Python as a Calculator

Table of contents

Outline . . . 1
Learning objectives . . . 2
The Python REPL . . . 2
Syntax vs. semantics . . . 2
Building the expression language . . . 3
Order of operations . . . 5
Associativity . . . 5
Division, floats, exponents . . . 6
Integer division . . . 7
Worked exam questions . . . 8
Variables . . . 8
`min` and `max` . . . 8
Exercise: second largest of three numbers . . . 9
Next lecture . . . 9

See csse1001[1] for course logistics. This is Lecture 1B. The same day's first hour is 2025-07-29-a-brief-history-of-computing[2] (Lecture 1A).

Outline

- The Python REPL
- Syntax vs. semantics
- Arithmetic expressions, order of operations, and associativity
- Variables
- Worked exam questions

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/2025-07-29-a-brief-history-of-computing.pdf

Learning objectives

1. Python expressions have syntax and semantics.
2. Python can evaluate arithmetic expressions, *but* this requires an order of operations and rules for associativity.

The Python REPL

A REPL (read-evaluate-print loop) is an interactive environment for a programmer to work in: it *reads* input, *evaluates* it, *prints* the result, then *loops*. A four-line transcript makes the acronym concrete, one letter per line:

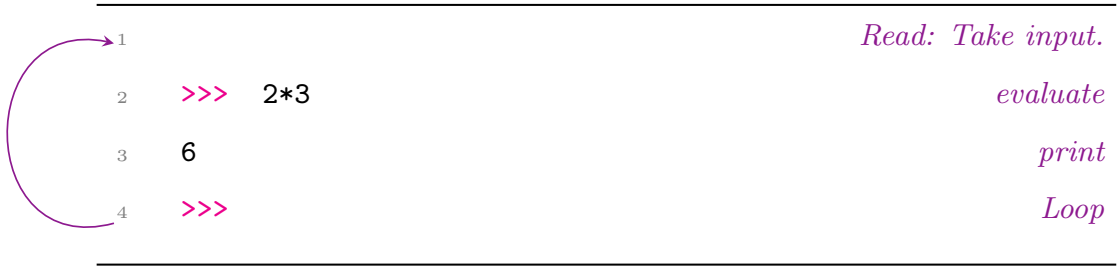

Figure 1: The four letters of read-evaluate-print-loop annotated onto the four lines of a transcript, with magenta `>>>` prompts and purple italic notes in the right margin. The return arrow is the loop. Only line 2 is typed by the user: line 3 is the printed result and line 4 is the prompt coming back.

The `>>>` prompt is only shown in the interactive REPL — it isn't included when writing Python instructions into files. Every transcript below is REPL output, so read `>>>` lines as what was typed and unprefixed lines as what Python printed back.

Syntax vs. semantics

A programming language like Python has both:

- **Syntax** — defines what it means to be a *valid program*.
- **Semantics** — defines what a valid program *actually does*.

`2 + 3` is syntactically correct Python and evaluates to `5`. Prefix notation like `+ 2 3` (valid in Lisp) is *not* valid Python syntax:

```
>>> + 2 3
    + 2 3
      ^
SyntaxError: invalid syntax
```

Python's language rules must be followed *precisely* for a program to run. Each new *construct* below has the same two parts: its **syntax**, which says what counts as a well-formed expression, and its **semantics**, which says what it evaluates to. See python-operator-precedence-and-associativity[3] for all of them collected together, plus the order-of-operations table, associativity, integer division, and the `^` xor gotcha.

Building the expression language

Arithmetic is built up one construct at a time, each with its own transcript.

Constants and brackets. A constant is the most basic expression; if `x` is valid then so is `(x)`, and bracketed expressions get evaluated first.

```
>>> 2
2
>>> (2)
2
>>> (  2   )                # space is ignored
2
>>> ((2))
2
>>> 2)
    2)
     ^
SyntaxError: unmatched ')'
```

Negation and affirmation. If `x` is valid then `+x` and `-x` are valid; affirming is multiplying by positive one, negating is multiplying by negative one. Both chain freely, and neither can trail off the end of an expression.

```
>>> ++3
3
>>> +(+(3))
3
>>> 3+
```

[3] courses/csse1001/python-operator-precedence-and-associativity.pdf

```
    3+
      ^
SyntaxError: invalid syntax
>>> --3
3
>>> -+3
-3
>>> +-+-+-3
-3
```

Adding and subtracting. As a matter of style, bind the sign to the number it belongs to rather than leaving it floating next to the operator:

```
>>> 3 + - 3
0
>>> 3 + -3                  # best to group like this
0
>>> + 1 + + + 3
4
>>> +1 + ++3                # best to group like this
4
>>> 2 - - 2
4
>>> 2 - -2                  # best to group like this
4
```

Multiplication, which is where evaluation order starts to matter:

```
>>> 2 + 3 * 2 - 1           # multiplication was done first
7
>>> (2 + 3) * 2 - 1         # brackets change outcome
9
>>> 1 + 2 * 3 + 4
11
```

That last line is the argument for needing a rule at all: since `1 + 2 * 3 + 4` is `11` and not `13`, `*` must be evaluated before `+`. Without an order of operations an expression like this is ambiguous.

Order of operations

For what has been introduced so far, the order is parentheses, unary `-x`/`+x`, `*`, then `+`/`-`. Once division, integer division, remainder and exponentiation are added, the full table becomes:

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

Highest precedence at the top. Worked examples:

```
>>> 10 - 2 * 3              # multiplication before subtraction
4
>>> 2 * - 3                 # negation before multiplication
-6
>>> 2 * -3                  # best to group according to precedence
-6
>>> 1 - - 2 * - 3 - 4
-9
>>> 1 - -2 * -3 - 4         # best to group according to precedence
-9
```

Associativity

Precedence alone is not enough, because some operators *share* a level:

```
>>> 1 - 2 + 3
2
>>> 1 - (2 + 3)
-4
>>> (1 - 2) + 3
2
```

Since Python's answer matches the left bracketing, `+` and `-` are **left-associative**. The rule: operators of the same precedence are left associative, *excepting exponentiation which is right associative* — so if `#` and `@` have equal precedence, `a # b @ c = (a # b) @ c`.

```
>>> 3 * 1 // 2
1
>>> 3 * (1 // 2)
0
>>> 2 ** 1 ** 0
2
>>> (2 ** 1) ** 0
1
```

The bracketing Python uses is `(3 * 1) // 2` for the first pair and `2 ** (1 ** 0)` for the second.

Division, floats, exponents

```
>>> 4 / 2                    # notice the decimal
2.0
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
>>> 6 / 2 * (1 + 2)
9.0
>>> 1 / 3                    # notice this is an approximate solution
0.3333333333333333
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 1 / float('inf')
0.0
>>> 2 ** 3.0
8.0
>>> 2 ** -1
0.5
>>> 0 ** 0
1
```

The `int`/`float` distinction is picked up properly next week in python-primitive-data-types[4].

Warning. The caret `^` is bit-wise xor, *not* exponentiation, and using it wrongly raises no error at all — `2 ^ 3` is `1` and `4 ^ 1` is `5`.

[4] courses/csse1001/python-primitive-data-types.pdf

Integer division

How can we divide two integers so that we get an integer?

Using "grade school" division: for `x = 17` and `y = 3` we say *17 ÷ 3 is 5 remainder 2*, or equivalently $17 = 5 \cdot 3 + 2$, or $\frac{17}{3} = 5 + \frac{2}{3}$.

```
>>> 17 // 3                  # quotient
5
>>> 17 % 3                   # remainder
2
>>> 3 * (17 // 3) + (17 % 3)
17
```

Division algorithm. For positive integers x and y there is a *unique* quotient q and a *unique* remainder r with $0 \leq r < y$ satisfying $x = q \cdot y + r$. The remainder can be computed two ways, both using machinery that comes later:

```
>>> def remainder(x: int, y: int) -> int:
...     ans = x
...     while ans >= y:
...         ans = ans - y
...     return ans
>>> remainder(17, 7)
3
```

```
>>> def remainder(x: int, y: int) -> int:
...     return x if x < y else remainder(x - y, y)
>>> remainder(17, 7)
3
```

The first is python-while-loops[5] (week 4); the second is python-recursion[6] (week 12).

[5] courses/csse1001/python-while-loops.pdf
[6] courses/csse1001/python-recursion.pdf

Worked exam questions

Q: What does `2 ** 3 % 5 - 1` evaluate to? (Options: 0, 2, 4, 7.)

`((2 ** 3) % 5) - 1 = (8 % 5) - 1 = 3 - 1 = 2`

Q: What value gets assigned to `x` in `x = 35 * 2 % 5 ** 2`? (Options: 0, 20, 70, 140, none of the above.)

`(35 * 2) % (5 ** 2) = 70 % 25 = 20`. python-operator-precedence-and-associativity[7] draws out the two steps that produce that bracketing.

Variables

Values can be named — these names are called **variables**. Assigning a value to a variable doesn't print anything in the REPL:

```
>>> width = 2
>>> height = 3
>>> area = width * height
>>> 4 * area
24
```

`min` and `max`

```
>>> max(1, 2)
2
>>> min(1, 2)
1
>>> max(5, 1, -2, 3, 2*7)
14
```

In the last one the arguments are themselves expressions, evaluated before `max` ever sees them.

[7] courses/csse1001/python-operator-precedence-and-associativity.pdf

Exercise: second largest of three numbers

Given three distinct numbers `a`, `b`, `c`, write an expression for the *second largest*. Three answers, set up with a multiple assignment:

```
>>> a, b, c = -9, 19, 3      # assignment like this saves vertical space
>>> a + b + c - max(a, b, c) - min(a, b, c)
3
>>> min( max(a, b), max(a, c), max(b, c) )
3
>>> max( min(a, b), min(a, c), min(b, c) )
3
```

Next lecture

1. 2025-08-04-python-memory-model[8] — Python memory model.
2. 2025-08-04-primitive-data[9] — Primitive data types.

[8] courses/csse1001/2025-08-04-python-memory-model.pdf
[9] courses/csse1001/2025-08-04-primitive-data.pdf