Python Operator Precedence and Associativity

Table of contents

Syntax vs. semantics . . . 1
Constants and brackets . . . 2
Negation and affirmation . . . 2
Arithmetic operators . . . 2
Order of operations . . . 2
Associativity . . . 3
Putting both rules on one expression . . . 3
Division and floats . . . 4
Exponentiation edge cases . . . 4
Warning: `^` is not exponentiation . . . 4
Integer division (the division algorithm) . . . 5

Syntax vs. semantics

Each *construct* below has two parts, and it is worth keeping them apart:

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

A syntax rule is always of the form "if `x` and `y` are valid Python expressions, then *this* is a valid Python expression" — which is what makes arbitrarily long arithmetic legal without a separate rule for every length. Precedence and associativity do not take this shape; they are plain rules.

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`.
- Style: write `3 + -3` rather than `3 + - 3`, and `2 - -2` rather than `2 - - 2` — group the sign with the number it belongs to.

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

`1 + 2 * 3 + 4` evaluates to `11`, not `13`, so `*` must happen before `+`. **Without an order of operations that expression is ambiguous.**

From highest to lowest precedence:

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

Two consequences of the middle row that catch people out: exponentiation outranks unary minus, and unary minus outranks multiplication. So `2 * -3` is `2 * (-3)` = `-6`, and `1 - -2 * -3 - 4` is `1 - ((-2) * (-3)) - 4` = `-9`.

`*`, `/`, `//`, and `%` share a precedence level, as do `+` and `-` — so the table by itself still leaves ties. See Associativity below.

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 `2`, matching `(1 - 2) + 3` rather than `1 - (2 + 3)` = `-4`, 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`, whereas `3 * (1 // 2)` = `0` (left-associative `*`/`//`)
- `2 ** 1 ** 0` = `2 ** (1 ** 0)` = `2`, whereas `(2 ** 1) ** 0` = `1` (right-associative `**`)

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

Putting both rules on one expression

The exam question `x = 35 * 2 % 5 ** 2` needs precedence *and* associativity to settle, which is why it is the one worth working slowly.

The other exam question resolves by precedence alone: `2 ** 3 % 5 - 1` is `((2 ** 3) % 5) - 1` = `(8 % 5) - 1` = `3 - 1` = `2`.

The tempting wrong answers to `35 * 2 % 5 ** 2` (`0`, `140` and `70`) are a good check on the table. Evaluating strictly left to right, ignoring precedence altogether, gives `((35 * 2) % 5) ** 2` = `0 ** 2` = `0`. Letting `%` bind first, then `**`, and leaving `*` until last gives `35 * ((2 % 5) ** 2)` = `35 * 4` = `140`. And `70` is just `35 * 2` with the rest of the expression dropped.

```
x = 35 * 2 % 5 ** 2
```
*** has the highest precedence, so it is bracketed first*

```
x = 35 * 2 % (5 ** 2)
```
** and % are equal precedence and left-associative, so the left-hand pair is bracketed first*

```
x = (35 * 2) % (5 ** 2)

x = 70 % 25

x = 20
```

Figure 1: Fully bracketing `x = 35 * 2 % 5 ** 2`, one rule at a time, to reach `(35 * 2) % (5 ** 2) = 70 % 25 = 20`. Each step boxes the tokens it brackets and names the rule in purple on the right; magenta marks the answer. Step one is precedence: `**` outranks everything else present. Step two is associativity: `*` and `%` sit on the same row of the table, so the tie is broken left-first. After that no choices remain.

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`.
- `6 / 2 * (1 + 2)` is `9.0` — brackets first, then `/` and `*` left to right, and the `/` makes the whole result a float.

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`. They are previews of python-while-loops[1] and python-recursion[2].

[1] courses/csse1001/python-while-loops.pdf
[2] courses/csse1001/python-recursion.pdf