



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	2
Collections & comprehensions	2
Strings & f-strings	3
Functions & arguments	5
Type hints	6
Dataclasses & JSON	8
Errors & context managers	10
Async & the event loop	11
Environments & packaging	13
Further reading	15

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
<code>[1, 2]</code>	list — ordered, mutable
<code>(1, 2)</code>	tuple — immutable, hashable
<code>{"a": 1}</code>	dict — insertion-ordered
<code>{1, 2}</code>	set — unique, unordered
<code>a b</code>	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
<code>f"{x}"</code>	interpolate <code>str(x)</code>

Form	Result
<code>f"{x!r}"</code>	interpolate <code>repr(x)</code>
<code>f"{x=}"</code>	debug — prints <code>x=value</code>
<code>f"{n:.2f}"</code>	two decimal places
<code>f"{n:,}"</code>	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
<code>def f(a, b=1)</code>	positional with default
<code>def f(*args)</code>	extra positionals, a tuple
<code>def f(**kw)</code>	extra keywords, a dict
<code>def f(*, key)</code>	keyword-only argument
<code>def f(a, /)</code>	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
<code>int None</code>	union, since 3.10
<code>list[str]</code>	builtin generic, since 3.9
<code>type Id = int</code>	type alias, since 3.12
<code>def f[T](x: T) -> T</code>	generic, since 3.12
<code>TypedDict</code>	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
<code>@dataclass</code>	generates <code>init</code> , <code>repr</code> , <code>eq</code>
<code>field(default_factory=list)</code>	fresh mutable default
<code>frozen=True</code>	immutable and hashable
<code>kw_only=True</code>	keyword-only fields, 3.10+
<code>asdict(obj)</code>	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
<code>except A as e:</code>	bind the exception
<code>except A, B:</code>	several types, 3.14+
<code>else:</code>	runs when nothing raised
<code>finally:</code>	always runs
<code>raise X from err</code>	keep the original cause
<code>with open(p) as f:</code>	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.

```
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
<code>async def</code>	defines a coroutine function
<code>await x</code>	suspend until <code>x</code> finishes
<code>asyncio.run(main())</code>	start the loop, run to done

API	Behavior
<code>TaskGroup</code>	concurrent; cancels on error, 3.11+
<code>gather(...)</code>	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.

```
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
<code>python -m venv .venv</code>	create an isolated env
<code>source .venv/bin/activate</code>	activate it (POSIX)
<code>python -m pip install -e .</code>	editable install

Command	Purpose
<code>python -m pip list</code>	what is actually installed
<code>uv sync</code>	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.

```
[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](#)