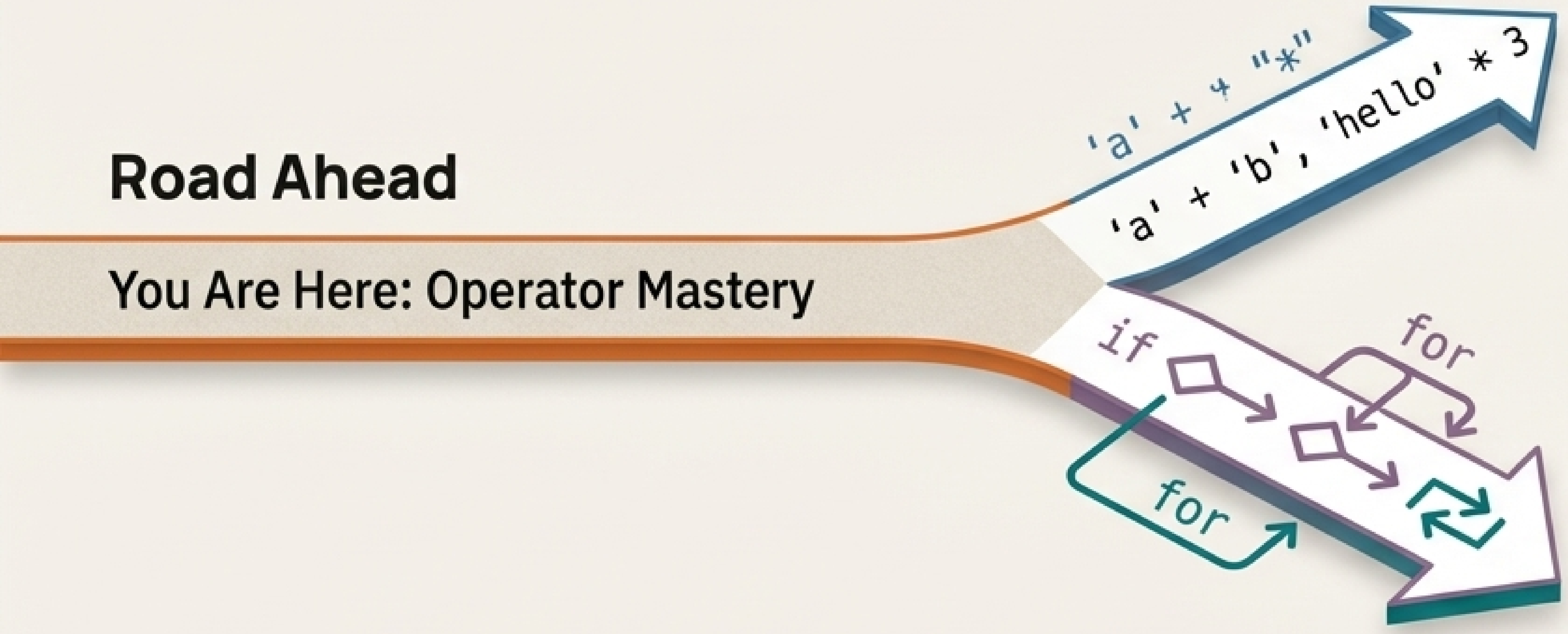
Why should you use parentheses in expressions like `price * (1 + tax_rate)`?

For clarity. While Python knows the order of operations, parentheses make the programmer's intent immediately and unmistakably clear to any human reader.

Your Foundation is Solid. What's Next?

You have mastered the four types of operators—the verbs of Python.
You can now move beyond simply storing data to performing calculations, asking complex questions, and managing state efficiently.

Road Ahead



Chapter 18: Strings & Text
Chapter 19: Strings & Text

Next, you'll see how operators like `+` and `*` can also be used to manipulate text.

Chapter 19: Control Flow

Soon after, you will use your comparison and logical operator skills to build **complete sentences**—making decisions and controlling program flow with `if`, `while`, and `for`.

Your operator toolkit is the foundation for all logic you will write from now on.