



Python for AI: the Subset That Matters

Programming | Types and containers, comprehensions, functions, classes and dataclasses, typing, generators, context managers, async basics, packaging with uv

#Python #Typing #uv #Async

Why a subset, and which one

Python in AI code

In one sentence
●○○

The glue language: it loads data, calls fast libraries and orchestrates training, while compiled code does the arithmetic.

Practitioner
●●○

A small set of features covers most AI code you will read.

- comprehensions and generators for data
- dataclasses and type hints for configs
- context managers for no_grad, files, timers
- dunder methods: Dataset is __len__ + __getitem__, a module is __call__

Expert
●●●

Performance comes from never looping over elements in Python: one call into NumPy or PyTorch replaces millions of interpreted steps.

- the GIL blocks CPU-parallel threads; use processes or vectorise
- asyncio helps only for waiting (HTTP, disk)

Analogy Python is the project manager who phones the specialists; it should never do the bricklaying itself.

Before Python for AI, can you explain...

	Not yet	Partly	Yes
MUST KNOW Variables, if, for, def Learn: any intro course	☐	☐	☐
MUST KNOW Run a script in a terminal Learn: 00-welcome workbench sheet	☐	☐	☐
HELPFUL What a class and an object are recalled below	☐	☐	☐
ADVANCED Threads vs processes only for the async section	☐	☐	☐

Types, containers and comprehensions

Type	Literal	Mutable	Typical AI use
int, float, bool	3, 3e-4, True	no	hyper-parameters, flags; float is 64-bit
str	"cat", f"{lr:.0e}"	no	labels, prompts, paths, run names
list	[1, 2, 3]	yes	ordered samples, metric history
tuple	(224, 224, 3)	no	shapes, fixed records, dict keys
dict	{"lr": 3e-4}	yes	configs, label → id maps, JSON

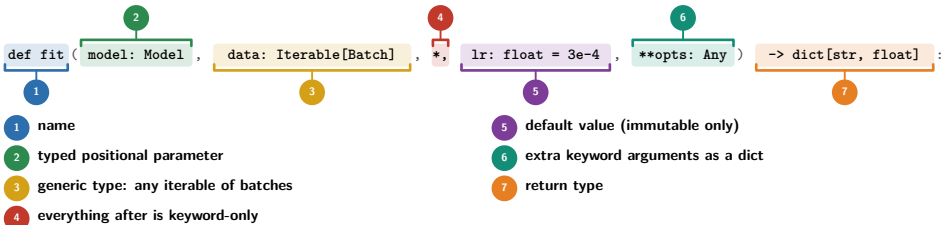
set	{"cat", "dog"}	yes	vocabularies, deduplication, fast 'in'
None	None	no	'not set yet'; test with 'is None'

```
comprehensions.py: the four shapes, with output in comments
labels = ["cat", "dog", "cat", "bird"]
vocab = sorted(set(labels)) # ['bird', 'cat', 'dog']
to_id = {w: i for i, w in enumerate(vocab)}
ids = [to_id[w] for w in labels] # [1, 2, 1, 0]
long = [w for w in labels if len(w) > 3] # ['bird']
sq = (i * i for i in range(10**9)) # generator: no memory used
pairs = list(zip(labels, ids)) # [('cat', 1), ('dog', 2), ...]
```

✗ Beginner mistake: `[[0] * 2] * 2` and copies that share data

`a = [[0] * 2] * 2; a[0][0] = 1` prints `[[1, 0], [1, 0]]`: the outer `*` copies the *reference* to one inner list. Build nested lists with a comprehension, `[[0] * 2 for _ in range(2)]`. Same family: `b = a` does not copy; use `copy.deepcopy(a)` for nested data.

⚡ Functions, classes and dataclasses



A modern function signature, part by part

⚠ Never use a mutable default

`def add(x, items=[])` creates the list **once**, at definition time: `add(1)`, `add(2)` returns `[1, 2]` `[1, 2]`. Write `items: list | None = None` then `items = []` if `items` is `None` else `items`. Dataclasses refuse tags: `list = []` and require `field(default_factory=list)` for the same reason.

```
config.py
Python
1 from dataclasses import dataclass, field, asdict
2
3
4 @dataclass(frozen=True, slots=True)
5 class TrainConfig:
6     lr: float = 3e-4
7     batch_size: int = 32
8     epochs: int = 10
9     tags: list[str] = field(default_factory=list)
10
11     def __post_init__(self) -> None:
12         if self.lr <= 0:
13             raise ValueError(f"lr must be > 0, got {self.lr}")
14
15
16 cfg = TrainConfig(lr=1e-3, tags=["baseline"])
17 print(asdict(cfg)) # plain dict: log it or dump to JSON
```

Line 4: frozen=True makes instances read-only (cfg.lr = 0.1 raises FrozenInstanceError), slots=True saves memory and catches typos in attribute names. Line 9: mutable default through a factory. Line 12: validate in __post_init__, so a bad config fails at start-up, not after three hours of training. Output: {'lr': 0.001, 'batch_size': 32, 'epochs': 10, 'tags': ['baseline']}

Dunder method	Makes this work	Where you meet it
__init__	Obj(...)	every class; nn.Module subclasses call super().__init__()
__len__, __getitem__	len(ds), ds[i]	a map-style torch Dataset is exactly these two
__iter__, __next__	for x in obj	iterable datasets, data streams
__call__	obj(x)	model(x) runs nn.Module.__call__, which calls forward plus hooks
__enter__, __exit__	with obj:	torch.no_grad(), open(), autocast
__repr__	print(obj)	readable logs; dataclasses generate it

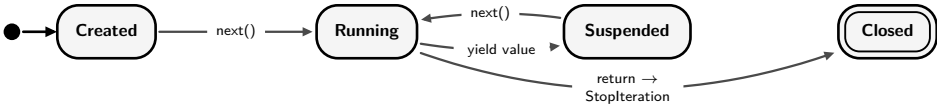
✓ Typing that tools can check

Write	Means	Since
list[int], dict[str, float]	built-in generics, no import	3.9
int None	union; replaces Optional[int]	3.10
Callable[[Tensor], Tensor]	a function from Tensor to Tensor	typing / collections.abc
Literal["adam", "sgd"]	only these exact values	3.8
TypedDict, Protocol	shape of a dict; 'anything with these methods'	3.8
def first[T](xs: list[T]) -> T	generic function, new syntax	3.12
type Batch = list[dict[str, int]]	type alias statement	3.12

TIP

Hints are **not enforced at run time**: TrainConfig(lr="0.1") runs and fails later. Run a checker in your editor and CI (mypy src or pyright), and validate external input (JSON, CLI, YAML) at the boundary with a library such as pydantic.

Generators, context managers and async



Life of a generator object. Nothing runs until the first next(); each yield pauses with all local variables kept

Stream a dataset into batches without loading it

PROBLEM

A JSONL file holds N = 1000 records. Yield batches of B = 32 records, never holding more than one batch in memory. How many batches, and how big is the last?

GIVEN

- N = number of records = 1000
- B = batch size = 32 records
- itertools.batched (Python 3.12+) groups any iterable into tuples of size B

1 Write a generator over the file

```
def records(p):
    with open(p) as f:
        for line in f:
            yield json.loads(line)
```

Why: one line in memory at a time, file closed by 'with'

2 Batch it lazily

```
for batch in batched(records(p), 32): train_step(batch)
```

Why: batched pulls 32 items, yields them, then pulls the next 32

3 Count batches

```
ceil(N / B) = ceil(1000 / 32) = ceil(31.25) = 32
```

Why: the last, partial batch still counts

4 Size of the last batch

```
N - B * floor(N / B) = 1000 - 32 * 31 = 8
```

Why: remainder of the division

ANSWER

32 batches: 31 of 32 records and a last one of 8

CHECK

✓ len(list(batched(range(1000), 32))) -> 32, and the last tuple has length 8; drop it (drop_last=True in PyTorch) if a step needs equal sizes

```
timer.py: write your own context manager
import time
from contextlib import contextmanager

@contextmanager
def timer(name: str):
    t0 = time.perf_counter()
    try:
        yield
    finally:
        print(f"{name}: {time.perf_counter() - t0:.2f}s")

with timer("epoch"):
    train_one_epoch()
```

```
fetch.py: 10 slow API calls at once, at most 5 in flight
import asyncio

async def fetch(i: int) -> int:
    await asyncio.sleep(1) # stands for an HTTP call
    return i * i

async def main() -> list[int]:
    sem = asyncio.Semaphore(5) # rate limit
    async def one(i: int) -> int:
        async with sem:
            return await fetch(i)
    return await asyncio.gather(*(one(i) for i in range(10)))

with timer("10 calls"):
    print(asyncio.run(main()))
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
# 10 calls: 2.00s (sequential: 10 s)
```

[x for x in xs]
list comprehension: builds everything now

vs

(x for x in xs)
generator expression: one at a time, single pass

Tell: Do you need to index it or iterate twice?

x == None
calls `__eq__`, can be overridden (NumPy returns an array)

vs

x is None
identity check, always a bool

Tell: Comparing with None, True, False: use is

threads / asyncio
help when waiting on I/O (HTTP, disk, LLM APIs)

vs

processes / vectorisation
help for CPU-bound Python work

Tell: Is the CPU busy or waiting?

dataclass
typed container, no validation of types

vs

pydantic model
parses and validates input at run time

Tell: Does the data come from outside the program?

Pairs people confuse in AI code

Packaging and projects with uv

A new AI project in four commands

```
$ uv init churn --python 3.12
Initialized project churn at /home/ana/churn

$ cd churn && uv add numpy polars torch
Resolved ... packages
+ numpy==2.x.y
+ polars==1.x.y
+ torch==2.x.y

$ uv add --dev pytest ruff mypy
+ pytest==... + ruff==... + mypy==...

$ uv run python -c "import numpy; print(numpy.__version__)"
2.x.y

$
```

uv init NAME	pyproject.toml, .python-version, main.py, git repo
uv add PKG / uv remove PKG	edit dependencies in pyproject.toml and update uv.lock
uv sync --locked	install exactly the lockfile into .venv (CI: fail if the lock is stale)
uv run CMD	run inside the project environment, syncing first; no activate needed
uv python install 3.12	download a managed interpreter
uvx ruff check .	run a tool in a throwaway environment
uv build && uv publish	build wheel + sdist, upload to an index

Commit `pyproject.toml`, `uv.lock` and `.python-version`; never commit `.venv/`. `uv.lock` pins every transitive package with hashes, so a colleague or CI gets the same set with `uv sync --locked`. PyTorch publishes CUDA-specific wheels on its own index: point uv at it with an `[[tool.uv.index]]` entry, as described in the uv PyTorch guide.

MYTH

✗ Type hints make Python check types at run time

→

FACT

✓ They are ignored when the program runs

✗ async makes my preprocessing faster

→

✓ async only overlaps waiting

✗ Threads never help in Python

→

✓ They help for I/O, and NumPy or PyTorch release the GIL inside heavy calls

✗ pip freeze is a lockfile

→

✓ It lists versions, without hashes or platform markers

Tadata | tadata.fr | contact@tadata.codes

Key takeaways

01 Python orchestrates, libraries compute
no element-wise loops over big data

02 Configs are frozen dataclasses
validated in `__post_init__`

03 Generators stream, comprehensions build
pick by memory and passes

04 'with' guarantees cleanup
files, `no_grad`, timers, locks

05 asyncio for waiting, not for computing
gather + Semaphore for API calls

06 uv: add, lock, sync --locked, run
commit pyproject.toml and uv.lock

Self-test

Q1 What does `@dataclass(frozen=True)` add?

A Generated `__init__`/`__repr__`/`__eq__` and read-only instances (assignment raises `FrozenInstanceError`)

Q2 How do you give a dataclass field a list default?

A `field(default_factory=list)`; a bare `[]` is refused because it would be shared

Q3 Which two dunder methods make a map-style PyTorch Dataset?

A `__len__` and `__getitem__`

Q4 What does `*` alone in a signature do?

A Makes every following parameter keyword-only

Q5 Generator vs list: which can you iterate twice?

A The list; a generator is exhausted after one pass (StopIteration)

Q6 When does asyncio speed things up?

A When tasks wait on I/O (HTTP, disk, model APIs), by overlapping the waits

Q7 Which uv command installs exactly the lockfile and fails if it is stale?

A `uv sync --locked`

Q8 Python 3.12 syntax for a generic function?

A `def first[T](xs: list[T]) -> T: ...`

Q9 Why `x` is `None` rather than `x == None`?

A `is` checks identity and always returns a bool; `==` can be overridden (NumPy returns an array)