Python

Version 3.14 · verified 2026-07-27

A gradual, compact walk through modern Python for web development — types, dataclasses, async, and packaging.

Contents

Mental model

Names are references: assignment binds a name to an object and never copies it, so two names can mutate the same list. Type hints are annotations the interpreter does not enforce — but web frameworks read them at import time to build validation and docs, which is why they are load-bearing in a FastAPI or Pydantic codebase rather than decoration. `async` buys concurrency on one thread, not parallelism: a single blocking call inside a coroutine stalls every other request the process serves.

Collections & comprehensions

Literal	Type
`[1, 2]`	list — ordered, mutable
`(1, 2)`	tuple — immutable, hashable
`{"a": 1}`	dict — insertion-ordered
`{1, 2}`	set — unique, unordered
`a | b`	merged dict, right wins

A comprehension builds a list, dict, or set in one expression and reads better than an append loop. Dicts preserve insertion order, so JSON round-trips keep their key order.

```
rows = [{"id": 1, "on": True},
        {"id": 2, "on": False}]
ids = [r["id"] for r in rows if r["on"]]
```

```
by_id = {r["id"]: r for r in rows}
print(ids, by_id[2]["on"])
# [1] False
```

> **Gotcha:** `b = a` binds a second name to the *same* list — `b.append(x)` changes `a` too. Copy with `a.copy()`, or `copy.deepcopy(a)` when nested.

Strings & f-strings

Form	Result
`f"{x}"`	interpolate `str(x)`
`f"{x!r}"`	interpolate `repr(x)`
`f"{x=}"`	debug — prints `x=value`
`f"{n:.2f}"`	two decimal places
`f"{n:,}"`	thousands separators

f-strings evaluate inline and are the default way to build log lines and messages. The `=` suffix prints both the expression and its value, which beats writing the name out twice while debugging.

```
name, total = "ada", 1234.5
print(f"{name.title()}: {total:,.2f}")
# Ada: 1,234.50
print(f"{total=}")
# total=1234.5
```

> **Warning:** never build SQL or HTML with an f-string. Interpolation happens before the driver sees the value, so it cannot escape it. Use query parameters instead.

Functions & arguments

Signature	Meaning
`def f(a, b=1)`	positional with default
`def f(*args)`	extra positionals, a tuple
`def f(**kw)`	extra keywords, a dict
`def f(*, key)`	keyword-only argument
`def f(a, /)`	positional-only argument

Defaults are evaluated once, when the `def` line runs — not per call. Keyword-only parameters after a bare `*` force callers to name the argument, which keeps a long signature readable at the call site.

```
def add(item, bucket=None):
  bucket = [] if bucket is None else bucket
  bucket.append(item)
  return bucket

print(add("a"), add("b"))
# ['a'] ['b']
```

> **Gotcha:** `def add(item, bucket=[])` evaluates `[]` **once**, at definition — every call then shares one list. Default to `None` and build it inside.

Type hints

Syntax	Meaning
`int | None`	union, since 3.10
`list[str]`	builtin generic, since 3.9
`type Id = int`	type alias, since 3.12
`def f[T](x: T) -> T`	generic, since 3.12
`TypedDict`	dict with fixed keys

Annotate what crosses a boundary — request bodies, return values, public functions — and let inference cover locals. `TypedDict` describes a JSON object's shape without building a class, which fits handler code that passes dicts straight through.

```
from typing import TypedDict


class User(TypedDict):
  id: int
  email: str


type UserId = int
```

```
def find(uid: UserId) -> User | None:
  return {"id": uid, "email": "a@b.c"}
```

> **Gotcha:** annotations are not checked at runtime — `find("oops")` runs happily. Only a type checker (mypy, pyright) or a validator like Pydantic catches it.

Dataclasses & JSON

Feature	Effect
`@dataclass`	generates init, repr, eq
`field(default_factory=list)`	fresh mutable default
`frozen=True`	immutable and hashable
`kw_only=True`	keyword-only fields, 3.10+
`asdict(obj)`	recursive plain dict

A dataclass turns a plain class into a data holder with no boilerplate — the right shape for request and response models when you are not already using Pydantic. `asdict` recurses into nested dataclasses.

```
import json
from dataclasses import dataclass, field, asdict


@dataclass
class User:
  email: str
```

```
  tags: list[str] = field(default_factory=list)


u = User("a@b.c", ["admin"])
print(json.dumps(asdict(u)))
# {"email": "a@b.c", "tags": ["admin"]}
```

> **Gotcha:** `json.dumps(u)` raises `TypeError: Object of type User is not JSON serializable`. Convert with `asdict()` first — `json` knows nothing about classes.

Errors & context managers

Construct	Use
`except A as e:`	bind the exception
`except A, B:`	several types, 3.14+
`else:`	runs when nothing raised
`finally:`	always runs
`raise X from err`	keep the original cause
`with open(p) as f:`	close on exit, even on error

A context manager guarantees cleanup on every exit path, which is how database sessions and HTTP clients avoid leaking. `@contextmanager` builds one from a generator: everything before `yield` is setup, the `finally` block is teardown.

```python
from contextlib import contextmanager


@contextmanager
def timer(label):
  print(f"start {label}")
  try:
    yield
  finally:
    print(f"end {label}")


with timer("query"):
  pass
# start query
# end query
```

> **Gotcha:** a bare `except:` also swallows `KeyboardInterrupt` and `SystemExit`, so Ctrl-C stops working. Catch `Exception` instead.

Async & the event loop

API	Behavior
`async def`	defines a coroutine function
`await x`	suspend until `x` finishes
`asyncio.run(main())`	start the loop, run to done
`TaskGroup`	concurrent; cancels on error, 3.11+
`gather(...)`	concurrent; siblings keep running

Calling a coroutine function returns a coroutine — it does nothing until awaited. Prefer `TaskGroup` over `gather`: when one task fails it cancels its siblings and raises an `ExceptionGroup`, so a failed request never leaves orphaned work running.

```python
import asyncio

async def fetch(n):
  await asyncio.sleep(0.1)
  return n * 2

async def main():
  async with asyncio.TaskGroup() as tg:
    a = tg.create_task(fetch(1))
    b = tg.create_task(fetch(2))
  print(a.result(), b.result())

asyncio.run(main())
# 2 4
```

> **Gotcha:** one blocking call (`requests.get`, `time.sleep`) inside a coroutine freezes every other request in the process. Use an async client, or hand the work to `asyncio.to_thread(fn)`.

Environments & packaging

Command	Purpose
`python -m venv .venv`	create an isolated env
`source .venv/bin/activate`	activate it (POSIX)
`python -m pip install -e .`	editable install
`python -m pip list`	what is actually installed
`uv sync`	resolve and install, fast

One virtual environment per project, never a shared one.

`pyproject.toml` is the single manifest: `[project]` holds metadata and dependencies, `[build-system]` names the backend that builds it.

```toml
[project]
name = "api"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = ["fastapi", "httpx>=0.27"]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

> **Gotcha:** a bare `pip` may belong to a different interpreter than the `python` you run, so the install lands where your code cannot see it. Always spell it `python -m pip`.

Further reading

- Python language reference
- What's new in Python 3.14
- typing — support for type hints
- asyncio — asynchronous I/O
- Python Packaging User Guide