Python Operator Precedence and Associativity

Table of contents

Syntax vs. semantics

Each rule below has two parts:

- **Syntax** — what makes the expression a *valid* Python program.
- **Semantics** — what the expression *evaluates to*.

Constants and brackets

- A constant (e.g. `2`) is the most basic expression.
- If `x` is a valid expression then `(x)` is also a valid expression — bracketed expressions get evaluated first. Whitespace inside brackets is ignored, and brackets can nest arbitrarily (`((2))`).
- An unmatched closing bracket (`2)`) is a `SyntaxError`.

Negation and affirmation

- If `x` is a valid expression then `+x` and `-x` are valid expressions.
- `+x` is equivalent to multiplying `x` by `1`; `-x` is equivalent to multiplying `x` by `-1`.
- These can be chained (`--3` is `3`, `+-+-+-3` is `-3`), but a trailing operator with nothing after it (`3+`) is a `SyntaxError`.

Arithmetic operators

If `x` and `y` are valid expressions, then all of the following are valid expressions, evaluated according to the rules of math:

Operator	Meaning
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Integer (floor) division
`x % y`	Remainder
`x ** y`	Exponentiation

Order of operations

From highest to lowest precedence:

Operator	Description
`()`	Parenthesis
`**`	Exponents
`-x`, `+x`	Negation, Affirmation
`*`, `/`, `//`, `%`	Multiplication, Division, Integer Division, Remainder
`+`, `-`	Addition, Subtraction

`*`, `/`, `//`, and `%` share a precedence level, as do `+` and `-` — see Associativity below for how ties between operators of equal precedence get resolved.

Associativity

An order of operations alone doesn't remove all ambiguity — e.g. `1 - 2 + 3` needs a rule for whether it means `(1 - 2) + 3` or `1 - (2 + 3)`. Python evaluates `1 - 2 + 3` as `(1 - 2) + 3 = 2`, so `+`/`-` are **left-associative**.

Operators of equal precedence are left-associative in general, **except exponentiation (`**`), which is right-associative**:

- `3 * 1 // 2` = `(3 * 1) // 2` = `1` (left-associative `*`/`//`)
- `2 ** 1 ** 0` = `2 ** (1 ** 0)` = `2` (right-associative `**`, *not* `(2 ** 1) ** 0 = 1`)

In general, if `#` and `@` are operators of equal precedence, `a # b @ c = (a # b) @ c` (left-associative case).

Division and floats

- `/` always returns a `float`, even when the division is exact (`4 / 2` is `2.0`, not `2`) — `type(2)` is `int`, `type(2.0)` is `float`.
- `1 / 3` gives an *approximate* result (`0.3333333333333333`) since floats have finite precision.
- `1 / 0` raises `ZeroDivisionError`; `1 / float('inf')` is `0.0`.

Exponentiation edge cases

- `2 ** 3` is `8` (int); `2 ** 3.0` is `8.0` (float) — a float exponent/base produces a float result.
- `2 ** -1` is `0.5`.
- `2 ** 0` and `0 ** 0` are both `1`.

Warning: `^` is not exponentiation

The caret `^` is Python's **bit-wise xor** operator, not exponentiation — e.g. `2 ^ 3` is `1` and `4 ^ 1` is `5`. Using `^` where you meant `**` does *not* raise an error, so this mistake can silently produce wrong results.

Integer division (the division algorithm)

For positive integers `x` and `y`, there is a unique quotient `q` and remainder `r` (with `0 <= r < y`) satisfying `x = q * y + r` — "grade school" division. E.g. for `x = 17`, `y = 3`: `17 = 5 * 3 + 2`, so `17 // 3` is `5` (the quotient) and `17 % 3` is `2` (the remainder), and `3 * (17 // 3) + (17 % 3) == 17`.

The division algorithm can be computed directly (repeated subtraction) or recursively:

```
def remainder(x: int, y: int) -> int:
    ans = x
    while ans >= y:
        ans = ans - y
    return ans
```

```
def remainder(x: int, y: int) -> int:
    return x if x < y else remainder(x - y, y)
```

Both give `remainder(17, 7) == 3`.