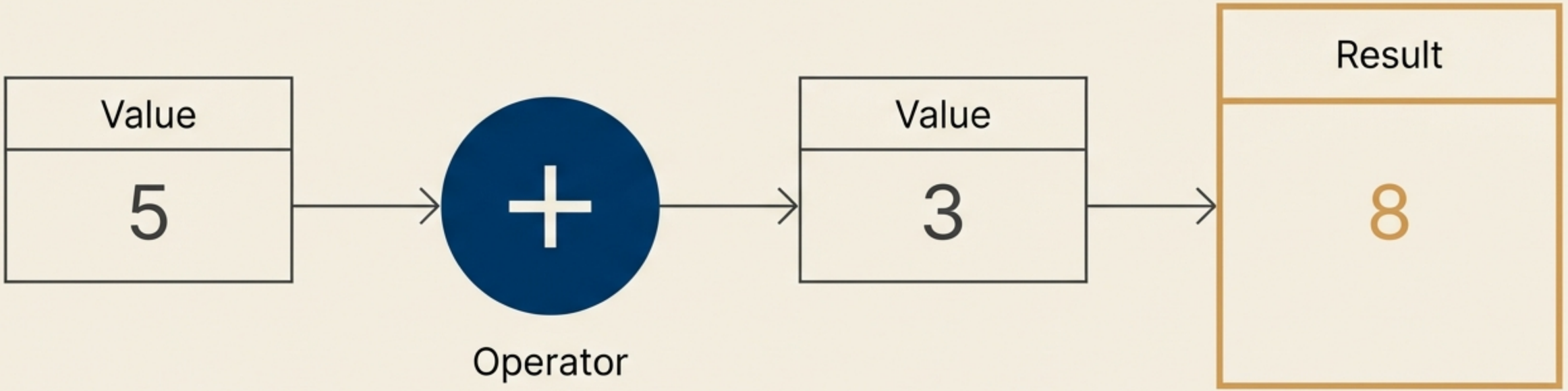
The Grammar of Python

A Guide to Mastering Operators, from Simple Calculation to Complex Control

Operators are the active verbs of your code.

In Python, operators are the symbols that perform operations on variables and values. They are the essential tools that allow you to manipulate data, make comparisons, and control the logical flow of your programmes. **Think of them not as mere symbols**, but as the powerful, expressive verbs that bring your code to life.



Arithmetic Operators: The Language of Calculation

Code in Action

```
# Basic operations
total_cost = 150 + 20
remaining_stock = 100 - 34

# Multiplication and exponentiation
area = 12 * 5
volume = 5 ** 3 # 5 to the power of 3

# Division and its variations
price_per_item = 199 / 4  # Float division
items_per_box = 199 // 4  # Floor division (discards remainder)
leftover_items = 199 % 4  # Modulus (gets only the remainder)
```

The Toolkit

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division (float)
//	Floor Division
%	Modulus (Remainder)
**	Exponent

Precision Matters: Understanding Float vs. Floor Division

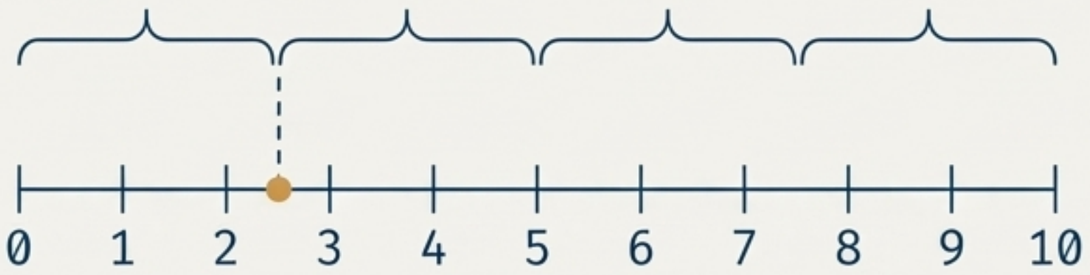
Python offers two distinct types of division. Choosing the right one is crucial for avoiding subtle bugs, especially when your logic depends on integer or floating-point results.

`/` Float Division

Always returns a `float`, preserving the precise decimal part.

```
10 / 4
```

```
2.5
```

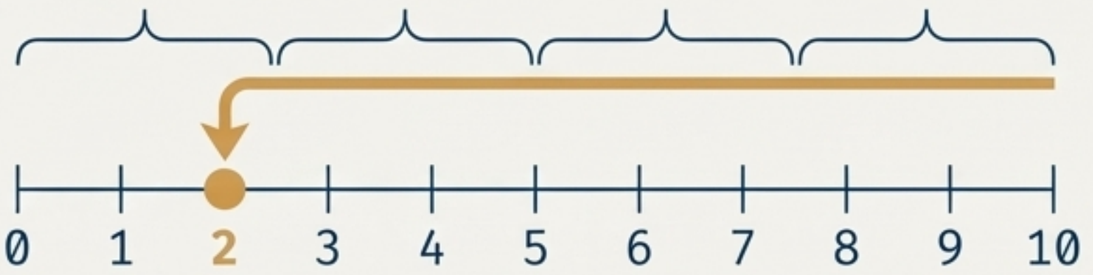


`//` Floor Division

Returns an integer, discarding the decimal part by rounding down to the nearest whole number.

```
10 // 4
```

```
2
```



Assignment Operators: Storing Results and Modifying State

Values are stored in variables using the assignment operator (`=`). Python also provides powerful 'augmented' assignment operators as a concise shorthand for modifying a variable's existing value.

1. The Foundation

```
x = 5
```

Fira Code (Fira Code)

2. The Verbose Update

```
x = x + 3
```

Fira Code (Fira Code)

3. The Pythonic Shorthand

```
x += 3
```

Fira Code (Fira Code)

The Toolkit

Operator	Example	Equivalent to
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 2`	`x = x - 2`
`*=`	`x *= 10`	`x = x * 10`
`/=`	`x /= 2`	`x = x / 2`
`**=`	`x **= 2`	`x = x ** 2`

Comparison Operators: The Basis of Decision-Making

Programmes gain intelligence by making decisions. Comparison operators are the tools for this; they evaluate a statement and return a Boolean value (`True` or `False`), forming the core of all conditional logic.

Code in Action

```
stock_level = 50
order_size = 75

# Is the stock level exactly 50?
stock_level == 50  # -> True

# Is the order larger than the stock?
order_size > stock_level # -> True

# Can we fulfil an order of 50?
stock_level >= 50 # -> True
```

The Toolkit

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

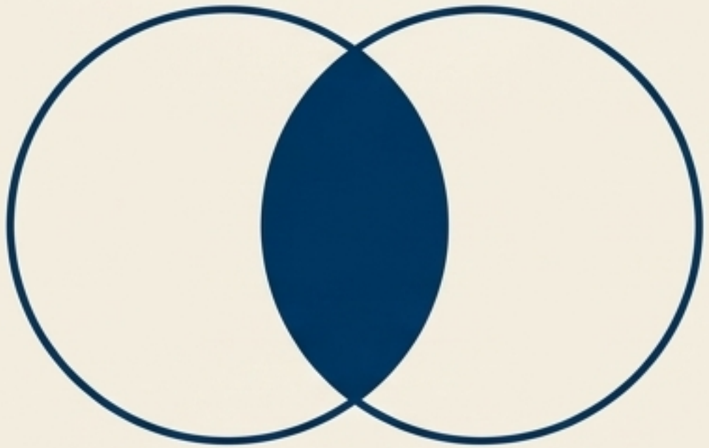
Logical Operators: Weaving Complex Conditions

Use `and`, `or`, and `not` to combine multiple `True`/`False` results into a single, more sophisticated logical statement. This allows for nuanced control over your programme's flow.

`and` (Intersection)

Returns `True` only if *both* conditions are true.

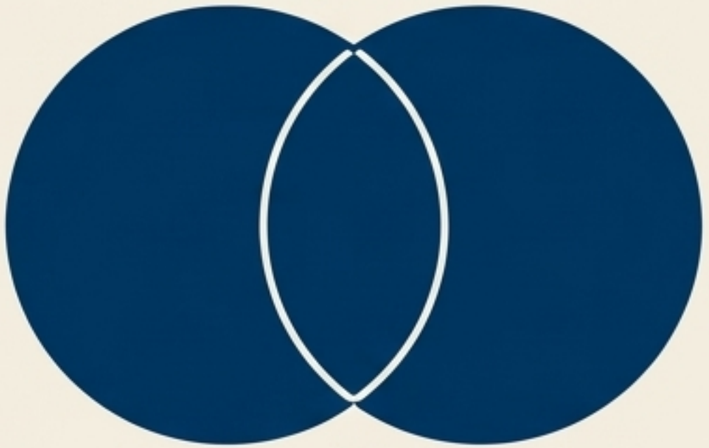
```
age > 18 and age < 65
```



`or` (Union)

Returns `True` if *at least one* condition is true.

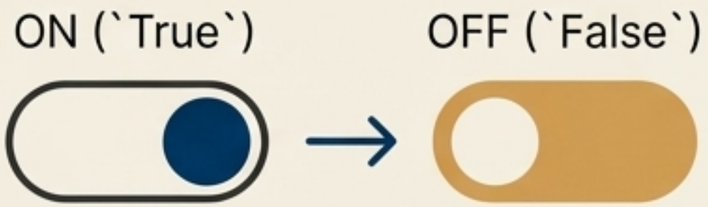
```
is_weekend or is_holiday
```



`not` (Inversion)

Reverses the logical state of a condition.

```
not is_logged_in
```



Membership Operators: Is it in the collection?

A common task is to check if an item exists within a sequence like a string, list, or tuple. Membership operators (`in` and `not in`) provide a clean and readable way to perform this check.

Code in Action

```
# Checking a list of approved users
approved_users = ['alice', 'bob', 'charlie']
user = 'dave'
user in approved_users  # -> False

# Checking for a substring
"a" in "cat"  # -> True

# The 'not in' operator
'z' not in 'python'  # -> True
```

The Toolkit

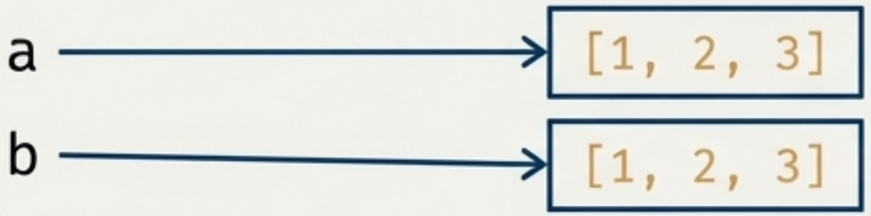
- `in`: Evaluates to `True` if a value is found in the sequence.
- `not in`: Evaluates to `True` if a value is *not* found in the sequence.

`==` vs. `is`: A Crucial Distinction Between Equality and Identity

Two variables can hold equal values, but are they the *exact same object* in memory? == checks for value equality, while `is` checks for object identity. This is a fundamental concept.

Equality is `True`, Identity is `False`

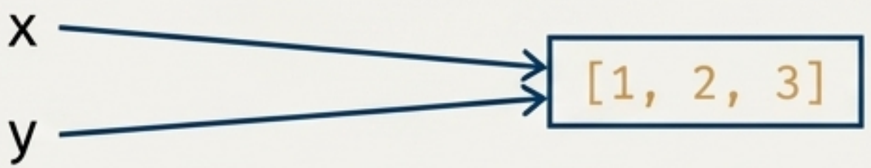
```
a = [1, 2, 3]
b = [1, 2, 3]
```



`a is b` is **`False`** (different objects). `a == b` is **`True`** (same content).

Equality is `True`, Identity is `True`

```
x = [1, 2, 3]
y = x
```



`x is y` is **`True`** (same object). `x == y` is **`True`** (same content).

Use `is` for identity. Use `==` for equality.

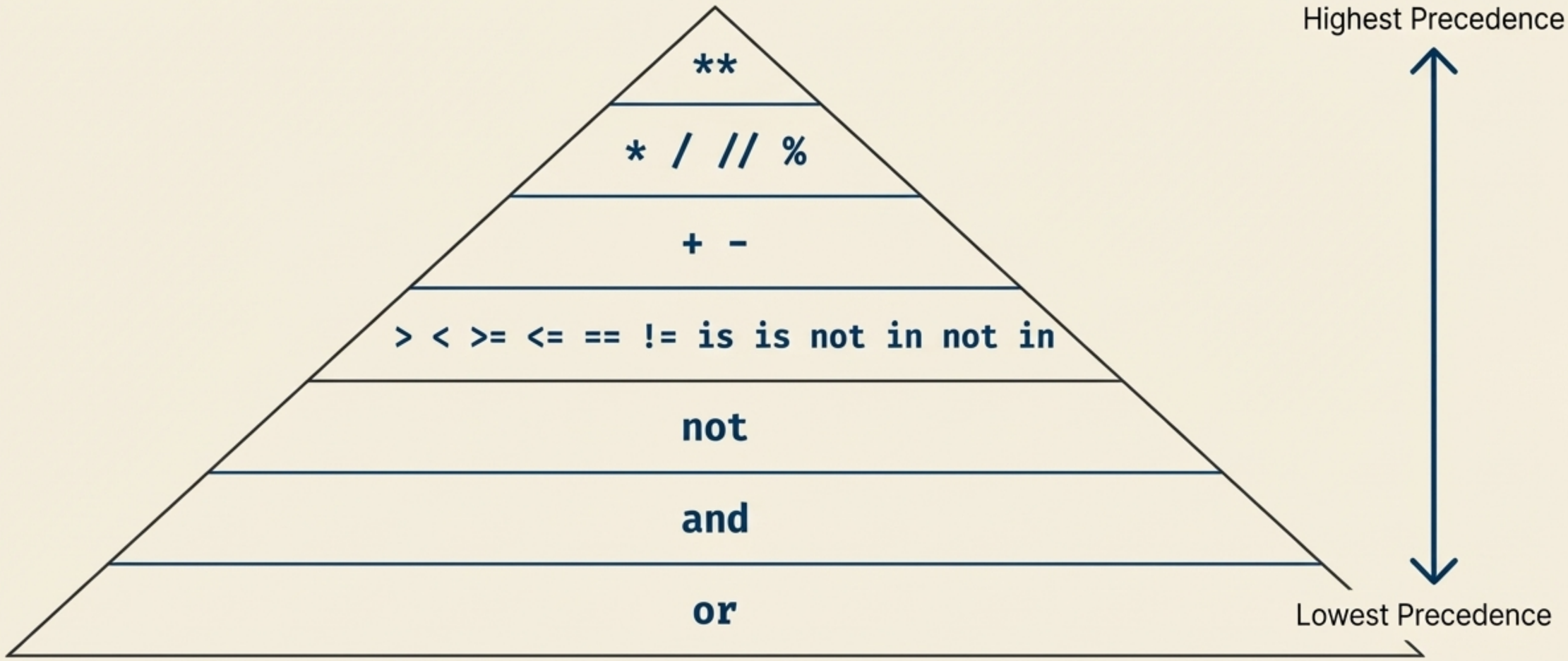
Bitwise Operators: Manipulating Data at the Binary Level

For performance-critical applications or low-level algorithms, Bitwise operators allow you to operate directly on the binary representation of integers. Consider them the tools for fine-grained, expert-level control.

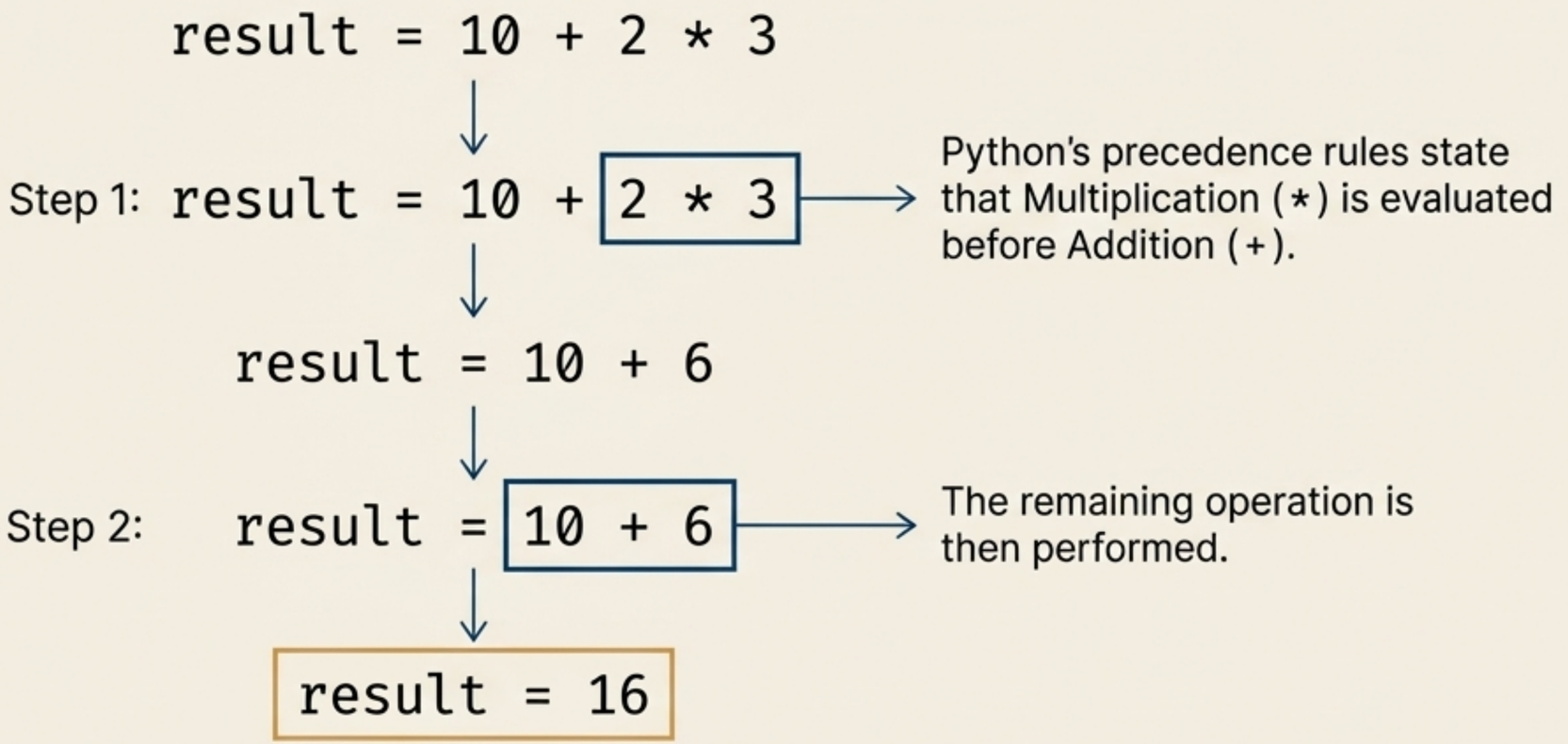
Operator	Meaning	Example (`5` and `3`)	Binary Explanation
`&`	AND	`5 & 3 → 1`	`101 & 011 → 001`
`|`	OR	`5 | 3 → 7`	`101 | 011 → 111`
`^`	XOR	`5 ^ 3 → 6`	`101 ^ 011 → 110`
`~`	NOT	`~5 → -6`	Inverts all bits
`<<`	Left shift	`5 << 1 → 10`	Shifts bits left
`>>`	Right shift	`5 >> 1 → 2`	Shifts bits right

Operator Precedence: The Unwritten Rules of Evaluation

When an expression contains multiple operators, which one runs first? Python follows a strict set of rules, known as operator precedence, to determine the order of evaluation. Mastering this hierarchy is essential for writing bug-free code.



Precedence in Practice: Deconstructing an Expression



The Golden Rule

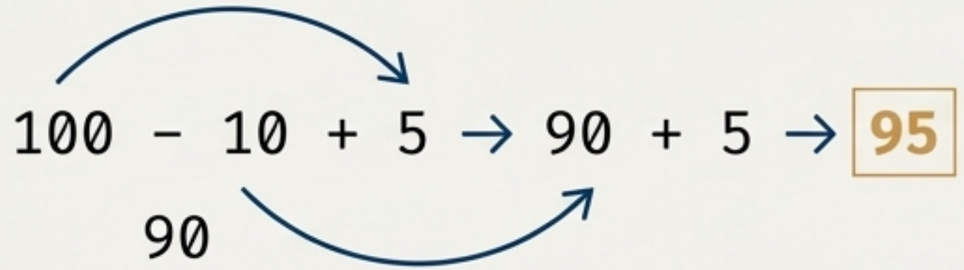
When in doubt, use parentheses ( ) to enforce your intended order of operations. `(10 + 2) * 3` evaluates to 36 , a completely different result.

Associativity: The Tie-Breaker for Equal Precedence

When an expression has multiple operators of the same precedence, associativity determines the direction of evaluation: left-to-right or right-to-left.

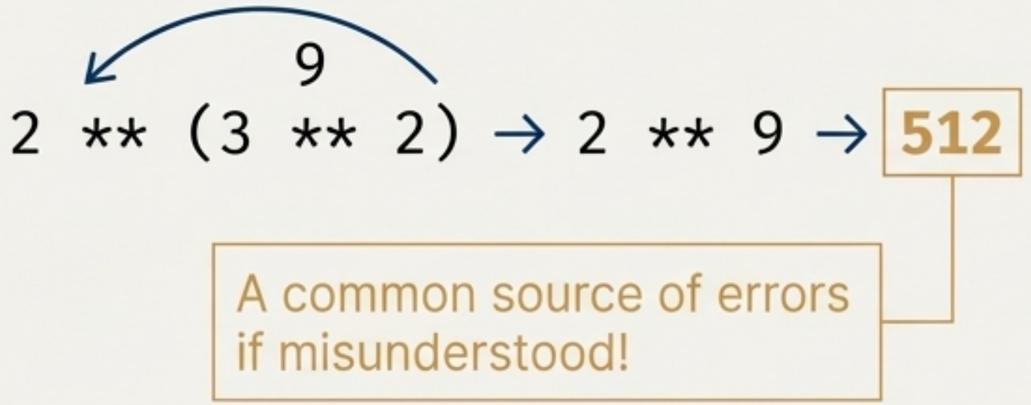
Left-to-Right

Most operators, including `+`, `-`, `*`, `/`, are left-associative.



Right-to-Left

The exponentiation operator `**` is the primary exception; it's right-associative.



Mastering the Grammar Lionar of Python

Understanding operators is fundamental to Python fluency. By mastering their function, precedence, and associativity, you move from simply writing code to constructing clear, predictable, and powerful logical expressions.

- **Manipulation:** Use Arithmetic and Assignment operators to calculate and store values.
- **Enquiry:** Use Comparison, Logical, Membership, and Identity operators to ask questions and make decisions. Remember: Remember: == is for value, `is` is for identity.
- **Grammar:** Precedence and Associativity are the rules that govern evaluation. Use parentheses `()` for absolute clarity and control.

The Python Operator Toolkit at a Glance

Precedence	Category	Operators	Description	Associativity
Highest	Exponent	`**`	Raises to the power of	Right-to-Left
	Arithmetic (Mult)	`*, /, //, %`	Multiplication, Division, Remainder	Left-to-Right
	Arithmetic (Add)	`+, -`	Addition, Subtraction	Left-to-Right
	Bitwise Shift	`<<, >>`	Bitwise shifts	Left-to-Right
	Bitwise AND	`&`		Left-to-Right
	Bitwise XOR	`^`		Left-to-Right
	Bitwise OR	`|`		Left-to-Right
	Enquiry	`==, !=, >, <, >=, <=, is, is not, in, not in`	Comparisons, Identity, Membership	Chained
	Logical	`not`	Logical NOT	N/A
	Logical	`and`	Logical AND	Left-to-Right
Lowest	Logical	`or`	Logical OR	Left-to-Right

A note: Assignment (`=`, `+=`, etc.) and the bitwise NOT (`~`) operator have their own specific places in the full precedence table but are grouped functionally in this deck for clarity.