



# Python Language Reference

Python | Syntax, types, operators, control flow, functions and parameters, LEGB scopes, comprehensions, unpacking, match, f-strings, exceptions (Python 3.12-3.14)

#Python #Syntax #Reference #Python3.13

## How Python reads your code

Before Python language reference, can you explain...

|                                                                                                                  | Not yet                             | Partly                              | Yes                                 |
|------------------------------------------------------------------------------------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| <b>MUST KNOW</b> Run a .py file and use the REPL<br>Learn: 00-welcome workbench sheet                            | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            |
| <b>MUST KNOW</b> Variables, if, for, def in any language<br>Learn: any intro course                              | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            |
| <b>HELPFUL</b> The AI subset of Python (dataclasses, typing, uv)<br>Learn: programming/01, recalled not repeated | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> |
| <b>ADVANCED</b> Binary floating point<br>Learn: maths/12 numerical computing                                     | <input type="checkbox"/>            | <input type="checkbox"/>            | <input type="checkbox"/>            |

This sheet is a reference: it assumes you have already written small Python programs

Python is **dynamically typed** (a name can point to any object, the type lives on the object) and **strongly typed** ("3" + 3 raises TypeError, no silent conversion). Blocks are delimited by **indentation** (4 spaces by convention, PEP 8). Every value is an object with an identity (id(x)), a type (type(x)) and a value; assignment y = x binds a second name to the **same** object, it never copies. Current versions: 3.12 and 3.13 are the common production targets as of 2026, 3.14 is the newest stable release.

| Type              | Literal examples                      | Facts that bite                                             |
|-------------------|---------------------------------------|-------------------------------------------------------------|
| int               | 42, 0x1f, 0b1010, 1_000_000           | arbitrary precision: 10**20 + 1 is exact; never overflows   |
| float             | 3.0, 1e-3, float('inf'), float('nan') | IEEE 754 64-bit, about 15-17 significant digits; nan != nan |
| bool              | True, False                           | subclass of int: True + True == 2                           |
| complex           | 2 + 3j                                | z.real, z.imag, abs(z)                                      |
| str               | 'a', "a", '''multi''', r'\d+', b'raw' | immutable Unicode; bytes (b'') is a separate type           |
| NoneType          | None                                  | the single 'no value' object; test with is None             |
| Decimal, Fraction | Decimal('0.1'), Fraction(1, 3)        | exact decimal / rational maths from decimal and fractions   |

Built-in scalar types (containers are in programming/01)

**Falsy values** — False, None, 0, 0.0, 0j, "", b'', [], (), {}, set(), range(0); any object whose \_\_bool\_\_ returns False or \_\_len\_\_ returns 0. bool('0') and bool([0]) are True.

**round()** uses **banker's rounding** — Ties go to the even integer: round(2.5) = 2, round(3.5) = 4; round(0.125, 2) = 0.12 because 0.125 rounds to even in binary.

**int()** truncates toward zero — int(-3.9) = -3, while math.floor(-3.9) = -4 and -7 // 2 = -4 (floor division).

## Operators and control flow

| Operators            | Meaning                                   | Example → result                     |
|----------------------|-------------------------------------------|--------------------------------------|
| **                   | power, right-associative                  | 232 → 512; -2**2 → -4                |
| +x -x ~x             | unary plus, minus, bitwise not            | ~5 → -6                              |
| * / // % @           | mul, true div, floor div, modulo, matmul  | 7 // -2 → -4; -7 % 3 → 2             |
| + -                  | add, subtract (also concat for str, list) | [1] + [2] → [1, 2]                   |
| << >> & ^            | shifts, bitwise and, xor, or              | 1 << 10 → 1024; d1   d2 merges dicts |
| == != < <= in is not | comparisons, membership, identity; chain  | 0 < x < 1 means 0 < x and x < 1      |
| not, and, or         | boolean; and/or return an operand         | " or 'default' → 'default'           |
| a if c else b        | conditional expression                    | x if x > 0 else 0                    |
| lambda               | anonymous one-expression function         | key=lambda r: r['score']             |
| :=                   | assignment expression (walrus)            | if (n := len(xs)) > 5: ...           |

Operators from highest to lowest precedence (same row = same level)

### Modulo and floor division follow the divisor

a == (a // b) \* b + a % b always holds, and a % b takes the sign of b: -7 % 3 = 2, 7 % -3 = -2. C, Java and math.fmod take the sign of a instead. For negative indices this is what makes i % n a safe wrap-around.

```
control_flow.py: loops with else, strict zip, walrus (outputs in comments)
for i in range(3):
    if i == 5:
        break
else:
    # runs only if no break happened
    print("no break") # no break

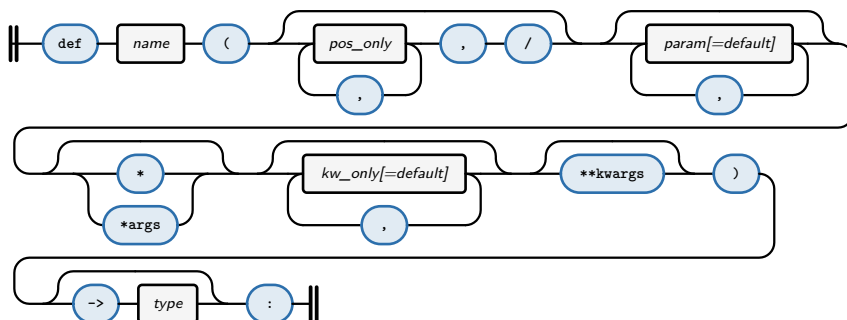
for i, (x, y) in enumerate(zip(xs, ys, strict=True), start=1):
    ... # strict=True (3.10): ValueError if lengths differ

while (line := f.readline()): # walrus: read until ""
    handle(line)

try:
    list(zip([1, 2], [3], strict=True))
except ValueError as e:
    print(e) # zip() argument 2 is shorter than argument 1
```

- **pass / ... (Ellipsis)** empty body placeholders; ... is also used in type stubs and NumPy slicing
- **for-else, while-else** else = 'loop finished without break'; ideal for search loops
- **range(start, stop, step)** lazy, stop excluded; len(range(0, 10, 3)) == 4
- **Short-circuit** 0 and 1/0 returns 0 without dividing; x and x.attr guards None

## ● Functions and parameters



Grammar of a parameter list, in the only order Python accepts. / ends the positional-only part, \* (or \*args) starts the keyword-only part

```
params.py: every parameter kind in one signature
def f(a, b=2, /, c=3, *args, d, e=5, **kw):
    return a, b, c, args, d, e, kw

f(1, d=4) # (1, 2, 3, (), 4, 5, {})
f(1, 2, 3, 4, 5, d=6, z=7)
# (1, 2, 3, (4, 5), 6, 5, {'z': 7})
f(a=1, d=4)
# TypeError: f() missing 1 required positional argument: 'a'
# (a=1 is not refused: it lands in **kw, since a is pos-only)
```

## Bind a call to a signature, by hand

**PROBLEM** With def f(a, b=2, /, c=3, \*args, d, e=5, \*\*kw), what does f(1, 2, 3, 4, 5, d=6, z=7) bind?

**GIVEN**

- a, b: positional-only (before /)
- c: positional-or-keyword
- \*args: extra positionals as a tuple
- d, e: keyword-only (after \*args); d has no default so it is required
- \*\*kw: extra keywords as a dict

### 1 Fill positional slots left to right

```
a=1, b=2, c=3
```

Why: three named positional slots exist before \*args

### 2 Collect surplus positionals

```
args = (4, 5)
```

Why: \*args takes everything after c

### 3 Match keywords to names

```
d=6; e keeps its default 5
```

Why: d and e can only be filled by name

### 4 Collect unknown keywords

```
kw = {'z': 7}
```

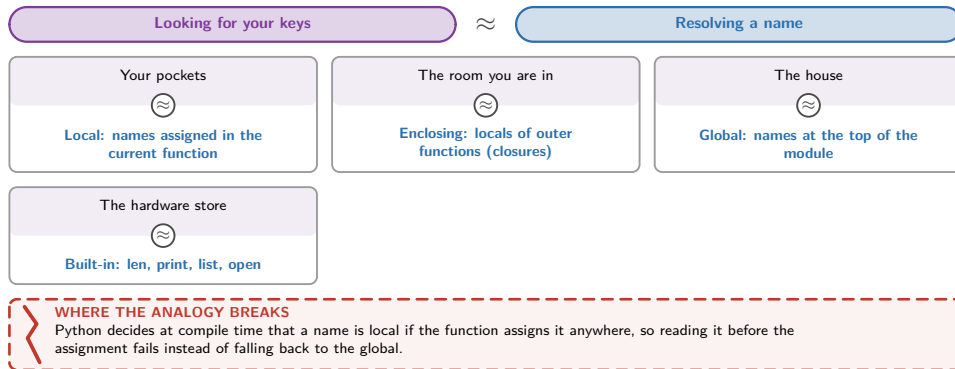
Why: z matches no parameter

**ANSWER** (1, 2, 3, (4, 5), 6, 5, {'z': 7})

**CHECK** ✓ Run it: the tuple printed by f equals the answer; drop d=6 and Python raises TypeError: missing 1 required keyword-only argument: 'd'

- **Call-site unpacking** f(\*xs, \*\*opts) spreads a list into positionals and a dict into keywords
- **Defaults are evaluated once** at def time; a list default is shared by every call (sheet 09)
- **Functions are objects** pass them, store them, return them; f.\_\_name\_\_, f.\_\_defaults\_\_, f.\_\_doc\_\_
- **lambda** one expression, no statements; use def when it needs a name or a docstring
- **return without value** returns None; a function that falls off its end also returns None

## Scopes, unpacking and comprehensions



LEGB name lookup: Python searches four places, innermost first, and stops at the first hit

scopes.py: nonlocal, global and the UnboundLocalError trap

```
x = "global"
def outer():
    x = "enclosing"
    def inner():
        nonlocal x      # rebind outer's x
        x = "changed"
    inner()
    return x
print(outer(), x)      # changed global
```

```
n = 0
def bump():
    n += 1              # n is local here: assigned
bump()
# UnboundLocalError: cannot access local variable 'n'
# where it is not associated with a value (fix: global n,
# or better, pass n in and return the new value)
```

unpack.py: star targets, nesting, swaps, merges

```
first, *rest = [1, 2, 3, 4]      # 1 [2, 3, 4]
*init, last = "abc"             # ['a', 'b'] 'c'
(a, b), c = (1, 2), 3           # nested: 1 2 3
a, b = b, a                     # swap, no temp variable
d1 = {"lr": 1e-3, "bs": 32}; d2 = {"bs": 64}
{**d1, **d2}                    # {'lr': 0.001, 'bs': 64}
d1 | d2                          # same (3.9); right side wins
print(*[1, 2, 3], sep=", ")     # 1, 2, 3
```

comprehensions.py: nesting order, filter vs conditional expression

```
nested = [[1, 2], [3], [4, 5]]
[v for row in nested for v in row]      # [1, 2, 3, 4, 5]
# read the for clauses left to right, as nested loops
```

```
[[r * c for c in range(1, 4)] for r in range(1, 3)]
# [[1, 2, 3], [2, 4, 6]] inner list per outer item
```

```
[x if x > 0 else 0 for x in [-1, 2, -3]] # [0, 2, 0]
# if-else BEFORE for transforms; if AFTER for filters
{v: k for k, v in {"a": 1, "b": 2}.items()} # {1: 'a', 2: 'b'}
sum(x * x for x in range(4)) # 14: no brackets needed
```

### INFO

A comprehension has its own scope: its loop variable does not leak ([i for i in range(3)] leaves no i behind), unlike a for statement. Since 3.12 (PEP 709) comprehensions are inlined for speed, with the same isolation.

### match, f-strings and exceptions

route.py: match on the shape of an LLM message (outputs below)

```
def route(msg):
    match msg:
        case {"role": "tool", "name": str(name),
              "args": dict(args)}:
            return f"call {name} with {sorted(args)}"
        case {"role": "assistant", "content": str(t)} if t.strip():
            return f"say {t!r}"
        case [x, y, *_]:
            return f"seq starting {x},{y}"
        case int() | float() as num if num < 0:
```

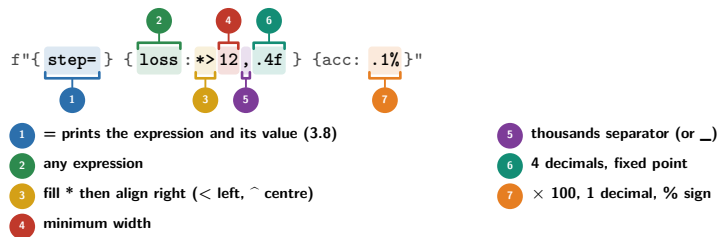
```
        return "negative"
        case _:
            return "ignore"
```

```
# {"role":"tool","name":"search","args":{"q":1,"k":2}}
#           -> call search with ['k', 'q']
# {"role":"assistant","content":"hi","id":7} -> say 'hi'
# {"role":"assistant","content":""} -> ignore (guard false)
# (1, 2, 3) -> seq starting 1,2           -2.5 -> negative
# "xy" -> ignore: str is never matched as a sequence
```

### Pattern

### Matches

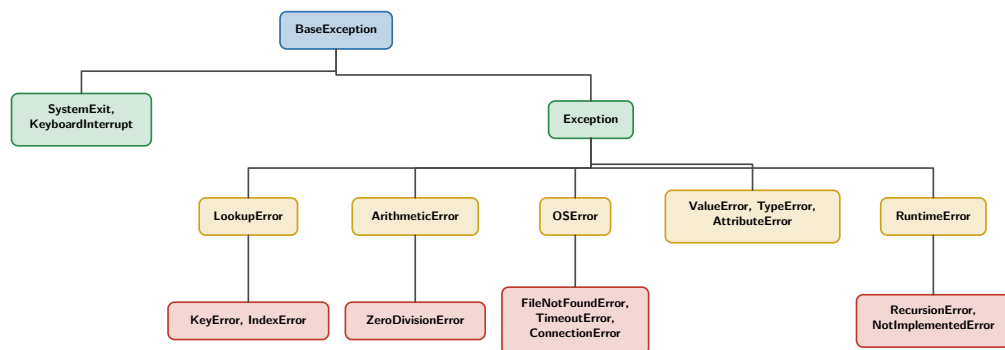
|                      |                                                            |
|----------------------|------------------------------------------------------------|
| 42, 'stop', None     | literal (None, True, False compared with is)               |
| name                 | anything, binds it (a bare name never compares!)           |
| Color.RED, mod.CONST | dotted name: compared with ==                              |
| [a, b, *rest]        | sequences (list, tuple) of length $\geq 2$ , not str/bytes |
| {'k': v, **rest}     | mappings having key 'k'; extra keys allowed                |
| Point(x=0, y=y)      | isinstance(Point) and attribute values                     |
| p1   p2, p as name   | alternatives; bind the whole match                         |



f-string anatomy: each field is `{expression[=][!conversion][.format_spec]}`. Output: `step=1200 *****0.0346 91.3%`

| Field                         | Output                 | Field                     | Output                |
|-------------------------------|------------------------|---------------------------|-----------------------|
| <code>{3e-4:.0e}</code>       | <code>3e-04</code>     | <code>{255:#x}</code>     | <code>0xff</code>     |
| <code>{1234567:,}</code>      | <code>1,234,567</code> | <code>{255:b}</code>      | <code>11111111</code> |
| <code>{1234567:_}</code>      | <code>1_234_567</code> | <code>{42:08.3f}</code>   | <code>0042.000</code> |
| <code>{'cat':!r:&gt;7}</code> | <code>'cat'</code>     | <code>{0.9132:.1%}</code> | <code>91.3%</code>    |

Format specs worth knowing (verified on Python 3.13)



The built-in exception tree (partial). `except` Parent catches every child; catch `Exception`, never `BaseException`

```

errors.py: try/else/finally, chaining, groups, notes
try:
    cfg = load("cfg.yaml")
except FileNotFoundError as err:    # narrowest first
    raise RuntimeError("config missing") from err
else:
    print("loaded")                # only if no exception
finally:
    print("always runs")           # cleanup

# repr: RuntimeError('config missing')
  
```

```

# __cause__: FileNotFoundError('cfg.yaml')

try:
    # 3.11+
    raise ExceptionGroup("batch",
        [ValueError("bad row"), TimeoutError("slow")])
except* ValueError as eg: print(eg.exceptions)
except* TimeoutError as eg: print(eg.exceptions)
# (ValueError('bad row'),) then (TimeoutError('slow'),)

e = ValueError("bad lr"); e.add_note("lr must be > 0")
  
```

### ⚠ Never write a bare except:

`except:` and `except BaseException:` also catch `KeyboardInterrupt` and `SystemExit`, so Ctrl-C cannot stop your training script. Catch the narrowest class you can handle, log it with `logger.exception(...)` (keeps the traceback), and re-raise with `raise` (no argument) when you cannot recover.

### 🚩 Myths, takeaways and self-test

#### MYTH

✗ Assignment copies the object

✗ case name: compares with a variable called name

✗ `round()` rounds .5 up

✗ Python ints overflow at  $2^{63}$

✗ `except Exception` catches Ctrl-C

#### FACT

✓ It binds another name to the same object

`b = a`; `b.append(1)` changes `a`. Copies are explicit: `list(a)`, `a.copy()`, `copy.deepcopy(a)`.

✓ A bare name captures anything

case limit: always matches and overwrites limit; use a dotted constant (`Config.LIMIT`) or a guard.

✓ It rounds half to even

`round(2.5) == 2`; use `decimal.Decimal` with `ROUND_HALF_UP` for money.

✓ Python ints are unbounded

Only NumPy and PyTorch fixed-width dtypes (`int64`, `int32`) overflow, and silently wrap.

✓ `KeyboardInterrupt` derives from `BaseException`

That is exactly why `except Exception` is the right broad catch.

## Key takeaways

01

**Names point to objects**

assignment never copies

02

**/ then \* shape a signature**

positional-only, normal, keyword-only

03

**LEGB, decided at compile time**

assigning makes a name local

04

**match tests shapes**

bare names capture; guards refine

05

**f"{x:=:.3f}" for debugging**

format spec: fill align width , .prec type

06

**Catch narrow, chain with from**

never bare except

**Q1 What does a / in a def line do?**

**A** Parameters before it are positional-only: callers cannot pass them by name

**Q2 Where does a=1 go in f(a=1) if a is positional-only and f has \*\*kw?**

**A** Into kw as {'a': 1}; Python then reports a missing positional argument

**Q3 Value of -7 // 2 and -7 % 2?**

**A** -4 and 1: floor division rounds down, modulo takes the divisor's sign

**Q4 What is the LEGB order?**

**A** Local, Enclosing, Global, Built-in: first match wins

**Q5 Why does n += 1 inside a function raise UnboundLocalError?**

**A** The assignment makes n local for the whole function, and it is read before being set

**Q6 Which case pattern never matches a str?**

**A** A sequence pattern such as [x, y, \* \_]: str and bytes are excluded

**Q7 f"{0.9132:.1%}" prints?**

**A** 91.3%

**Q8 What does raise X from err add?**

**A** Stores err as X.\_\_cause\_\_; the traceback shows 'The above exception was the direct cause'

**Q9 except\* is for?**

**A** Handling parts of an ExceptionGroup (3.11), for example errors from asyncio.TaskGroup

**Q10 What runs in for ... else?**

**A** The else block, only when the loop ended without break