INTERVIEWER'S COPY · ANSWERS INLINE

Python Interview Question Bank

Independent mid-level engineers, 3–4 years of professional Python

DE · Data Engineer | DS · Data Scientist | MLE · Machine Learning Engineer | BE · Python Backend Engineer | FS · Python Full Stack Engineer

74	162	4	5
full questions 54 master + 20 role-exclusive	theory questions 27 categories × 6 tiers	interview rounds role × round matrix	code-review exercises with injected bugs

Baselines: Python 3.12 · pandas 2.2 (3.0 differences flagged) · NumPy 2.x · FastAPI + Pydantic v2

Contents

How to use this bank 3

Part 1a · Coverage matrix 4

- Core-topic coverage (master bank) 6
- Theory bank index (Part 6) 7

Part 1b · EASY (E) 9

Part 1c · EASY & TRICKY (ET) 18

Part 2a · MEDIUM (M) 27

Part 2b · MEDIUM & TRICKY (MT) 38

Part 3a · HARD (H) 50

Part 3b · HARD & TRICKY (HT) 63

Part 4 · Role-wise section 77

- DE · Data Engineer 78
- DS · Data Scientist 85
- MLE · Machine Learning Engineer 90
- BE · Python Backend Engineer 97
- FS · Python Full Stack Engineer 104

Part 5 · Round-wise section 110

- Round definitions 110
- Role × round matrix 110
- Round 4 code-review exercises 111
- Interviewer guide 111

Part 6 · Theory and technical questions 113

- Core Python topics 113
- Role-specific topics 132

How to use this bank

What's inside. Part 1a is the coverage matrix of every question. Parts 1b–3b are the master bank of general Python questions, grouped by difficulty tier. Part 4 is the role-wise section: calibration notes, must-ask IDs, four role-exclusive questions per role, and red flags. Part 5 is the round-wise section: round definitions, a role × round matrix that assembles a complete loop for any role, code-review exercises, and the interviewer guide (hint policy, scoring, calibration). Part 6 is the theory bank: verbal questions for every core topic and every role, one per tier.

Tiers.

Tier	Meaning at 3–4 years	Time (output / coding)
E · Easy	Fluency; one concept. A one-year engineer should get it.	2–4 min / 8–12 min
ET · Easy & Tricky	Looks trivial; the obvious answer is wrong because of one mechanism	2–4 min / 8–12 min
M · Medium	The core bar: two concepts or careful tracing	4–8 min / 15–20 min
MT · Medium & Tricky	Two mechanisms interact to produce the trap	5–8 min / 15–20 min
H · Hard	A stretch: internals, concurrency, performance, design	8–12 min / 25–35 min
HT · Hard & Tricky	Deep, subtle behaviour; a differentiator, not a requirement	8–12 min / 30–35 min

Question format. Output-based questions give a complete runnable snippet, the exact expected output, a first-principles explanation, escalating follow-ups, and a Strong / Acceptable / Red-flag scoring line. Tricky questions add the trap answer candidates usually give and the real production bug the trap corresponds to. Coding tasks give the problem, signature, constraints, examples, graded hints, a type-hinted reference solution, complexity, assert-based tests, alternatives with a recommendation, follow-ups, and scoring.

Verified, not just traced. Every snippet, reference solution, and test in this document was executed before publication: CPython 3.12.3; pandas 2.2.3 (baseline) and 3.0.2; NumPy 2.x; Pydantic 2.13; FastAPI 0.14x; Jinja2 3.1. Expected outputs are copied from real runs. Behaviour that differs in other versions is flagged inline (3.10, 3.11, 3.13, 3.14; pandas 3.0). CPython-specific behaviour appears only in Hard & Tricky questions or questions explicitly about it, and is labelled.

Before the interview. Pick the round's questions from the role × round matrix in Part 5, and keep a per-candidate log of IDs so re-interviews use the listed alternates. Read the follow-ups in advance: they're where most of the signal comes from.

Part 1a · Coverage matrix

Every question in the bank, generated from the question headers themselves so the matrix can't drift from the content. Type: O = output-based, C = coding. Round: R1 Screening, R2 Core deep dive, R3 Role practical, R4 Advanced and production. Role-exclusive questions carry their own difficulty tier.

ID	Title	Tier	Type	Topics	Roles	Round
E-O01	Invoice line with f-string format specs	E	O	strings, f-string formatting	All	R1
E-O02	Merging config dicts: precedence and key order	E	O	dict behaviour, insertion order	All	R1
E-O03	try/except/else/finally in a record loader	E	O	exceptions, finally	All	R1
E-O04	Class vs instance attribute on an Invoice	E	O	OOP, attribute lookup	All	R1
E-O05	Stable multi-key sort of support tickets	E	O	sorting, stability, key functions	All	R1
E-O06	Tracing a lazy generator pipeline	E	O	generators, laziness	All	R1
E-C01	Batch records for a bulk API	E	C	iterators, generators, typing generics	All	R1
E-C02	Top error codes from log lines	E	C	Counter, sorting, parsing	All	R1
E-C03	Deep-merge layered configs without mutation	E	C	dicts, copying, recursion	All	R1
ET-O01	Rounding invoice amounts	ET	O	float precision, banker's rounding, Decimal	All	R1
ET-O02	Stripping file extensions	ET	O	strings	All	R1
ET-O03	A refund check that eats the iterator	ET	O	iterators, exhaustion, short-circuiting	All	R1
ET-O04	Deduplicating order keys in a set	ET	O	`__eq__`/`__hash__` contract, hashing	All	R1
ET-O05	Per-job config from shared defaults	ET	O	shallow copy, mutation vs rebinding, `**kwargs`	All	R1
ET-O06	Type hints on a line-total function	ET	O	typing, runtime behaviour	All	R1
ET-C01	Split an amount into exact installments	ET	C	Decimal, integer arithmetic, divmod	All (DE, BE, FS emphasis)	R1
ET-C02	Deduplicate user-entered tags	ET	C	Unicode normalisation, casefold, sets	All	R1
ET-C03	Leaderboard with tie-breaks	ET	C	multi-key sort, `reverse`, stability	All	R1
M-O01	Stacked decorators on an audited handler	M	O	decorators with arguments, `functools.wraps`, stacking order	All	R2
M-O02	A contextmanager transaction that suppresses errors	M	O	context managers, `contextlib`, exception suppression	All	R2
M-O03	Event publisher signature: *args, keyword-only, unpacking	M	O	functions, `*args`/`**kwargs`, keyword-only args, unpacking	All	R2
M-O04	Exception chaining in a payment client	M	O	exception chaining, custom exceptions	All	R2
M-O05	Dataclass defaults, equality and ordering	M	O	dataclasses, generated dunder methods	All	R2
M-O06	mock.patch where the name is looked up	M	O	testing, `unittest.mock`, imports and module namespaces	All (BE/FS must-ask)	R2
M-C01	Retry decorator with exponential backoff	M	C	decorators, exceptions, testability	All	R2
M-C02	Inventory reconciliation with Counter	M	C	`collections.Counter`, multiset arithmetic	All	R2
M-C03	Order pipeline tasks by dependency	M	C	graphs, topological sort, `deque` (the tier's one algorithm question)	DE, BE, MLE	R2

MT-O01	Per-column validators built in a comprehension	MT	O	closures, late binding, comprehension scope	All	R2
MT-O02	Default headers leaking between requests	MT	O	mutable default arguments, aliasing	All	R2
MT-O03	Processed-count helper and UnboundLocalError	MT	O	scoping (LEGB), `global/nonlocal`	All	R2
MT-O04	Purging expired sessions while iterating	MT	O	iteration + mutation (lists and dicts)	All	R2
MT-O05	Grouping events with itertools.groupby	MT	O	`itertools.groupby`, shared iterators	All	R2
MT-O06	Spot the bug: a cleanup helper that swallows errors	MT	O	exceptions, `finally` semantics	All	R2
MT-C01	Memoize functions that take lists and dicts	MT	C	`functools.lru_cache`, hashing, mutability	All (MLE, DS emphasis)	R2
MT-C02	Decode lines from a chunked byte stream	MT	C	UTF-8, incremental decoding, generators	DE, BE, MLE, FS	R2
MT-C03	Typed settings from environment variables	MT	C	runtime types vs hints, truthiness, parsing	All	R2
H-O01	ThreadPoolExecutor.map ordering and failures	H	O	threads, futures, GIL	DE, BE, MLE, FS	R4
H-O02	A blocking call inside a coroutine	H	O	asyncio event loop, blocking, `gather`	BE, FS, MLE, DE	R4
H-O03	yield from and a sub-generator's return value	H	O	generators, `yield from`, `StopIteration.value`	All	R4
H-O04	`__slots__` on a high-volume event record	H	O	`__slots__`, `dataclass(slots=True)`, attribute storage	All	R4
H-O05	Measuring memory with sys.getsizeof	H	O	memory, `sys.getsizeof`, generators vs lists (CPython 64-bit)	All	R4
H-O06	Circular import between orders and pricing	H	O	import system, `sys.modules`, circular imports	All	R4
H-C01	Bounded-concurrency async fetcher	H	C	asyncio, Semaphore, timeouts, cancellation	BE, FS, DE, MLE	R4
H-C02	Thread-safe TTL cache decorator	H	C	threads, locks, `OrderedDict`, `functools.wraps`	BE, MLE, DE	R4
H-C03	External sort of a file larger than memory	H	C	memory, generators, `heapq.merge`, `contextlib.ExitStack`, `tempfile`	DE, MLE, BE	R4
HT-O01	Cooperative super() across pipeline mixins	HT	O	MRO, `super()`, cooperative inheritance	All	R4
HT-O02	cached_property vs property on a model wrapper	HT	O	descriptors, properties, attribute precedence	All (MLE emphasis)	R4
HT-O03	send/throw/close on a running aggregator	HT	O	generator protocol, `send`, `throw`, `close`, `GeneratorExit`	All	R4
HT-O04	TaskGroup failure, cancellation and except*	HT	O	asyncio cancellation, `TaskGroup`, exception groups (3.11+)	BE, FS, MLE, DE	R4
HT-O05	`__del__` timing and reference cycles (CPython-specific)	HT	O	reference counting, cyclic GC, finalizers (CPython-specific)	All	R4
HT-O06	Status-code identity check (CPython-specific)	HT	O	object identity, `is` vs `==`, small-int cache (CPython-specific)	All	R4
HT-C01	Async worker pool with graceful shutdown	HT	C	`asyncio.Queue`, cancellation, graceful shutdown	BE, FS, DE	R4
HT-C02	Nested transactions with savepoints	HT	C	context managers, exceptions, `sqlite3`	BE, DE, FS	R4

				transactions		
HT-C03	Per-instance memoization that doesn't leak	HT	C	memory, weak references, GC, caching	MLE, BE, DE	R4
DE-O01	Comparing naive and aware timestamps	MT	O	datetime, timezones, DST	DE	R3
DE-O02	csv.DictReader on malformed rows	M	O	CSV quoting, ragged rows	DE	R3
DE-C01	SCD Type 2 merge for a customer dimension	H	C	SCD2, idempotent reruns, validation	DE	R3
DE-C02	Sessionise clickstream events	M	C	sessionisation, windowing, timezone-aware parsing	DE	R3
DS-O01	Broadcasting and dtype casting in feature scaling	MT	O	NumPy broadcasting, dtypes, in-place casting	DS, MLE	R3
DS-O02	Chained assignment under 2.2 vs Copy-on-Write	HT	O	views vs copies, SettingWithCopy, Copy-on-Write	DS, DE	R3
DS-O03	NaN in groupby keys and aggregations	M	O	NaN semantics, groupby, dtypes	DS	R3
DS-C01	Leakage-free rolling features per customer	M	C	groupby/shift/rolling, train–serve leakage	DS, MLE	R3
MLE-O01	In-place normalisation through NumPy views	MT	O	views vs copies, memory, contiguity	MLE, DS	R3
MLE-O02	Seeding random, NumPy and generators	M	O	reproducibility, RNG state	MLE, DS	R3
MLE-C01	FastAPI inference service with startup model loading	H	C	model serving, Pydantic v2 validation, sync vs async endpoints	MLE	R3
MLE-C02	Top-k cosine similarity search	M	C	embeddings, vectorised NumPy, argpartition, normalisation	MLE, DS	R3
BE-O01	Pydantic v2 coercion, strict mode and dumping	M	O	Pydantic v2 validation and serialisation	BE, FS	R3
BE-O02	sqlite3 transactions with the connection context manager	MT	O	transactions, rollback, connection lifecycle	BE, DE	R3
BE-C01	Per-client token-bucket rate limiter	H	C	rate limiting, locks, injectable clocks	BE, FS	R3
BE-C02	Idempotent order-creation endpoint with dependency injection	H	C	FastAPI dependency injection, status codes, idempotency, testing	BE, FS	R3
FS-O01	json.dumps on datetimes, Decimals, sets and NaN	M	O	JSON serialisation edge cases	FS, BE	R3
FS-O02	Offset pagination under concurrent inserts	M	O	offset vs cursor pagination	FS, BE	R3
FS-O03	Template autoescaping and XSS	M	O	Jinja2 autoescaping, `Markup`, XSS contexts	FS	R3
FS-C01	Cursor-paginated feed endpoint	H	C	keyset pagination, opaque cursors, UTC ISO 8601, FastAPI	FS, BE	R3

Core-topic coverage (master bank)

Core topic	Master-bank IDs	Theory
Data model, object identity	HT-O06, HT-O05	TH-DM
Mutability, aliasing, copying	ET-O05, E-C03, MT-O02, MT-O04	TH-MUT
Scoping (LEGB, global/nonlocal)	MT-O03	TH-SCO
Closures, late binding	MT-O01	TH-CLO
Functions (defaults, `*args`/`**kwargs`, keyword-only, unpacking)	M-O03, MT-O02, ET-O05	TH-FN

Comprehensions and their scope	MT-O01	TH-CMP
Iterators, generators (laziness, exhaustion, `yield from`, send/close)	E-O06, E-C01, ET-O03, H-O03, HT-O03, MT-C02	TH-GEN
Decorators	M-O01, M-C01, H-C02	TH-DEC
Context managers	M-O02, H-C03, HT-C02	TH-CTX
Exceptions (chaining, finally, custom, groups)	E-O03, M-O04, MT-O06, HT-O04	TH-EXC
OOP (attributes, MRO, properties, dunders, eq/hash, dataclasses, slots, descriptors)	E-O04, ET-O04, M-O05, H-O04, HT-O01, HT-O02	TH-OOP
Typing (runtime behaviour, Protocol, generics)	ET-O06, E-C01, MT-C03	TH-TYP
Strings and bytes	E-O01, ET-O02, ET-C02, MT-C02	TH-STR
Numbers (float, rounding, negative `//` and `%`, Decimal)	ET-O01, ET-C01	TH-NUM
collections / itertools / functools	E-C02, M-C02, MT-O05, MT-C01, H-C02	TH-COL
Sorting	E-O05, E-C02, ET-C03	TH-SRT
Hashing, dict/set behaviour	E-O02, ET-O04, MT-C01	TH-HSH
Concurrency (GIL, threads vs processes vs asyncio)	H-O01, H-C02	TH-CON
asyncio	H-O02, H-C01, HT-O04, HT-C01	TH-ASY
Memory (refcount, GC, getsizeof)	H-O05, HT-O05, HT-C03	TH-MEM
Modules and imports	M-O06, H-O06	TH-MOD
Testing (fixtures, mocking)	M-O06, M-C01	TH-TST

Theory bank index (Part 6)

Six questions per category, one per tier: suffix -01 is Easy, -02 Easy & Tricky, -03 Medium, -04 Medium & Tricky, -05 Hard, -06 Hard & Tricky.

Category	Topic	IDs (E → HT)
TH-DM	Data model and object identity	TH-DM-01 … TH-DM-06
TH-MUT	Mutability, aliasing and copying	TH-MUT-01 … TH-MUT-06
TH-SCO	Scoping (LEGB, global, nonlocal)	TH-SCO-01 … TH-SCO-06
TH-CLO	Closures and late binding	TH-CLO-01 … TH-CLO-06
TH-FN	Functions and arguments	TH-FN-01 … TH-FN-06
TH-CMP	Comprehensions	TH-CMP-01 … TH-CMP-06
TH-GEN	Iterators and generators	TH-GEN-01 … TH-GEN-06
TH-DEC	Decorators	TH-DEC-01 … TH-DEC-06
TH-CTX	Context managers	TH-CTX-01 … TH-CTX-06
TH-EXC	Exceptions	TH-EXC-01 … TH-EXC-06
TH-OOP	Object-oriented Python	TH-OOP-01 … TH-OOP-06
TH-TYP	Typing	TH-TYP-01 … TH-TYP-06
TH-STR	Strings and bytes	TH-STR-01 … TH-STR-06
TH-NUM	Numbers	TH-NUM-01 … TH-NUM-06
TH-COL	collections, itertools, functools	TH-COL-01 … TH-COL-06
TH-SRT	Sorting	TH-SRT-01 … TH-SRT-06
TH-HSH	Hashing, dicts and sets	TH-HSH-01 … TH-HSH-06
TH-CON	Concurrency (GIL, threads, processes)	TH-CON-01 … TH-CON-06
TH-ASY	asyncio	TH-ASY-01 … TH-ASY-06

TH-MEM	Memory management	TH-MEM-01 … TH-MEM-06
TH-MOD	Modules and imports	TH-MOD-01 … TH-MOD-06
TH-TST	Testing	TH-TST-01 … TH-TST-06
TH-DE	Data Engineer	TH-DE-01 … TH-DE-06
TH-DS	Data Scientist	TH-DS-01 … TH-DS-06
TH-MLE	Machine Learning Engineer	TH-MLE-01 … TH-MLE-06
TH-BE	Python Backend Engineer	TH-BE-01 … TH-BE-06
TH-FS	Python Full Stack Engineer	TH-FS-01 … TH-FS-06

Part 1b · EASY (E)

Fluency checks. One concept each. A one-year engineer should get these; a four-year engineer should get them fast and explain them cleanly.

E-O01 · Invoice line with f-string format specs

Easy · Output · All roles · Round 1 · strings, f-string formatting · 3 min

Question: What does this print?

```
customer = "Asha"
amount = 1234567.891
discount = 0.125
order_id = 42
print(f"{customer!r} owes {amount:,.2f}")
print(f"discount={discount:.1%}")
print(f"[{order_id:>6}] [{order_id:<6}] [{order_id:06d}]")
print(f"{amount=:.1f}")
```

Expected output:

```
'Asha' owes 1,234,567.89
discount=12.5%
[    42] [42    ] [000042]
amount=1234567.9
```

Explanation: Each `{expr:spec}` evaluates `expr` at runtime and calls `format(value, spec)`. The `!r` conversion applies `repr()` first, which is why `Asha` appears in quotes. In `,.2f`, the comma adds thousands separators and `.2f` gives fixed-point with two decimals; the rounding happens only in the output text and the float itself is unchanged. `.1%` multiplies by 100, formats with one decimal, and appends `%`. `>6` and `<6` right- and left-align in a field of width 6; `06d` zero-pads an integer. `{amount=:.1f}` (3.8+) prints the expression's source text, `=`, then the formatted value, a cheap debug-logging idiom. Takeaway: format specs control presentation only; never "round money" by formatting a float (see ET-O01).

Follow-ups:

1. "What changed about f-strings in 3.12?" PEP 701 lets an f-string reuse its own quote type inside the braces (`f"{row["sku"]}"`) and allows multi-line expressions with comments. Before 3.12 the nested-quote version is a SyntaxError.
2. "How would you print 1234567.891 as 12,34,567.89 for an Indian invoice?" The format mini-language only groups in thousands. Use a locale-aware library (Babel's `format_decimal(..., locale="en_IN")`) or a small helper. The signal is knowing where the built-in spec stops.

Scoring: Strong: all four lines exact, including the quotes from `!r` and `12.5%`. Acceptable: slips on padding direction and self-corrects when asked. Red flag: believes `:.2f` changes the stored value, or cannot read a format spec.

E-O02 · Merging config dicts: precedence and key order

Easy · Output · All roles · Round 1 · dict behaviour, insertion order · 3 min

Question: What does this print?

```
defaults = {"timeout": 30, "retries": 3, "region": "ap-south-1"}
overrides = {"retries": 5, "debug": True}
config = {**defaults, **overrides}
print(config)
print(config.get("verbose", False), config.get("debug", False))
del config["timeout"]
config["timeout"] = 60
```

```
print(list(config))
print(defaults | {"region": "eu-west-1"} == {**defaults, "region": "eu-west-1"})
```

Expected output:

```
{'timeout': 30, 'retries': 5, 'region': 'ap-south-1', 'debug': True}
False True
['retries', 'region', 'debug', 'timeout']
True
```

Explanation: `{**defaults, **overrides}` builds a new dict by inserting keys left to right. A repeated key keeps its original position but takes the later value, so `retries` stays second with value 5 and `debug` is appended. Insertion order is a language guarantee since 3.7. `.get(key, default)` never raises. Updating an existing key keeps its position, but deleting and re-inserting it moves it to the end. `a | b` (3.9+) is the operator form of the same merge, right side wins, and returns a new dict. Takeaway: right-most value wins; first insertion decides position.

Follow-ups:

1. "Does this deep-merge nested dicts?" No. Nested values are replaced wholesale and the result shares inner objects with the inputs (a shallow merge). Bridges to E-C03 and ET-O05.
2. "`config = config.update(extra)`. What's in `config`?" `None`. `update` mutates in place and returns `None`. Use `|` for a new dict, or `|=`/`update` without reassignment.

Scoring: Strong: exact output including the moved `timeout`. Acceptable: values right, misses the reorder, explains it once shown. Red flag: thinks dict order is random in modern Python.

E-O03 · try/except/else/finally in a record loader

Easy · Output · All roles · Round 1 · exceptions, finally · 3 min

Question: What does this print?

```
def load(record):
    try:
        print("parse", record)
        value = int(record)
    except ValueError:
        print("bad record")
        return None
    else:
        print("ok")
        return value
    finally:
        print("cleanup")

print(load("7"))
print(load("seven"))
```

Expected output:

```
parse 7
ok
cleanup
7
parse seven
bad record
cleanup
None
```

Explanation: The `try` body runs; if it raises nothing, `else` runs; `finally` always runs last, whether the block exits normally, through `return`, or through an exception. For `"7"`, `else` evaluates `return value`: the 7 is computed and held, `finally` prints `cleanup`, and only then does control return to the caller, which prints 7. For `"seven"`, `int()` raises ValueError, the handler prints and prepares `return None`, `finally` runs, and the caller prints `None`. Code lives in

`else` rather than `try` to keep the `try` narrow, so a ValueError raised by success-path code isn't misreported as "bad record". Takeaway: `finally` runs after the return expression is evaluated but before control leaves the function.

Follow-ups:

1. "What if `finally` also contained `return -1`?" It overrides both returns and swallows any in-flight exception (MT-O06). 3.14 emits a SyntaxWarning for this (PEP 765).
2. "What does `load(None)` do?" Prints `parse None`; `int(None)` raises TypeError, which `except ValueError` doesn't match; `finally` still prints `cleanup`; the TypeError propagates.

Scoring: Strong: exact order and explains when the return value is captured. Acceptable: puts `cleanup` after the returned value, fixes it after one hint. Red flag: believes `finally` is skipped when a `return` executes.

E-O04 · Class vs instance attribute on an Invoice

Easy · Output · All roles · Round 1 · OOP, attribute lookup · 3 min

Question: What does this print?

```python
class Invoice:
    currency = "INR"

    def __init__(self, amount):
        self.amount = amount

a = Invoice(100)
b = Invoice(200)
a.currency = "USD"
Invoice.currency = "EUR"
print(a.currency, b.currency, Invoice.currency)
print("currency" in vars(a), "currency" in vars(b))
del a.currency
print(a.currency)
```

Expected output:

```
USD EUR EUR
True False
EUR
```

Explanation: Reading `obj.attr` checks the instance's `__dict__` first, then the class and its MRO (data descriptors such as `property` are the exception; see HT-O02). `a.currency = "USD"` always writes to `a`'s own `__dict__`; the class is untouched. `Invoice.currency = "EUR"` rebinds the class attribute, which every instance without its own entry sees, so `b` reads EUR. `vars()` confirms only `a` has an instance entry. `del a.currency` removes that entry, so lookup falls through to the class again: EUR. Takeaway: assignment through an instance writes to the instance; reads fall back to the class.

Follow-ups:

1. "If the class had `tags = []` and we did `a.tags.append("gst")`, what is `b.tags`?" `['gst']`. No assignment happened, so the single class-level list was mutated and every instance shares it. Fix: create it in `__init__`, or `field(default_factory=list)` in a dataclass.
2. "With a class-level `count = 0`, what does `a.count += 1` do?" Reads the class's 0, writes 1 into `a.__dict__`. The class stays 0; other instances still see 0.

Scoring: Strong: exact output and explains lookup order. Acceptable: line 1 right, hesitates on `del`. Red flag: thinks `a.currency = ...` changes it for all instances.

E-O05 · Stable multi-key sort of support tickets

Easy · Output · All roles · Round 1 · sorting, stability, key functions · 4 min

Question: What does this print?

```
tickets = [
    ("T1", "high", 3),
    ("T2", "low", 1),
    ("T3", "high", 1),
    ("T4", "medium", 2),
    ("T5", "low", 2),
]
rank = {"high": 0, "medium": 1, "low": 2}
by_priority = sorted(tickets, key=lambda t: rank[t[1]])
print([t[0] for t in by_priority])
by_age = sorted(by_priority, key=lambda t: t[2], reverse=True)
print([t[0] for t in by_age])
print([t[0] for t in sorted(tickets, key=lambda t: (rank[t[1]], -t[2]))])
```

Expected output:

```
['T1', 'T3', 'T4', 'T2', 'T5']
['T1', 'T4', 'T5', 'T3', 'T2']
['T1', 'T3', 'T4', 'T5', 'T2']
```

Explanation: Python's sort is stable: items whose keys compare equal keep their input order, so within "high" T1 stays ahead of T3 and within "low" T2 stays ahead of T5. The second sort orders by age, descending. `reverse=True` is specified to preserve stability (equal keys are not flipped), so the age-2 tie keeps T4 before T5 and the age-1 tie keeps T3 before T2, their `by_priority` order. The third sort uses a tuple key, compared element by element; negating the numeric age gives "priority ascending, then age descending" in one pass. Takeaway: multi-key sort = tuple key (negate numerics for descending), or successive stable sorts from least to most significant key.

Follow-ups:

1. "What if we'd used `sorted(by_priority, key=lambda t: t[2])[::-1]` instead of `reverse=True`?" `['T1', 'T5', 'T4', 'T2', 'T3']`. Reversing the list afterwards also reverses the ties.
2. "A ticket arrives with priority `'urgent'`." The key function raises KeyError. Decide explicitly: fail loudly, or `rank.get(p, len(rank))` to sink unknowns to the end.

Scoring: Strong: all three lines exact and states that `reverse=True` keeps stability. Acceptable: lines 1 and 3 right, slips on tie order in line 2. Red flag: doesn't know Python's sort is stable.

E-O06 · Tracing a lazy generator pipeline

Easy · Output · All roles · Round 1 · generators, laziness · 4 min

Question: What does this print?

```
def read_orders(order_ids):
    for order_id in order_ids:
        print(f"read {order_id}")
        yield order_id

def enrich(orders):
    for order_id in orders:
        print(f"enrich {order_id}")
        yield {"id": order_id, "total": order_id * 10}

pipeline = enrich(read_orders([101, 102, 103]))
print("pipeline built")
first = next(pipeline)
print(first)
print(next(pipeline)["total"])
```

Expected output:

```
pipeline built
read 101
enrich 101
{'id': 101, 'total': 1010}
read 102
enrich 102
1020
```

Explanation: Calling a generator function runs none of its body; it returns a generator object paused before the first line. So building `pipeline` prints nothing and `pipeline built` comes first. `next(pipeline)` resumes `enrich`; its `for` asks `read_orders` for an item, which runs until its first `yield` (printing `read 101`) and hands 101 up; `enrich` prints and yields the dict. Each `next` pulls exactly one item through the whole chain: items flow one at a time, not stage by stage. Order 103 is never read. Takeaway: generators are pull-based, so memory per stage is constant, and side effects in an unconsumed generator never happen.

Follow-ups:

1. "What if `read_orders` built and returned a list instead of yielding?" Arguments are evaluated eagerly, so `read 101`, `read 102`, `read 103` all print before `pipeline built`. `enrich` is still lazy.
2. "A final stage does `map(write_to_db, rows)` and nobody iterates it. What gets written?" Nothing: `map` is lazy in Python 3, a classic silent no-op.

Scoring: Strong: exact interleaving and explains pull-based flow. Acceptable: right order after tracing slowly. Red flag: prints all three `read` lines before any `enrich`.

E-C01 · Batch records for a bulk API

Easy · Coding · All roles · Round 1 · iterators, generators, typing generics · 10–12 min

Problem: A bulk-ingest API accepts at most `size` records per request. Write `batched`, which takes any iterable (a list, a generator reading a 20 GB file, a DB cursor) and yields lists of at most `size` items in input order. It must never hold more than one batch in memory. An invalid `size` must fail at call time, not on the first iteration.

Signature: `def batched(items: Iterable[T], size: int) -> Iterator[list[T]]:`

Constraints: The input may be a one-shot iterator of unknown length (no `len()`, no slicing). `size < 1` is an error. Standard library only.

Examples:

- `batched(range(7), 3)` → `[0, 1, 2]`, `[3, 4, 5]`, `[6]`
- `batched([], 3)` → yields nothing
- `batched([1, 2], 0)` → `ValueError`, raised by the call itself

Hints:

1. "Can you call `len()` or slice a generator?"
2. "`itertools.islice(iterator, n)` takes the next n items from an iterator."
3. "If the `raise` lives inside a generator function, when does it actually execute?"

Reference solution:

```python
from collections.abc import Iterable, Iterator
from itertools import islice
from typing import TypeVar

T = TypeVar("T")


def batched(items: Iterable[T], size: int) -> Iterator[list[T]]:
    """Yield consecutive lists of at most `size` items from any iterable.
```

```
    Validates `size` eagerly (at call time), then streams: only one batch is
    held in memory at a time, so generators and file objects work too.
    """
    if not isinstance(size, int) or size < 1:
        raise ValueError(f"size must be a positive int, got {size!r}")
    return _batches(iter(items), size)


def _batches(iterator: Iterator[T], size: int) -> Iterator[list[T]]:
    while batch := list(islice(iterator, size)):
        yield batch
```

Complexity: O(n) time (each item touched once); O(size) extra space (one batch alive at a time).

Tests:

```
assert list(batched([], 3)) == []
assert list(batched(["o1"], 3)) == [["o1"]]
assert list(batched(range(7), 3)) == [[0, 1, 2], [3, 4, 5], [6]]
assert list(batched(range(6), 3)) == [[0, 1, 2], [3, 4, 5]]

pulled = []
def order_stream():
    for i in range(10):
        pulled.append(i)
        yield i
gen = batched(order_stream(), 4)
assert next(gen) == [0, 1, 2, 3]
assert pulled == [0, 1, 2, 3]          # lazy: nothing beyond the first batch was read

try:
    batched([1, 2], 0)                   # no next() needed: fails at call time
    raise AssertionError("expected ValueError")
except ValueError:
    pass
```

Alternatives and tradeoffs: `itertools.batched` (3.12+) does this in C but yields tuples; 3.13 adds `strict=True` to raise on a short final batch. A slice loop `items[i:i + size]` is simple but only works on sequences. A manual append-and-flush loop is fine, just more code. Pick: `itertools.batched` on 3.12+ when tuples are acceptable, else this `islice` version. The helper split exists only to make validation eager; as a single generator function the `ValueError` would surface on the first `next()`, possibly far from the bad call site.

Follow-ups:

- "Batches must also stay under 5 MB of serialised JSON." Keep a running byte count and flush before an item would overflow; decide what happens to a single oversize item (reject or dead-letter).
- "Some batches fail. How do you retry without re-reading the source?" Keep the failed batch in hand, retry with backoff, and send an idempotency key per batch so a retry after a timeout doesn't double-insert.
- "Make it async." An async generator consumed with `async for`, fed by an async source.

Scoring: Strong: streams from generators, validates eagerly, can explain why the split exists. Acceptable: slice-based version for lists that adapts when told the input is a generator. Red flag: calls `list(items)` up front on a "20 GB" input without noticing.

E-C02 · Top error codes from log lines

Easy · Coding · All roles · Round 1 · Counter, sorting, parsing · 10 min

Problem: Given application log lines of the form `<iso-timestamp> <LEVEL> <CODE> <free text>`, return the `k` most frequent error codes among `ERROR` lines as `(code, count)` pairs. Order by count descending, then code ascending, so the report is identical regardless of input order. Skip malformed lines silently.

Signature: `def top_error_codes(lines: list[str], k: int) -> list[tuple[str, int]]:`

Constraints: Up to 10^6 lines; distinct codes are few (at most a few thousand); `k >= 0`.

Examples:

- The test log below with `k=2` → `[("E401", 2), ("E502", 2)]` (tie broken alphabetically)
- `[]` with `k=3` → `[]`
- A single ERROR line with `k=1` → `[("E999", 1)]`

Hints:

1. "Which stdlib type counts hashable things?"
2. "How does `Counter.most_common` order items with equal counts?"
3. "What does `line.split(maxsplit=3)` give you for a malformed line?"

Reference solution:

```python
from collections import Counter


def top_error_codes(lines: list[str], k: int) -> list[tuple[str, int]]:
    """Return the k most frequent ERROR codes as (code, count) pairs.

    Line format: '<iso-timestamp> <LEVEL> <CODE> <free text>'. Only ERROR lines
    count; malformed lines are skipped. Order: count descending, then code
    ascending, so ties are deterministic.
    """
    if k <= 0:
        return []
    counts: Counter[str] = Counter()
    for line in lines:
        parts = line.split(maxsplit=3)
        if len(parts) >= 3 and parts[1] == "ERROR":
            counts[parts[2]] += 1
    return sorted(counts.items(), key=lambda item: (-item[1], item[0]))[:k]
```

Complexity: O(n + d log d) time for n lines and d distinct codes (one counting pass, then a sort of distinct codes); O(d) space.

Tests:

```python
logs = [
    "2026-09-01T10:00:00Z ERROR E502 upstream timeout",
    "2026-09-01T10:00:01Z INFO  OK200 health check",
    "2026-09-01T10:00:02Z ERROR E401 token expired",
    "2026-09-01T10:00:03Z ERROR E502 upstream timeout",
    "2026-09-01T10:00:04Z ERROR E401 token expired",
    "2026-09-01T10:00:05Z ERROR E500 null pointer in pricing",
    "garbage line",
    "",
]
assert top_error_codes(logs, 2) == [("E401", 2), ("E502", 2)]
assert top_error_codes(logs, 10) == [("E401", 2), ("E502", 2), ("E500", 1)]
assert top_error_codes([], 3) == []
assert top_error_codes(["2026-09-01T10:00:00Z ERROR E999 disk full"], 1) == [("E999", 1)]
assert top_error_codes(logs, 0) == []
# most_common breaks ties by first-seen order, not alphabetically
assert Counter(["E502", "E401", "E502", "E401"]).most_common(1) == [("E502", 2)]
```

Alternatives and tradeoffs: `counts.most_common(k)` is shorter, but ties follow first-seen order, so the same data in a different order produces a different report. `heapq.nsmallest(k, counts.items(), key=lambda i: (-i[1], i[0]))` is O(d log k), better when d is huge and k small. Pick: explicit sort for small d, `nsmallest` for large d.

Follow-ups:

- "The log is 50 GB." Iterate the file object directly (already line-lazy): same code, O(d) memory.
- "Logs come from 40 servers." Counters merge with `+` or `.update()`: natural map-reduce.

- "Top codes per hour." Key the counter by `(hour, code)`, or keep a dict of Counters.

Scoring: Strong: Counter, deterministic ties, skips malformed lines without crashing. Acceptable: correct dict-based count, needs a nudge on ties. Red flag: crashes on `"garbage line"` or re-scans the log once per code, O(n·d).

E-C03 · Deep-merge layered configs without mutation

Easy · Coding · All roles · Round 1 · dicts, copying, recursion · 12 min

Problem: A service loads `base` config, then an environment `override`. Merge rules: when both sides are dicts, merge recursively; any other override value replaces the base value, including lists and an explicit `None`. Neither input may be mutated, and the result must share no mutable objects with either input. Key order: base keys first, then override-only keys.

Signature: `def deep_merge(base: dict[str, Any], override: dict[str, Any]) -> dict[str, Any]:`

Constraints: Configs are small (hundreds of keys), nested fewer than 10 levels, JSON-like values.

Examples:

- `{"db": {"host": "localhost", "port": 5432}}` + `{"db": {"host": "db.internal"}}` → `{"db": {"host": "db.internal", "port": 5432}}`
- `{"retry": {"max": 3}}` + `{"retry": None}` → `{"retry": None}`
- `{}` + `{}` → `{}`

Hints:

1. "What does `{**base, **override}` do to a nested dict?"
2. "When both values are dicts, what smaller problem are you left with?"
3. "If you put `base["features"]` into the result as-is and the caller appends to it, what happens?"

Reference solution:

```python
import copy
from typing import Any


def deep_merge(base: dict[str, Any], override: dict[str, Any]) -> dict[str, Any]:
    """Return a new dict with `override` layered on top of `base`.

    Nested dicts merge recursively; any other value in `override` (including
    lists and None) replaces the base value. Neither input is mutated, and the
    result shares no mutable objects with either input. Key order: base keys
    first, then keys that only exist in override.
    """
    merged: dict[str, Any] = {}
    for key, base_value in base.items():
        if key not in override:
            merged[key] = copy.deepcopy(base_value)
        elif isinstance(base_value, dict) and isinstance(override[key], dict):
            merged[key] = deep_merge(base_value, override[key])
        else:
            merged[key] = copy.deepcopy(override[key])
    for key, value in override.items():
        if key not in base:
            merged[key] = copy.deepcopy(value)
    return merged
```

Complexity: O(total size of both inputs) time and space; each value is copied exactly once.

Tests:

```python
base = {"db": {"host": "localhost", "port": 5432, "pool": {"min": 1, "max": 5}},
        "features": ["search"], "debug": False}
prod = {"db": {"host": "db.internal", "pool": {"max": 20}}, "features": ["search", "recs"]}
snapshot = copy.deepcopy(base)
```

```
merged = deep_merge(base, prod)
assert merged == {"db": {"host": "db.internal", "port": 5432, "pool": {"min": 1, "max": 20}},
                  "features": ["search", "recs"], "debug": False}
assert base == snapshot                                    # base untouched
merged["db"]["pool"]["min"] = 99
merged["features"].append("ads")
assert base == snapshot and prod["features"] == ["search", "recs"]   # no shared objects
assert deep_merge({}, {}) == {}
assert deep_merge({"a": 1}, {}) == {"a": 1}
assert deep_merge({"retry": {"max": 3}}, {"retry": None}) == {"retry": None}      # explicit
null wins
assert deep_merge({"retry": 3}, {"retry": {"max": 5}}) == {"retry": {"max": 5}}   # type
conflict
assert list(deep_merge({"b": 1, "a": 2}, {"c": 3, "a": 4})) == ["b", "a", "c"]
```

Alternatives and tradeoffs: An in-place recursive `update` into `base` is faster but mutates shared defaults (exactly the ET-O05 bug). `collections.ChainMap(override, base)` gives layered lookups with zero copying, but only at the top level; nested dicts are not merged. Pick: the copy-merge. Configs are tiny; isolation is worth more than the copies.

Follow-ups:

- "How can an override delete a key?" A sentinel such as `DELETE = object()`, skipped when seen. Don't overload `None`, which already means "explicit null".
- "Should lists concatenate?" A product decision: make it an explicit strategy parameter rather than a silent default.
- "Very deep nesting?" The recursion limit (1000) is irrelevant for configs; use an explicit stack for untrusted input.

Scoring: Strong: recursive, never mutates, proves independence with a mutation test. Acceptable: recursive and correct but returns shared nested lists until asked. Red flag: `base.update(override)` and done.

Part 1c · EASY & TRICKY (ET)

Each looks trivial; the obvious answer is wrong because of exactly one mechanism. Every trap here corresponds to a bug that has shipped in real systems.

ET-O01 · Rounding invoice amounts

Easy & Tricky · Output · All roles · Round 1 · float precision, banker's rounding, Decimal · 4 min

Question: What does this print?

```
from decimal import Decimal, ROUND_HALF_UP

print(round(2.5), round(3.5))
print(round(0.125, 2), round(2.675, 2))
print(f"{0.125:.2f} {2.675:.2f}")
cents = Decimal("0.01")
print(Decimal("2.675").quantize(cents, rounding=ROUND_HALF_UP))
print(Decimal(2.675).quantize(cents, rounding=ROUND_HALF_UP))
```

Expected output:

```
2 4
0.12 2.67
0.12 2.67
2.68
2.67
```

Trap answer: `3 4`, `0.13 2.68`, `0.13 2.68`, `2.68`, `2.68`. Tempting because school rounding is half-up, and `2.675` looks exact on screen.

Explanation: Two different mechanisms. First, `round()` uses round-half-to-even on the exact value it receives: 2.5 → 2, 3.5 → 4, which avoids the upward bias half-up adds when many rounded values are summed. 0.125 is exactly representable in binary (1/8), so it is a true tie and goes to the even neighbour, 0.12. Second, 2.675 is *not* representable; the nearest double is 2.67499999999999982236..., below the midpoint, so it rounds down to 2.67. That is representation error, not a tie rule. `:.2f` uses the same correctly-rounded conversion, so it agrees. `Decimal("2.675")` is an exact decimal, so `ROUND_HALF_UP` gives 2.68; `Decimal(2.675)` converts the float exactly, carrying its error, so 2.67. Fix: money is `Decimal` built from strings (or integer paise/cents) with the rounding mode the tax or business rule specifies.

Real-world bug it maps to: Invoice totals differ by one paisa from the payment gateway's or tax authority's half-up calculation and reconciliation flags thousands of mismatches. Often `Decimal(float)` in the ingestion layer "looks right" while preserving the float error.

Follow-ups:

1. "Why default to half-even?" Half-up biases sums of many values ending in 5 upward; half-even is unbiased on average.
2. "Where does `Decimal(float)` sneak in?" `json.loads` gives floats unless `parse_float=Decimal`; float64 pandas columns; DB drivers returning NUMERIC as float.
3. "Is `sum([0.1] * 3) == 0.3`?" False (`0.30000000000000004`). `math.isclose` for float comparison, `math.fsum` for accurate float sums, Decimal for money.

Scoring: Strong: separates half-even from representation error; fixes with `Decimal("…")` plus explicit rounding. Acceptable: knows banker's rounding, misses why 2.675 → 2.67. Red flag: "add 0.0001 before rounding", or "floats are fine for money".

ET-O02 · Stripping file extensions

Easy & Tricky · Output · All roles · Round 1 · strings · 2 min

Question: What does this print?

```
files = ["orders.csv", "access.csv", "logs.csv.gz"]
print([f.rstrip(".csv") for f in files])
print([f.removesuffix(".csv") for f in files])
```

Expected output:

```
['order', 'acce', 'logs.csv.gz']
['orders', 'access', 'logs.csv.gz']
```

Trap answer: `['orders', 'access', 'logs.csv.gz']` on both lines. Tempting because `rstrip(".csv")` reads like "remove the .csv suffix".

Explanation: `str.rstrip(chars)` treats its argument as a *set of characters*, not a substring. It removes trailing characters while they belong to {`.`, `c`, `s`, `v`} and stops at the first that doesn't. `"orders.csv"` loses `v`, `s`, `c`, `.`, then the `s` of "orders" → `"order"`. `"access.csv"` loses the extension and then both `ss` → `"acce"`. `"logs.csv.gz"` ends in `z`, not in the set, so nothing is removed. `removesuffix` (3.9+) removes the exact substring once, only if present. The same set semantics apply to `strip`/`lstrip` (substring version: `removeprefix`). For paths, use `pathlib`.

Real-world bug it maps to: A loader derives table names from filenames with `rstrip(".csv")`. `access.csv` lands in table `acce` and `orders.csv` in `order`, a SQL reserved word, which is a memorable way to spend a Friday.

Follow-ups:

1. "What is `Path("logs.csv.gz").stem`?" `'logs.csv'`: `stem` removes only the last suffix; `.suffixes` gives `['.csv', '.gz']`.
2. "Same fix on Python 3.8?" `name[:-4] if name.endswith(".csv") else name`.

Scoring: Strong: exact output, explains character-set semantics, reaches for `removesuffix`/`pathlib`. Acceptable: suspects the trap, gets `order` but misses `acce`. Red flag: confident in the trap and can't explain it once shown.

ET-O03 · A refund check that eats the iterator

Easy & Tricky · Output · All roles · Round 1 · iterators, exhaustion, short-circuiting · 3 min

Question: What does this print?

```
amounts = map(float, ["120.0", "-15.5", "80.0", "42.5"])
has_refund = any(a < 0 for a in amounts)
print(has_refund, sum(amounts))
print(sum(amounts))
```

Expected output:

```
True 122.5
0
```

Trap answer: `True 227.0` then `227.0` (or `0.0`). Tempting because `amounts` looks like a list of four numbers.

Explanation: In Python 3 `map` returns a lazy, single-pass iterator. `any()` pulls items only until it sees a truthy result, then stops: it consumed `120.0` and `-15.5`. `sum(amounts)` resumes from where the iterator stopped: 80.0 + 42.5 = 122.5. The iterator is now exhausted; the second `sum` sees zero items and returns its start value, the int `0` (not `0.0`). Nothing raises at any point. Fix: `list()` it if you need multiple passes, or compute both results in one loop.

Real-world bug it maps to: A validation step (`any`, `all`, `next`, or `len(list(rows))` "just for logging") runs over a generator of rows before the load step; the loader then writes a partial or empty batch, with no error and just missing data.

Follow-ups:

1. "Fix it for a 10 GB stream you can't `list()`." One loop accumulating total and a `has_refund` flag.
2. "Why `0` rather than `0.0`?" `sum`'s `start` defaults to int 0, and an empty iterable returns `start` unchanged. Use `sum(x, 0.0)` or `math.fsum`.
3. "Would a list behave the same?" No: each `for` over a list creates a fresh iterator.

Scoring: Strong: `122.5` and `0` with the short-circuit explanation. Acceptable: knows iterators exhaust but says the first sum is 0. Red flag: `227.0` with no suspicion even after being told `map` is lazy.

ET-O04 · Deduplicating order keys in a set

Easy & Tricky · Output · All roles · Round 1 · `__eq__`/`__hash__` contract, hashing · 4 min

Question: What happens when this runs?

```
class OrderKey:
    def __init__(self, order_id, region):
        self.order_id = order_id
        self.region = region

    def __eq__(self, other):
        return (self.order_id, self.region) == (other.order_id, other.region)

a = OrderKey(101, "south")
b = OrderKey(101, "south")
print(a == b)
print(OrderKey.__hash__)
seen = {a, b}
print(len(seen))
```

Expected output:

```
True
None
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'OrderKey'
```

Trap answer: `True`, some function repr, then `1` (or `2`). Tempting because every object is hashable by default.

Explanation: Sets and dicts locate an item by hash first and confirm with `==`, which requires the invariant `a == b ⇒ hash(a) == hash(b)`. The default hash is identity-based and would violate that invariant as soon as `__eq__` compares values. So when a class body defines `__eq__` without `__hash__`, Python sets `__hash__ = None`, making instances unhashable, and building `{a, b}` raises TypeError. Fix: `def __hash__(self): return hash((self.order_id, self.region))`, valid only if those fields never change while stored, or `@dataclass(frozen=True)`, which generates both consistently. Separately, `__eq__` should return `NotImplemented` for foreign types instead of raising AttributeError.

Real-world bug it maps to: Someone adds `__eq__` to a model "for nicer test assertions", and a production path that dedups models in a set or uses them as dict keys starts raising TypeError. Worse, someone "fixes" it with `__hash__ = object.__hash__` and dedup silently stops deduplicating.

Follow-ups:

1. "With `__hash__ = object.__hash__`, what is `len(seen)`?" `2` in CPython: equal objects land in different buckets and `==` is never consulted, so they are equal but not deduplicated.
2. "You hash on fields, insert into a set, then mutate a field?" The object is stranded in its old bucket: `x in s` is False even though it's in there.
3. "Which dataclass settings produce a hash?" `frozen=True` with default `eq=True` generates one; `eq=True, frozen=False` sets `__hash__` to None; `unsafe_hash=True` forces generation.

Scoring: Strong: predicts the TypeError and states the invariant. Acceptable: knows `__hash__` is needed but predicts `2`. Red flag: predicts `1` and can't explain how sets use hashes.

ET-O05 · Per-job config from shared defaults

Easy & Tricky · Output · All roles · Round 1 · shallow copy, mutation vs rebinding, `**kwargs` · 4 min

Question: What does this print?

```
DEFAULTS = {"retry": {"max_attempts": 3, "backoff": 2.0}, "tags": ["etl"]}

def config_for(job_name, **retry_overrides):
    config = DEFAULTS.copy()
    config["retry"].update(retry_overrides)
    config["tags"] = config["tags"] + [job_name]
    return config

nightly = config_for("nightly", max_attempts=10)
hourly = config_for("hourly")
print(hourly["retry"]["max_attempts"], hourly["tags"])
print(DEFAULTS["retry"]["max_attempts"], DEFAULTS["tags"])
```

Expected output:

```
10 ['etl', 'hourly']
10 ['etl']
```

Trap answer: `3 ['etl', 'hourly']` and `3 ['etl']`. Tempting because `.copy()` looks like it isolates each job's config.

Explanation: `dict.copy()` is shallow: a new outer dict whose values are the *same objects*. `config["retry"]` is the very dict inside `DEFAULTS`, so the nightly job's `.update(max_attempts=10)` mutates the shared inner dict; `DEFAULTS` and every later config (including hourly) see 10. The tags line behaves differently: `config["tags"] + [job_name]` builds a new list and rebinds the key in the copy only; the shared list is never mutated. Two operations, two outcomes: mutating through a shared reference leaks, rebinding doesn't. Fix: `copy.deepcopy(DEFAULTS)`, or build the inner dict fresh with `{**DEFAULTS["retry"], **retry_overrides}`; better, make defaults immutable so the leak fails loudly.

Real-world bug it maps to: A job runner or multi-tenant API builds per-tenant config from module-level defaults; one tenant's override silently becomes everyone's default until the process restarts ("fine after deploy, wrong after the first request").

Follow-ups:

1. "If `hourly` were built first, would it be safe?" No. It holds a reference to the same inner dict, so after nightly runs, `hourly["retry"]["max_attempts"]` is 10 too.
2. "Does wrapping `DEFAULTS` in `types.MappingProxyType` fix it?" Only the top level becomes read-only; the inner dict is still mutable. Needs deep immutability (frozen dataclasses, tuples) or a deep copy.

Scoring: Strong: exact output and contrasts mutation with rebinding unprompted. Acceptable: catches the retry leak, gets the tags line wrong. Red flag: believes `.copy()` is deep.

ET-O06 · Type hints on a line-total function

Easy & Tricky · Output · All roles · Round 1 · typing, runtime behaviour · 3 min

Question: What does this print?

```
def line_total(quantity: int, unit_price: int) -> int:
    return quantity * unit_price

row = {"sku": "A-17", "quantity": "3", "unit_price": 2}
print(line_total(row["quantity"], row["unit_price"]))
print(line_total(2, 1.5))
print(line_total.__annotations__["return"])
```

Expected output:

```
33
3.0
<class 'int'>
```

Trap answer: A `TypeError` on the first call ("the hints say int"), or `6`. Tempting because the signature looks like a contract.

Explanation: Annotations are metadata stored in `__annotations__`; the interpreter never checks them on call. So `line_total("3", 2)` evaluates `"3" * 2`, which is sequence repetition → `"33"`. `line_total(2, 1.5)` returns float `3.0` despite `-> int`. Enforcement comes from static checkers (mypy, pyright) before runtime, or from validation libraries (Pydantic, beartype) at runtime boundaries. Fix: parse and validate where strings enter the system (CSV, query params, env vars), e.g. `int(row["quantity"])` or a Pydantic model, and trust types after that boundary. (3.14: annotations are evaluated lazily under PEP 649/749; the output here is unchanged.)

Real-world bug it maps to: CSV quantities arrive as strings; line totals become `"33"`. Some get string-concatenated into reports; others explode with a TypeError in a `sum()` three services downstream, far from the cause.

Follow-ups:

1. "Would mypy catch the first call?" Yes: mypy infers the dict's value type as `object` (pyright: `str | int`), neither of which is `int`. Rows from `csv.DictReader` are `dict[str, str]` and are flagged too. That's the case for a type checker in CI.
2. "Does `isinstance(x, SomeProtocol)` with `@runtime_checkable` verify method signatures?" No: it only checks that the attributes/methods exist.

Scoring: Strong: `33/3.0`, explains that annotations are metadata, names static and boundary validation. Acceptable: knows hints aren't enforced but predicts `6`. Red flag: insists Python raises TypeError on hint mismatch.

ET-C01 · Split an amount into exact installments

Easy & Tricky · Coding · All roles (DE, BE, FS emphasis) · Round 1 · Decimal, integer arithmetic, divmod · 12 min

Problem: A checkout offers "pay in N installments". Split a non-negative total into `parts` installments that sum exactly to the total; leftover cents go one each to the earliest installments. Return strings with two decimals. Raise `ValueError` for more than two decimal places, negative or non-finite values, and non-numeric strings.

Signature: `def split_amount(total: str, parts: int) -> list[str]:`

Constraints: `0 <= total <= 10^12` with at most 2 decimal places; `1 <= parts <= 10^4`.

Examples:

- `("100.00", 3)` → `["33.34", "33.33", "33.33"]`
- `("0.02", 3)` → `["0.01", "0.01", "0.00"]`
- `("1.005", 2)` → `ValueError`

Trap answer: `[f"{float(total) / parts:.2f}"] * parts`. For 100 over 3 that is three × 33.33 = 99.99: a cent vanishes.

Real-world bug it maps to: Sum of installments ≠ order total, so the payment gateway rejects the mandate or the ledger never balances.

Hints:

1. "What is 3 × 33.33?"
2. "Can you do all the arithmetic in integers?"
3. "`divmod` gives quotient and leftover in one step."

Reference solution:

```python
from decimal import Decimal, InvalidOperation


def split_amount(total: str, parts: int) -> list[str]:
```

```
    """Split a non-negative currency amount into `parts` installments.

    The installments sum exactly to `total`. All arithmetic is done in integer
    cents; the leftover cents go one each to the earliest installments.
    `total` is a decimal string with at most two decimal places.
    """
    if parts < 1:
        raise ValueError("parts must be >= 1")
    try:
        amount = Decimal(total)
    except InvalidOperation:
        raise ValueError(f"not a decimal amount: {total!r}") from None
    if not amount.is_finite() or amount < 0:
        raise ValueError(f"amount must be finite and non-negative: {total!r}")
    if amount.as_tuple().exponent < -2:
        raise ValueError(f"more than 2 decimal places: {total!r}")
    cents = int(amount * 100)
    base, remainder = divmod(cents, parts)
    shares = [base + (1 if i < remainder else 0) for i in range(parts)]
    return [f"{Decimal(share) / 100:.2f}" for share in shares]
```

Complexity: O(parts) time and space: one integer division, one pass to build shares.

Tests:

```
assert split_amount("100.00", 3) == ["33.34", "33.33", "33.33"]
assert split_amount("10", 1) == ["10.00"]
assert split_amount("0.00", 2) == ["0.00", "0.00"]
assert split_amount("0.02", 3) == ["0.01", "0.01", "0.00"]
assert sum(Decimal(x) for x in split_amount("999.99", 7)) == Decimal("999.99")
assert split_amount("1E+2", 4) == ["25.00"] * 4
for bad in ["1.005", "-5.00", "abc", "NaN"]:
    try:
        split_amount(bad, 2)
        raise AssertionError(bad)
    except ValueError:
        pass
```

Alternatives and tradeoffs: Quantize each Decimal share and push the rounding difference into the last installment: exact, but produces a visibly odd last payment and is easy to get subtly wrong. For weighted splits (70/30), the largest-remainder method. Pick: integer cents with `divmod`; exact by construction and trivially auditable.

Follow-ups:

- "Split by weights instead of equally." Largest remainder: floor each weighted share in cents, hand leftover cents to the largest fractional remainders with a deterministic tie-break.
- "Refunds (negative totals)?" `divmod(-10000, 3)` is `(-3334, 2)` because Python floors toward negative infinity, which flips the distribution. Split the absolute value and reapply the sign.
- "JPY has 0 minor units, KWD has 3." Parameterise the exponent per currency (ISO 4217) instead of hardcoding 100.

Scoring: Strong: integer cents with `divmod`, validation, mentions negative-`divmod` behaviour when asked about refunds. Acceptable: a correct Decimal "last installment absorbs the difference" version. Red flag: float arithmetic, and never checks that parts sum to the total.

ET-C02 · Deduplicate user-entered tags

Easy & Tricky · Coding · All roles · Round 1 · Unicode normalisation, casefold, sets · 12 min

Problem: Users attach free-text tags to articles. Deduplicate so that tags differing only in case, surrounding or repeated whitespace, or Unicode representation count as the same tag. Keep the first-seen display form (whitespace collapsed, stored composed/NFC), preserve first-seen order, drop blanks.

Signature: `def normalise_tags(tags: list[str]) -> list[str]:`

Constraints: Up to 10^5 tags, each at most 100 characters, any script.

Examples:

- `["Python", " python ", "Café", "Cafe\u0301", "STRASSE", "straße", "Data Engineering", "data engineering", " "]` → `["Python", "Café", "STRASSE", "Data Engineering"]`
- `[]` → `[]`
- `["Ｐｙｔｈｏｎ", "python"]` → `["Ｐｙｔｈｏｎ"]` (full-width characters)

Trap answer: `list(dict.fromkeys(t.strip().lower() for t in tags if t.strip()))` returns 7 tags: `"Café"` and `"Cafe\u0301"` render identically but differ, `"straße".lower()` stays `"straße"`, inner spaces aren't collapsed, and the display form is lost.

Real-world bug it maps to: Duplicate tags or usernames that look identical in the UI; uniqueness constraints bypassed by a different code-point sequence. Text copied from macOS filenames and some mobile keyboards often arrives decomposed.

Hints:

1. "Are `"Café"` and `"Cafe\u0301"` equal? What's `len()` of each?"
2. "Look at `unicodedata.normalize`."
3. "Compare `"straße".lower()` with `"straße".casefold()`."

Reference solution:

```python
import unicodedata


def normalise_tags(tags: list[str]) -> list[str]:
    """Deduplicate user-entered tags, keeping the first-seen display form.

    Two tags match if their keys are equal: whitespace collapsed, NFKC
    normalisation, casefolding, then NFKC again (casefold can emit sequences
    that need renormalising). The display form is the first occurrence with
    whitespace collapsed and NFC applied. Blank tags are dropped.
    """
    seen: set[str] = set()
    result: list[str] = []
    for tag in tags:
        display = unicodedata.normalize("NFC", " ".join(tag.split()))
        key = unicodedata.normalize("NFKC", unicodedata.normalize("NFKC", display).casefold())
        if key and key not in seen:
            seen.add(key)
            result.append(display)
    return result
```

Complexity: O(total characters) time; O(unique tags) space.

Tests:

```python
raw = ["Python", " python ", "Café", "Cafe\u0301", "STRASSE", "straße",
       "Data  Engineering", "data engineering", "   "]
assert normalise_tags(raw) == ["Python", "Café", "STRASSE", "Data Engineering"]
assert normalise_tags([]) == []
assert normalise_tags(["  "]) == []
assert normalise_tags(["Kafka"]) == ["Kafka"]
out = normalise_tags(["Cafe\u0301"])
assert out == ["Café"] and len(out[0]) == 4                    # stored composed (NFC)
assert normalise_tags(["Ｐｙｔｈｏｎ", "python"]) == ["Ｐｙｔｈｏｎ"]  # full-width folds via NFKC
naive = list(dict.fromkeys(t.strip().lower() for t in raw if t.strip()))
assert len(naive) == 7
```

Alternatives and tradeoffs: NFC-only keys are less aggressive: full-width "Ｐｙｔｈｏｎ" and ligatures like "fi" stay distinct, right for some content, wrong for identifiers. The display form stays NFC because NFKC is lossy for presentation. Pick: NFKC + casefold for the key, NFC for display (Python itself uses NFKC for identifiers).

Follow-ups:

- "Enforce uniqueness in Postgres." Store the computed key in its own column with a unique index; application-side checks alone race under concurrent inserts.
- "Cyrillic 'a' vs Latin 'a' (homoglyphs)?" Normalisation doesn't touch these; you need confusable skeletons (Unicode TR39). Matters for usernames and anti-phishing.
- "Turkish dotted/dotless i?" `casefold` is locale-independent; locale-correct Turkish needs ICU or explicit rules.

Scoring: Strong: names normalisation forms and casefold unprompted; keeps order and display form. Acceptable: case and whitespace right, adds normalisation after hint 1. Red flag: `set(t.lower() for t in tags)`, losing order, unaware of representation differences.

ET-C03 · Leaderboard with tie-breaks

Easy & Tricky · Coding · All roles · Round 1 · multi-key sort, `reverse`, stability · 8 min

Problem: A hackathon leaderboard shows the top `n` submissions: score descending; on equal score the earlier submission wins; if both tie, keep input order. Return submission ids.

Signature: `def top_submissions(submissions: list[tuple[str, int, int]], n: int) -> list[str]:` where each tuple is `(submission_id, score, submitted_at_epoch)`.

Constraints: Up to 10^6 submissions; ints for score and timestamp; if `n` exceeds the count return all; `n <= 0` returns `[]`.

Examples:

- `[("asha", 90, 1005), ("ravi", 95, 1010), ("meera", 90, 1001), ("kiran", 95, 1020)]`, `n=3` → `["ravi", "kiran", "meera"]`
- `[]`, `n=3` → `[]`
- Full tie `[("x", 50, 7), ("y", 50, 7), ("z", 50, 7)]`, `n=2` → `["x", "y"]`

Trap answer: `sorted(subs, key=lambda s: (s[1], s[2]), reverse=True)`. `reverse` applies to the whole tuple, so ties go to the *later* submission: `["kiran", "ravi", "asha"]`.

Real-world bug it maps to: "First to hit the threshold wins" payouts or SLA rankings reward the latest entry; the same inverted-tuple mistake in "keep the latest record" dedup keeps the oldest.

Hints:

1. "What does `reverse=True` do to the second element of your key?"
2. "How do you make a number sort descending inside an ascending sort?"
3. "What if the tie-break were a string, which you can't negate?"

Reference solution:

```python
def top_submissions(submissions: list[tuple[str, int, int]], n: int) -> list[str]:
    """Return the ids of the top-n submissions.

    Each submission is (submission_id, score, submitted_at_epoch). Order:
    higher score first; on equal score, earlier submission first; if both
    tie, original input order (sorted() is stable).
    """
    ranked = sorted(submissions, key=lambda s: (-s[1], s[2]))
    return [s[0] for s in ranked[:max(n, 0)]]
```

Complexity: O(N log N) time; O(N) space for the sorted copy.

Tests:

```python
subs = [("asha", 90, 1005), ("ravi", 95, 1010), ("meera", 90, 1001), ("kiran", 95, 1020)]
assert top_submissions(subs, 3) == ["ravi", "kiran", "meera"]
assert top_submissions([], 3) == []
assert top_submissions([("solo", 10, 1)], 5) == ["solo"]
assert top_submissions([("x", 50, 7), ("y", 50, 7), ("z", 50, 7)], 2) == ["x", "y"]
assert top_submissions(subs, 0) == []
```

```python
naive = [s[0] for s in sorted(subs, key=lambda s: (s[1], s[2]), reverse=True)[:3]]
assert naive == ["kiran", "ravi", "asha"]
two_pass = sorted(sorted(subs, key=lambda s: s[2]), key=lambda s: s[1], reverse=True)
assert [s[0] for s in two_pass[:3]] == ["ravi", "kiran", "meera"]
```

Alternatives and tradeoffs: Two stable passes (secondary key first, then primary with `reverse=True`) work for any key type including strings, at the cost of two sorts. `heapq.nsmallest(n, subs, key=lambda s: (-s[1], s[2]))` is O(N log n), better when n ≪ N. Pick: negated tuple key for numeric keys, two passes for non-numeric secondary keys, `nsmallest` for a top-10 out of 10^8.

Follow-ups:

- "Show each participant once, with their best submission." Reduce per participant first (max by `(score, -ts)`), then rank.
- "10^8 submissions arriving as a stream." A bounded heap of size n; O(n) memory.
- "Scores are floats and some are NaN." NaN compares False against everything, breaking the total ordering the sort assumes; results become garbage. Filter or map NaN explicitly first.

Scoring: Strong: negated tuple key or two passes, explains why `reverse=True` breaks the tie-break. Acceptable: writes the trap, fixes it when shown the example. Red flag: hand-written $O(N^2)$ sort, or doesn't see why tie order matters.

Part 2a • MEDIUM (M)

The core bar for a 3–4 year engineer. Each question combines two concepts or needs careful tracing. Output questions: 4–8 minutes. Coding tasks: 15–20 minutes.

M-O01 · Stacked decorators on an audited handler

Medium · Output · All roles · Round 2 · decorators with arguments, `functools.wraps`, stacking order · 6 min

Question: What does this print?

```python
import functools

def audit(action):
    def decorator(func):
        print(f"decorating {func.__name__} with audit({action})")
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            print(f"audit:{action} before")
            result = func(*args, **kwargs)
            print(f"audit:{action} after")
            return result
        return wrapper
    return decorator

def require_role(role):
    def decorator(func):
        print(f"decorating {func.__name__} with require_role({role})")
        def wrapper(*args, **kwargs):
            print(f"check role {role}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@audit("refund")
@require_role("admin")
def issue_refund(order_id):
    print(f"refunding {order_id}")
    return "ok"

print("---")
print(issue_refund(7))
print(issue_refund.__name__)
```

Expected output:

```
decorating issue_refund with require_role(admin)
decorating wrapper with audit(refund)
---
audit:refund before
check role admin
refunding 7
audit:refund after
ok
wrapper
```

Explanation: `@audit("refund")` is two steps: call the factory `audit("refund")` to get a decorator, then apply that decorator to whatever is below it. Stacked decorators apply bottom-up: `issue_refund = audit("refund")(require_role("admin")(issue_refund))`. So `require_role`'s decorator runs first and prints the real name. `audit`'s decorator then receives `require_role`'s inner `wrapper`, and since `require_role` didn't use `functools.wraps`, that object's `__name__` is literally `"wrapper"`. At call time the outermost wrapper runs first: audit before → role check → function → audit after. The final `__name__` is `"wrapper"`: `audit` *did* use `wraps`, but it faithfully

copied metadata from the already-anonymous inner wrapper. Takeaway: every decorator in a stack must use `wraps`; one missing link loses the name, docstring, and `__wrapped__` for everything above it.

Follow-ups:

1. "Why does it matter that `__name__` is wrong?" Logging and metrics keyed by function name collapse into "wrapper"; FastAPI/Flask route names and `inspect.signature` break; pytest parametrisation and Celery task registration can collide.
2. "What does `wraps` copy?" `__module__`, `__name__`, `__qualname__`, `__doc__`, `__dict__` (updated), `__type_params__` (3.12+), and sets `__wrapped__`, which `inspect.signature` follows.
3. "Swap the decorator order: when is the role checked relative to the audit log?" Role check becomes the outer layer, so denied calls never reach audit. Order is policy, not style.

Scoring: Strong: exact output including `decorating wrapper` and final `wrapper`. Acceptable: gets call order right, misses the name leak. Red flag: applies decorators top-down or can't explain the factory layer.

M-O02 · A contextmanager transaction that suppresses errors

Medium · Output · All roles · Round 2 · context managers, `contextlib`, exception suppression · 7 min

Question: What does this print?

```python
from contextlib import contextmanager

class PaymentDeclined(Exception):
    pass

@contextmanager
def transaction(name):
    print(f"begin {name}")
    try:
        yield name.upper()
        print(f"commit {name}")
    except PaymentDeclined as exc:
        print(f"rollback {name}: {exc}")
    finally:
        print(f"release {name}")

with transaction("order-1") as tx:
    print(f"inside {tx}")

with transaction("order-2"):
    raise PaymentDeclined("card expired")
print("after order-2")

with transaction("order-3"):
    raise KeyError("sku")
print("after order-3")
```

Expected output:

```
begin order-1
inside ORDER-1
commit order-1
release order-1
begin order-2
rollback order-2: card expired
release order-2
after order-2
begin order-3
release order-3
Traceback (most recent call last):
  ...
KeyError: 'sku'
```

Explanation: `@contextmanager` turns a generator into an object with `__enter__`/`__exit__`. `__enter__` runs the generator to its `yield`; the yielded value is what `as` binds. On a clean exit, `__exit__` resumes the generator after `yield` (commit, then `finally`). On an exception, `__exit__` *throws* it into the generator at the `yield`. For order-2 the `except` handles it and the generator finishes normally, which `contextmanager` reports as `__exit__` returning True: the exception is **suppressed**, and `after order-2` prints. For order-3 the KeyError isn't caught; `finally` runs and the exception propagates out of the `with`. Takeaway: catching an exception inside a `@contextmanager` generator without re-raising silently swallows it; add `raise` after the rollback.

Follow-ups:

1. "Write the same with a class." `__exit__(self, exc_type, exc, tb)` returns True to suppress, False/None to propagate.
2. "What if the generator yields twice?" `RuntimeError: generator didn't stop` on exit.
3. "Name a stdlib context manager that suppresses on purpose." `contextlib.suppress(FileNotFoundError)`; also `ExitStack` for a dynamic number of managers.

Scoring: Strong: sees that order-2 is suppressed and explains the throw-into-generator mechanism. Acceptable: gets order-1/order-3 right, thinks order-2 re-raises. Red flag: can't describe `__enter__`/`__exit__`.

M-O03 · Event publisher signature: *args, keyword-only, unpacking

Medium · Output · All roles · Round 2 · functions, `*args`/`**kwargs`, keyword-only args, unpacking · 5 min

Question: What does this print?

```python
def publish(topic, *payloads, retries=3, **headers):
    print(topic, payloads, retries, headers)

publish("orders", {"id": 1}, {"id": 2}, trace="abc")
batch = [{"id": 3}, {"id": 4}]
meta = {"retries": 5, "tenant": "acme"}
publish("orders", *batch, **meta)
publish(*["invoices", {"id": 9}], retries=0)
publish("orders", {"id": 5}, 5)
try:
    publish("orders", topic="dup")
except TypeError as exc:
    print("TypeError:", exc)
```

Expected output:

```
orders ({'id': 1}, {'id': 2}) 3 {'trace': 'abc'}
orders ({'id': 3}, {'id': 4}) 5 {'tenant': 'acme'}
invoices ({'id': 9},) 0 {}
orders ({'id': 5}, 5) 3 {}
TypeError: publish() got multiple values for argument 'topic'
```

Explanation: Binding goes left to right: the first positional fills `topic`, all remaining positionals are packed into the tuple `payloads`. Anything after `*payloads` in the signature (`retries`) is keyword-only: it can never be filled positionally, which is why the bare `5` in call 4 lands in `payloads` and `retries` stays 3. `**headers` collects unmatched keyword arguments into a dict. At the call site, `*batch` spreads a list into positionals and `**meta` spreads a dict into keywords; `retries` matches its parameter and `tenant` falls into `headers`. The last call binds `topic` twice (positionally and by keyword), which is a TypeError. Takeaway: keyword-only parameters prevent "magic positional number" bugs; that's exactly why the signature is designed this way.

Follow-ups:

1. "How do you make `topic` positional-only?" `def publish(topic, /, *payloads, ...)` (3.8+). Then `**headers` can even accept a `topic` key without colliding.
2. "`payloads` is a tuple, not a list. Why does that matter?" It's immutable and a fresh object per call; appending requires converting.

3. "What if `meta` had a key `'topic'`?" Same "multiple values" TypeError, since `topic` was also given positionally.

Scoring: Strong: all five lines, including `5` landing in `payloads`. Acceptable: slips on one line, fixes it by re-reading the signature. Red flag: thinks `publish("orders", {"id": 5}, 5)` sets `retries=5`.

M-O04 · Exception chaining in a payment client

Medium · Output · All roles · Round 2 · exception chaining, custom exceptions · 6 min

Question: What does this print?

```
class GatewayError(Exception):
    pass

def charge(card):
    try:
        return {"visa": 100}[card]
    except KeyError as exc:
        raise GatewayError(f"unsupported card {card}") from exc

def charge_quietly(card):
    try:
        return {"visa": 100}[card]
    except KeyError:
        raise GatewayError(f"unsupported card {card}") from None

def charge_sloppy(card):
    try:
        return {"visa": 100}[card]
    except KeyError:
        raise GatewayError(f"unsupported card {card}")

for fn in (charge, charge_quietly, charge_sloppy):
    try:
        fn("amex")
    except GatewayError as err:
        print(fn.__name__, repr(err.__cause__), repr(err.__context__),
err.__suppress_context__)
```

Expected output:

```
charge KeyError('amex') KeyError('amex') True
charge_quietly None KeyError('amex') True
charge_sloppy None KeyError('amex') False
```

Explanation: Raising inside an `except` block always records the exception being handled as `__context__`: that's implicit chaining, and it happens in all three functions. `raise X from Y` additionally sets `__cause__ = Y` and `__suppress_context__ = True` (explicit chaining). `from None` sets `__cause__ = None` and `__suppress_context__ = True`, but `__context__` is still recorded; it's only hidden from the printed traceback. The traceback printer uses these: `from exc` prints "The above exception was the direct cause of the following exception", the sloppy version prints "During handling of the above exception, another exception occurred" (which reads like a bug in your handler), and `from None` shows only GatewayError. Takeaway: use `from exc` when translating errors across a boundary, `from None` when the inner error is noise, never the sloppy form.

Follow-ups:

1. "Which one would you use at a public API boundary?" `from None` if the inner detail leaks internals, but log the original first; otherwise `from exc`.
2. "How would Sentry or a logger see these?" They walk `__cause__`/`__context__`; `from exc` keeps the full chain diagnosable.
3. "Should `GatewayError` subclass `Exception` or `BaseException`?" `Exception`. `BaseException` is for control flow like `KeyboardInterrupt`, `SystemExit`, and `CancelledError`, which generic handlers must not catch.

Scoring: Strong: all three rows including `__context__` still set under `from None`. Acceptable: gets `__cause__` right, thinks `from None` clears `__context__`. Red flag: unaware of `raise ... from`.

M-O05 · Dataclass defaults, equality and ordering

Medium · Output · All roles · Round 2 · dataclasses, generated dunder methods · 6 min

Question: What does this print?

```python
from dataclasses import dataclass, field

@dataclass(order=True)
class Shipment:
    priority: int
    shipment_id: str = field(compare=False)
    items: list[str] = field(default_factory=list, compare=False, repr=False)
    weight_kg: float = 0.0

a = Shipment(2, "S-1", weight_kg=5.0)
b = Shipment(2, "S-2", ["tv"], 5.0)
c = Shipment(1, "S-3")
a.items.append("fan")
print(a)
print(a == b, a is b)
print([s.shipment_id for s in sorted([a, b, c])])
print(c.items, Shipment.__hash__)
```

Expected output:

```
Shipment(priority=2, shipment_id='S-1', weight_kg=5.0)
True False
['S-3', 'S-1', 'S-2']
[] None
```

Explanation: `@dataclass` generates `__init__`, `__repr__`, and `__eq__` from the annotated fields; `order=True` adds `__lt__` and friends. `field(repr=False)` hides `items` from the repr. `compare=False` removes `shipment_id` and `items` from both `__eq__` and ordering, so the comparison tuple is `(priority, weight_kg)`: a and b are both `(2, 5.0)`, hence equal though distinct objects. Sorting compares those tuples; c `(1, 0.0)` comes first, and the a/b tie keeps input order because sort is stable. `default_factory=list` builds a fresh list per instance, so appending to `a.items` doesn't touch `c.items`. A plain `items: list = []` is rejected with ValueError at class creation. With `eq=True` and `frozen=False`, dataclasses set `__hash__ = None`, the same rule as ET-O04. Takeaway: know what each flag generates; "equal" here means "equal on compare fields".

Follow-ups:

1. "Two shipments with different IDs are `==`. Is that a bug?" Depends on intent; usually yes for entities. Identity fields should compare, or define `__eq__` on the ID only.
2. "How do you make it hashable?" `frozen=True` (then mutation raises `FrozenInstanceError`) or `unsafe_hash=True` if you accept the risk.
3. "`slots=True` (3.10+)?" Generates `__slots__`: less memory per instance, no `__dict__`, catches attribute typos (H-O04).

Scoring: Strong: all four lines; explains `compare=False` and why `__hash__` is None. Acceptable: misses the hash line or the repr omission. Red flag: expects `a == b` to be False "because the IDs differ" and can't read the field options.

M-O06 · mock.patch where the name is looked up

Medium · Output · All roles (BE/FS must-ask) · Round 2 · testing, `unittest.mock`, imports and module namespaces · 7 min

Question: What does this print? (The first block just builds two small modules in memory, as if `pricing.py` and `invoices.py` existed on disk.)

```python
import sys
import types
from unittest import mock

pricing = types.ModuleType("pricing")
exec("def get_rate(currency):\n    return {'USD': 83.0}[currency]\n", pricing.__dict__)
sys.modules["pricing"] = pricing

invoices = types.ModuleType("invoices")
exec(
    "from pricing import get_rate\n"
    "import pricing\n"
    "def total_direct(amount):\n    return amount * get_rate('USD')\n"
    "def total_via_module(amount):\n    return amount * pricing.get_rate('USD')\n",
    invoices.__dict__,
)
sys.modules["invoices"] = invoices

with mock.patch("pricing.get_rate", return_value=1.0):
    print(invoices.total_direct(10), invoices.total_via_module(10))

with mock.patch("invoices.get_rate", return_value=2.0):
    print(invoices.total_direct(10), invoices.total_via_module(10))

print(invoices.total_direct(10))
```

Expected output:

```
830.0 10.0
20.0 830.0
830.0
```

Explanation: `from pricing import get_rate` executes once, at import, and binds a *second name*, `invoices.get_rate`, to the same function object. `mock.patch("pricing.get_rate")` replaces the attribute on the `pricing` module only. `total_direct` looks up `get_rate` in `invoices`' globals, which still points at the original, so it's unaffected (830.0); `total_via_module` looks up `pricing.get_rate` at call time and sees the mock (10.0). Patching `"invoices.get_rate"` flips it. When the `with` exits, the patch restores the original binding. Rule: patch the name where it is *looked up*, not where it is defined.

Follow-ups:

1. "Which import style is easier to test?" `import pricing` + `pricing.get_rate(...)`: one patch point. Or better, inject the dependency (pass a rate provider) and avoid patching entirely.
2. "What does `autospec=True` add?" The mock enforces the real signature, so calling with wrong arguments fails the test instead of silently passing.
3. "Pytest equivalent?" `monkeypatch.setattr(invoices, "get_rate", fake)` inside a test, with automatic undo; fixtures give setup/teardown with scopes (function/module/session).

Scoring: Strong: exact output and states the rule unprompted. Acceptable: gets the second line only after a hint about `from ... import`. Red flag: believes patching the definition site patches every importer.

M-C01 · Retry decorator with exponential backoff

Medium · Coding · All roles · Round 2 · decorators, exceptions, testability · 20 min

Problem: Calls to a flaky upstream (payments API, S3, a feature store) sometimes fail transiently. Write a decorator factory `retry` that re-invokes the function when it raises one of the configured exception types, sleeping `min(base_delay * 2**(k-1), max_delay)` before retry k. After the final attempt, re-raise the *last* exception unchanged. Exceptions not listed must propagate immediately. The sleep function must be injectable so tests run instantly, and the wrapped function must keep its name and docstring.

Signature: `def retry(attempts: int = 3, exceptions: tuple[type[BaseException], ...] = (Exception,),`

`base_delay: float = 0.5, max_delay: float = 30.0, sleep: Callable[[float], None] = time.sleep) -> Callable[[Callable[P, R]], Callable[P, R]]:`

Constraints: `attempts >= 1` (validate at decoration time). No external libraries.

Examples:

- Fails twice with `ConnectionError`, then returns `"ok"`, with `attempts=3, base_delay=1` → returns `"ok"`; sleeps `[1, 2]`.
- Raises `ValueError` when only `ConnectionError` is retryable → propagates on the first call; no sleeps.
- Always fails with `attempts=3` → raises the third exception; sleeps `[1, 2]`.

Hints:

1. "This is three layers deep: factory → decorator → wrapper. Which layer holds the loop?"
2. "`except exceptions:` accepts a tuple of types."
3. "How will a test check the delays without actually waiting?"

Reference solution:

```python
import functools
import time
from collections.abc import Callable
from typing import ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")


def retry(
    attempts: int = 3,
    exceptions: tuple[type[BaseException], ...] = (Exception,),
    base_delay: float = 0.5,
    max_delay: float = 30.0,
    sleep: Callable[[float], None] = time.sleep,
) -> Callable[[Callable[P, R]], Callable[P, R]]:
    """Retry the decorated function on `exceptions` with exponential backoff.

    Delay before retry k (1-based) is min(base_delay * 2**(k-1), max_delay).
    After the final failed attempt the last exception is re-raised unchanged.
    Exceptions not in `exceptions` propagate immediately.
    """
    if attempts < 1:
        raise ValueError("attempts must be >= 1")

    def decorator(func: Callable[P, R]) -> Callable[P, R]:
        @functools.wraps(func)
        def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
            for attempt in range(1, attempts + 1):
                try:
                    return func(*args, **kwargs)
                except exceptions:
                    if attempt == attempts:
                        raise
                    sleep(min(base_delay * 2 ** (attempt - 1), max_delay))
            raise AssertionError("unreachable")

        return wrapper

    return decorator
```

Complexity: O(attempts) calls and sleeps; O(1) extra space.

Tests:

```python
slept: list[float] = []
calls = {"n": 0}
```

```python
@retry(attempts=3, exceptions=(ConnectionError,), base_delay=1, sleep=slept.append)
def fetch_rate(currency: str) -> str:
    """Fetch an FX rate."""
    calls["n"] += 1
    if calls["n"] < 3:
        raise ConnectionError("reset by peer")
    return f"{currency}:ok"

assert fetch_rate("USD") == "USD:ok" and slept == [1, 2] and calls["n"] == 3
assert fetch_rate.__name__ == "fetch_rate" and fetch_rate.__doc__ == "Fetch an FX rate."

slept.clear()
@retry(attempts=3, exceptions=(ConnectionError,), sleep=slept.append)
def bad_input() -> None:
    raise ValueError("not retryable")
try:
    bad_input()
    raise AssertionError("expected ValueError")
except ValueError:
    assert slept == []

errors = iter([TimeoutError("t1"), TimeoutError("t2"), TimeoutError("t3")])
@retry(attempts=3, exceptions=(TimeoutError,), base_delay=1, sleep=slept.append)
def always_fails() -> None:
    raise next(errors)
try:
    always_fails()
    raise AssertionError("expected TimeoutError")
except TimeoutError as exc:
    assert str(exc) == "t3" and slept == [1, 2]

slept.clear()
@retry(attempts=6, base_delay=10, max_delay=25, sleep=slept.append)
def capped() -> None:
    raise OSError
try:
    capped()
except OSError:
    assert slept == [10, 20, 25, 25, 25]

@retry(attempts=1, sleep=slept.append)
def once() -> int:
    return 1
assert once() == 1

try:
    retry(attempts=0)
    raise AssertionError("expected ValueError")
except ValueError:
    pass
```

Alternatives and tradeoffs: `tenacity` is the production answer (jitter, stop conditions, async support, logging hooks). A retry loop inlined at each call site is simpler but duplicates policy. Pick: `tenacity` in production; this exercise checks you understand what it does. Add full jitter (`random.uniform(0, delay)`) in real systems so a fleet of clients doesn't retry in lockstep.

Follow-ups:

- "Which exceptions should *not* be retried?" 4xx client errors (400/401/403/404/422), validation errors, and anything non-idempotent unless an idempotency key is sent. Retrying a non-idempotent POST can double-charge.
- "Support `async def` functions." Detect with `inspect.iscoroutinefunction` and use `await asyncio.sleep`; never `time.sleep` inside a coroutine (H-O02).
- "A total time budget, not just attempts?" Track a deadline with `time.monotonic()` and stop when the next sleep would exceed it.

Scoring: Strong: three layers, `wraps`, re-raises the last exception with bare `raise`, injectable sleep, raises the non-retryable immediately. Acceptable: works but hardcodes `time.sleep` or loses metadata. Red flag: catches everything (`except:` / `BaseException`), or wraps the final error in a new exception that loses the traceback.

M-C02 · Inventory reconciliation with Counter

Medium · Coding · All roles · Round 2 · `collections.Counter`, multiset arithmetic · 15 min

Problem: A warehouse compares a shipment manifest (expected SKUs, with repeats) against scanner output (received SKUs). Scanner output is messy: lowercase and stray whitespace. Return what's missing, what's unexpected (extra or unknown), and how many units matched, with dicts sorted by SKU for stable reports.

Signature: `def reconcile(expected: list[str], received: list[str]) -> Reconciliation:` where `Reconciliation` is a frozen dataclass with `missing: dict[str, int]`, `unexpected: dict[str, int]`, `matched: int`.

Constraints: Up to 10^6 scans. SKUs compare case-insensitively after stripping. Blank scans are ignored.

Examples:

- expected `["A1", "A1", "B2", "C3"]`, received `["a1", " B2", "B2", "D4"]` → missing `{"A1": 1, "C3": 1}`, unexpected `{"B2": 1, "D4": 1}`, matched `2`
- both empty → empty dicts, matched 0

Hints:

1. "Think of both lists as multisets."
2. "What does `Counter(a) - Counter(b)` do with counts that go to zero or below?"
3. "What does `&` mean for Counters?"

Reference solution:

```python
from collections import Counter
from dataclasses import dataclass


@dataclass(frozen=True)
class Reconciliation:
    missing: dict[str, int]
    unexpected: dict[str, int]
    matched: int


def _normalise(skus: list[str]) -> Counter[str]:
    return Counter(s.strip().upper() for s in skus if s.strip())


def reconcile(expected: list[str], received: list[str]) -> Reconciliation:
    """Compare expected vs received SKUs as multisets.

    missing    = units expected but not received
    unexpected = units received beyond what was expected (including unknown SKUs)
    matched    = units present in both
    Counter subtraction drops non-positive counts, which is exactly the
    semantics needed here.
    """
    exp, rec = _normalise(expected), _normalise(received)
    return Reconciliation(
        missing=dict(sorted((exp - rec).items())),
        unexpected=dict(sorted((rec - exp).items())),
        matched=sum((exp & rec).values()),
    )
```

Complexity: O(n + m + d log d) time for list sizes n, m and d distinct SKUs; O(d) space.

Tests:

```
r = reconcile(["A1", "A1", "B2", "C3"], ["a1", " B2", "B2", "D4"])
assert r == Reconciliation({"A1": 1, "C3": 1}, {"B2": 1, "D4": 1}, 2)
assert reconcile([], []) == Reconciliation({}, {}, 0)
assert reconcile(["X"], ["x"]) == Reconciliation({}, {}, 1)
assert reconcile(["X", "X"], ["", "  "]) == Reconciliation({"X": 2}, {}, 0)
assert reconcile([], ["Z9", "z9"]) == Reconciliation({}, {"Z9": 2}, 0)
# subtract() keeps negatives; the '-' operator drops them
c = Counter({"A1": 1}); c.subtract({"A1": 3})
assert c["A1"] == -2 and (Counter({"A1": 1}) - Counter({"A1": 3})) == Counter()
```

Alternatives and tradeoffs: Manual dicts with `get(k, 0)` work but reimplement multiset math and invite sign bugs. A SQL `FULL OUTER JOIN` on grouped counts is the answer when data lives in a warehouse. Pick: Counter in Python; SQL when the data is already there.

Follow-ups:

- "Scans arrive as a stream all day." Keep a running Counter per shipment; emit a report when the shipment is closed.
- "10^9 scans across 500 warehouses." Partition by `(warehouse, shipment)`, count per partition, merge; Counters (and SQL GROUP BY) are mergeable.
- "Tolerance: up to 1% short is OK." Compute `missing` per SKU and compare with `ceil(0.01 * expected)`; keep tolerance out of the core function.

Scoring: Strong: Counter subtraction and `&`, normalisation, knows `-` drops non-positive counts. Acceptable: dict-based and correct. Red flag: `set(expected) - set(received)`, which loses quantities.

M-C03 · Order pipeline tasks by dependency

Medium · Coding · DE, BE, MLE · Round 2 · graphs, topological sort, `deque` (the tier's one algorithm question) · 20 min

Problem: A job scheduler (think a tiny Airflow) receives `tasks: dict[str, list[str]]` mapping each task to the upstream tasks it depends on. Return an execution order in which every task comes after all its dependencies. Dependencies that aren't keys are still tasks (with no dependencies). When several tasks are ready, run them in first-seen order so the output is deterministic. On a cycle, raise `ValueError` naming the tasks involved.

Signature: `def execution_order(tasks: dict[str, list[str]]) -> list[str]:`

Constraints: Up to 10^5 tasks and 10^6 edges. Duplicate dependencies may appear. A task depending on itself is a cycle.

Examples:

- `{"load": ["transform"], "transform": ["extract"], "report": ["load", "transform"]}` → `["extract", "transform", "load", "report"]`
- `{"a": ["b"], "b": ["a"]}` → `ValueError` mentioning `a` and `b`
- `{}` → `[]`

Hints:

1. "Which tasks can run first?"
2. "Track how many unfinished dependencies each task still has."
3. "If the queue empties but tasks remain, what does that tell you?"

Reference solution:

```
from collections import deque


def execution_order(tasks: dict[str, list[str]]) -> list[str]:
    """Return a dependency-respecting execution order (Kahn's algorithm).

    `tasks` maps task -> upstream dependencies. Ready tasks run in first-seen
    order. Raises ValueError listing the tasks stuck in (or behind) a cycle.
    """
```

```
    nodes: dict[str, None] = {}                # ordered set of all task names
    for task, deps in tasks.items():
        nodes.setdefault(task)
        for dep in deps:
            nodes.setdefault(dep)
    pending = {node: 0 for node in nodes}      # unfinished dependency counts
    downstream: dict[str, list[str]] = {node: [] for node in nodes}
    for task, deps in tasks.items():
        for dep in dict.fromkeys(deps):        # ignore duplicate edges
            pending[task] += 1
            downstream[dep].append(task)

    ready = deque(node for node in nodes if pending[node] == 0)
    order: list[str] = []
    while ready:
        node = ready.popleft()
        order.append(node)
        for child in downstream[node]:
            pending[child] -= 1
            if pending[child] == 0:
                ready.append(child)

    if len(order) != len(nodes):
        stuck = sorted(node for node in nodes if pending[node] > 0)
        raise ValueError(f"cycle detected; blocked tasks: {stuck}")
    return order
```

Complexity: O(V + E) time (each task enqueued once, each edge relaxed once); O(V + E) space.

Tests:

```
dag = {"load": ["transform"], "transform": ["extract"], "report": ["load", "transform"]}
assert execution_order(dag) == ["extract", "transform", "load", "report"]
assert execution_order({}) == []
assert execution_order({"solo": []}) == ["solo"]
assert execution_order({"b": ["a", "a"], "c": ["a"]}) == ["a", "b", "c"]   # duplicate edge
for cyclic in ({"a": ["b"], "b": ["a"]}, {"x": ["x"]}):
    try:
        execution_order(cyclic)
        raise AssertionError("expected ValueError")
    except ValueError as exc:
        assert "blocked" in str(exc)

from graphlib import TopologicalSorter
static = list(TopologicalSorter(dag).static_order())
pos = {t: i for i, t in enumerate(static)}
assert all(pos[d] < pos[t] for t, deps in dag.items() for d in deps)
```

Alternatives and tradeoffs: `graphlib.TopologicalSorter` (stdlib, 3.9+) does this and supports parallel scheduling via `get_ready()/done()`; it raises `CycleError`. DFS with three colours finds cycles and can report the exact cycle path. Pick: `graphlib` in production; Kahn's by hand here because it shows the mechanism and gives deterministic ordering.

Follow-ups:

- "Run independent tasks in parallel." Everything in `ready` at the same moment is a wave; with `graphlib`, call `get_ready()`, dispatch to workers, call `done()` as each finishes.
- "Report the actual cycle path." DFS with a recursion stack; when you hit a grey node, unwind the stack to it.
- "A task fails; what reruns?" Everything downstream of it (a BFS over `downstream`), which is Airflow's "clear downstream".

Scoring: Strong: Kahn's or DFS, cycle detection, handles undeclared dependencies and duplicate edges, mentions `graphlib`. Acceptable: correct DFS that needs prompting for cycle detection. Red flag: repeated full scans ($O(V^2)$) or infinite loop on a cycle.

Part 2b · MEDIUM & TRICKY (MT)

The trap comes from two mechanisms interacting. A 3–4 year candidate is expected to get about half of these with correct reasoning; the reasoning matters more than the first answer.

MT-O01 · Per-column validators built in a comprehension

Medium & Tricky · Output · All roles · Round 2 · closures, late binding, comprehension scope · 6 min

Question: What does this print?

```
required = ["order_id", "customer_id", "amount"]
validators = [lambda row: row.get(col) is not None for col in required]

row = {"order_id": 7, "customer_id": None, "amount": 120.0}
print([check(row) for check in validators])
print(all(check(row) for check in validators))
print("col" in globals())

validators = [lambda row, col=col: row.get(col) is not None for col in required]
print([check(row) for check in validators])
```

Expected output:

```
[True, True, True]
True
False
[True, False, True]
```

Trap answer: `[True, False, True]`, `False`, and possibly `True` for the globals check. Tempting because each lambda "looks like" it was created with its own `col`.

Explanation: Two mechanisms interact. (1) A comprehension runs in its own scope, so `col` is a variable of that scope and never leaks into globals (hence `False`; true since Python 3; 3.12's PEP 709 inlines comprehensions but keeps this isolation). (2) A lambda that references `col` doesn't store its value; it closes over the *variable* (a cell) and looks it up when called. All three lambdas share the one `col` cell, which holds `"amount"` after the loop finishes, so every validator checks `amount`, and the null `customer_id` sails through. The fix binds the value at definition time: a default argument (`col=col`) is evaluated when each lambda is created. `functools.partial` or a factory function `make_check(col)` also works.

Real-world bug it maps to: A data-quality gate built from a list of column names passes rows with null keys; the pipeline loads orphaned records and the dashboard's join silently drops them.

Follow-ups:

1. "Same with a plain `for` loop appending lambdas?" Same late binding, and additionally `col` leaks into the enclosing scope after the loop.
2. "A class body with `cols = [...]` and `checks = [c for c in cols if c in allowed]` where `allowed` is a class attribute?" NameError: comprehension scopes skip class scope, except for the outermost iterable.
3. "Why is the default-argument fix sometimes discouraged?" It changes the signature; a caller can override `col`. A factory function or `partial` is cleaner.

Scoring: Strong: exact output, names late binding *and* comprehension scope. Acceptable: explains late binding after seeing the output. Red flag: believes each lambda captured its own value.

MT-O02 · Default headers leaking between requests

Medium & Tricky · Output · All roles · Round 2 · mutable default arguments, aliasing · 5 min

Question: What does this print?

```python
def build_headers(token, headers={}):
    headers["Authorization"] = f"Bearer {token}"
    return headers

alice = build_headers("alice-token")
alice["X-Tenant"] = "acme"
bob = build_headers("bob-token")
print(bob)
print(alice is bob, alice["Authorization"])
print(build_headers.__defaults__)
```

Expected output:

```
{'Authorization': 'Bearer bob-token', 'X-Tenant': 'acme'}
True Bearer bob-token
({'Authorization': 'Bearer bob-token', 'X-Tenant': 'acme'},)
```

Trap answer: `{'Authorization': 'Bearer bob-token'}`, `False Bearer alice-token`, and `({},)`. Tempting because `headers={}` reads like "a new empty dict each call".

Explanation: Default values are evaluated once, when the `def` statement executes, and stored in `build_headers.__defaults__`. Every call that omits `headers` receives that same dict object (mechanism 1). The function mutates it and returns it, so the caller now holds an alias to the shared default (mechanism 2). Alice's caller adds `X-Tenant` to what it thinks is its own dict; Bob's call overwrites `Authorization` in the same object; `alice is bob` is True and Alice's "copy" now carries Bob's token. Fix: `headers=None` then `headers = {} if headers is None else dict(headers)`, which both avoids the shared default and avoids mutating a dict the caller passed in.

Real-world bug it maps to: Credentials or tenant IDs leak across requests in a long-running API process; user A's call goes out with user B's token. This is a security incident, not a style issue.

Follow-ups:

1. "Why does Python evaluate defaults once?" The `def` statement is executed once and builds a function object; defaults are part of that object.
2. "Is `headers: dict | None = None` enough?" Only if you also copy a caller-supplied dict before mutating it.
3. "How would a linter catch this?" Ruff/flake8-bugbear rule B006 (mutable argument default).

Scoring: Strong: exact output and frames it as a cross-request leak. Acceptable: knows mutable defaults but misses the aliasing through the return value. Red flag: no idea why defaults persist.

MT-O03 · Processed-count helper and UnboundLocalError

Medium & Tricky · Output · All roles · Round 2 · scoping (LEGB), `global`/`nonlocal` · 5 min

Question: What happens when this runs?

```python
processed = 0
BATCH_LIMIT = 2

def mark_processed(order_id):
    if processed >= BATCH_LIMIT:
        print("limit reached")
        return False
    processed += 1
    print(f"processed {order_id}")
    return True

def make_counter():
    count = 0
    def mark(order_id):
        nonlocal count
        count += 1
        return f"{order_id}:{count}"
    return mark
```

```
mark = make_counter()
print(mark("A"), mark("B"))
print(mark_processed("A"))
```

Expected output:

```
A:1 B:2
Traceback (most recent call last):
  ...
UnboundLocalError: cannot access local variable 'processed' where it is not associated with a
value
```

(3.10 wording: `local variable 'processed' referenced before assignment`.)

Trap answer: `A:1 B:2`, `processed A`, `True`. Tempting because `processed` is plainly a global defined two lines above.

Explanation: Scope is decided at compile time, per function: any assignment to a name anywhere in a function body (including `+=`) makes that name local for the *entire* body, unless declared `global`/`nonlocal`. `processed += 1` makes `processed` local in `mark_processed`, so the earlier read in the `if` refers to an unassigned local and raises, even though the assignment comes later. `BATCH_LIMIT` is only read, so LEGB lookup finds the global. The closure works because `nonlocal count` tells the compiler to bind to the enclosing function's variable. Fix: `global processed` (quick), or better, keep the counter in an object or closure and pass state explicitly.

Real-world bug it maps to: A module-level counter or cache added "quickly" to a worker function crashes only on the code path that increments it, typically the first real run in production after a test that only exercised the early-return path.

Follow-ups:

1. "What would `global processed` change in a multi-threaded worker?" The error goes away but `+=` isn't atomic across threads; you need a lock or `itertools.count`.
2. "What does LEGB stand for?" Local, Enclosing, Global, Builtins, the lookup order for reads.
3. "Why is `nonlocal` rarely needed for mutable state?" `counts["n"] += 1` mutates without rebinding the name, so no declaration is required.

Scoring: Strong: predicts UnboundLocalError and explains compile-time scoping. Acceptable: suspects an error but calls it NameError, then corrects. Red flag: thinks Python resolves scope line by line at runtime.

MT-O04 · Purging expired sessions while iterating

Medium & Tricky · Output · All roles · Round 2 · iteration + mutation (lists and dicts) · 6 min

Question: What does this print?

```
sessions = [
    {"id": "s1", "expired": True},
    {"id": "s2", "expired": True},
    {"id": "s3", "expired": False},
    {"id": "s4", "expired": True},
    {"id": "s5", "expired": True},
]
for session in sessions:
    if session["expired"]:
        sessions.remove(session)
print([s["id"] for s in sessions])

active = {"u1": "ok", "u2": "expired"}
try:
    for user, state in active.items():
        if state == "expired":
            del active[user]
except RuntimeError as exc:
    print("RuntimeError:", exc)
```

Expected output:

```
['s2', 's3', 's5']
RuntimeError: dictionary changed size during iteration
```

Trap answer: `['s3']` and no error. Tempting because "remove every expired one" is what the code says.

Explanation: A list iterator is just an index into the live list (mechanism 1); `remove` shifts later elements left (mechanism 2). Trace: index 0 → s1 expired → removed, list is `[s2, s3, s4, s5]`; index 1 → s3 (s2 was skipped) → kept; index 2 → s4 → removed, list is `[s2, s3, s5]`; index 3 → past the end, loop stops. Two expired sessions (s2 and s5) survive and no error is raised. Dicts are different: their iterators detect a size change and raise RuntimeError on the next step, so here the failure is at least loud. Fix: build a new collection (`sessions = [s for s in sessions if not s["expired"]]`), or iterate over a copy (`for s in sessions[:]`, `for k in list(d)`).

Real-world bug it maps to: A cleanup job that "removes expired sessions" leaves roughly half of consecutive expired entries in place; tokens stay valid past their expiry, and nobody notices because no exception was raised.

Follow-ups:

1. "In-place, O(n), for a huge list?" Two-pointer compaction: `write = 0; for s in sessions: if keep: sessions[write] = s; write += 1`, then `del sessions[write:]`.
2. "Does modifying *values* of a dict during iteration raise?" No: `d[k] = new` for existing keys doesn't change the size; only inserts/deletes do.
3. "Sets?" Same as dicts: `RuntimeError: Set changed size during iteration`.

Scoring: Strong: exact list and the dict error, explains index shifting. Acceptable: knows it's unsafe but gets the surviving IDs wrong. Red flag: believes the list version works.

MT-O05 · Grouping events with itertools.groupby

Medium & Tricky · Output · All roles · Round 2 · `itertools.groupby`, shared iterators · 7 min

Question: What does this print?

```python
from itertools import groupby

events = [
    ("u1", "login"), ("u1", "click"), ("u2", "login"),
    ("u1", "logout"), ("u2", "click"),
]
by_user = {user: grp for user, grp in groupby(events, key=lambda e: e[0])}
print(list(by_user))
print([list(g) for g in by_user.values()])

counts = {user: len(list(grp)) for user, grp in groupby(events, key=lambda e: e[0])}
print(counts)

ordered = sorted(events, key=lambda e: e[0])
print({user: [a for _, a in grp] for user, grp in groupby(ordered, key=lambda e: e[0])})
```

Expected output:

```
['u1', 'u2']
[[], []]
{'u1': 1, 'u2': 1}
{'u1': ['login', 'click', 'logout'], 'u2': ['login', 'click']}
```

Trap answer: `[[('u1', 'login'), ('u1', 'click'), ('u1', 'logout')], [...]]` and `{'u1': 3, 'u2': 2}`. Tempting because "groupby" sounds like SQL `GROUP BY`.

Explanation: Mechanism 1: `groupby` only groups *consecutive* items with equal keys, like the Unix `uniq`. The input produces four groups: u1(2), u2(1), u1(1), u2(1); later groups overwrite earlier ones in the dict, so `counts` keeps the last run lengths, `{'u1': 1, 'u2': 1}`. Mechanism 2: every group is a view onto the *same* underlying iterator. When

`groupby` advances to the next key, it consumes the rest of the current group, and the old group object yields nothing afterwards. The first dict comprehension stored group objects without consuming them, so by the time we `list()` them they're empty. Fix: sort by the key first (stable, so event order within a user is preserved), and materialise each group immediately. Or skip `groupby` and use `defaultdict(list)`, which needs no sort.

Real-world bug it maps to: Per-user or per-partition aggregates computed with `groupby` on unsorted data under-count silently; the storing-groups variant produces empty batches that are then "successfully" written.

Follow-ups:

1. "When is `groupby` the right tool?" Already-sorted streams (e.g. a sorted file or `ORDER BY` query) where you want O(1) memory per group.
2. "`defaultdict(list)` vs sort+groupby?" O(n) with O(n) memory vs O(n log n) with streaming groups; pick defaultdict for in-memory data.
3. "pandas equivalent?" `df.groupby("user")` groups non-consecutive keys (SQL semantics), which is exactly why people expect `itertools.groupby` to do the same.

Scoring: Strong: all four lines, both mechanisms named. Acceptable: gets the consecutive-grouping issue, misses the empty groups. Red flag: SQL-semantics answer and no recovery after seeing the output.

MT-O06 · Spot the bug: a cleanup helper that swallows errors

Medium & Tricky · Output (spot the bug) · All roles · Round 2 · exceptions, `finally` semantics · 5 min

Question: This runs without errors. What does it print, and what's wrong with it?

```python
def close_batch(batch_id, rows):
    status = "unknown"
    try:
        if not rows:
            raise ValueError(f"batch {batch_id} is empty")
        status = f"committed {len(rows)} rows"
        return status
    finally:
        print(f"closing {batch_id}")
        return status

print(close_batch("b1", [1, 2]))
print(close_batch("b2", []))
```

Expected output:

```
closing b1
committed 2 rows
closing b2
unknown
```

(3.14+: the compiler emits `SyntaxWarning: 'return' in a 'finally' block`, per PEP 765.)

Trap answer: A `ValueError: batch b2 is empty` traceback after `closing b2`. Tempting because `finally` is "cleanup" and cleanup shouldn't change control flow.

Explanation: `finally` always runs (mechanism 1), and a `return` (or `break`/`continue`) inside `finally` *replaces* whatever was in flight: a pending return value or a propagating exception (mechanism 2). For b1 the `try` returns `"committed 2 rows"`, then `finally` returns `status`, which happens to be the same value, so it looks fine. For b2 the ValueError is propagating when `finally` executes `return status`; the exception is discarded and the function returns `"unknown"` as if nothing happened. The bug: an empty-batch error is silently converted into a normal return. Fix: keep `finally` for side-effect cleanup only and move the `return` after the `try` statement.

Real-world bug it maps to: A loader's close/cleanup helper reports "unknown" or "ok" for batches that actually failed; the orchestrator marks the task green, and the missing data surfaces days later in a reconciliation report.

Follow-ups:

1. "What if `finally` raised its own exception?" It replaces the original, but the original is preserved as `__context__`.
2. "Where else does this pattern hide?" `break`/`continue` inside `finally` in a loop, same swallowing.
3. "How do you log and still propagate?" `except Exception: log.exception(...); raise` (bare `raise` preserves the traceback).

Scoring: Strong: exact output and identifies the swallowed exception as the bug unprompted. Acceptable: predicts the traceback, understands once shown. Red flag: sees the output and says the code is fine.

MT-C01 · Memoize functions that take lists and dicts

Medium & Tricky · Coding · All roles (MLE, DS emphasis) · Round 2 · `functools.lru_cache`, hashing, mutability · 20 min

Problem: `build_features(user_ids: list[int], options: dict)` is expensive and is called repeatedly with equal arguments. Write a decorator factory `memoize_by_value(maxsize=128)` that caches by *value* even when arguments contain lists, dicts, sets, or tuples. Dict arguments and keyword arguments match regardless of key order. A list and a tuple with the same elements are *different* keys. Callers must not be able to corrupt the cache by mutating a returned value. Evict least-recently-used entries beyond `maxsize`. Expose `cache_info()` returning `(hits, misses, size)`.

Signature: `def memoize_by_value(maxsize: int = 128) -> Callable[[Callable[..., R]], Callable[..., R]]:`

Constraints: Arguments are JSON-like (plus sets and tuples); other unhashable types may raise TypeError. Single-threaded use.

Examples:

- `f([1, 2], {"a": 1, "b": 2})` then `f([1, 2], {"b": 2, "a": 1})` → one miss, then one hit
- `f([1, 2])` then `f((1, 2))` → two misses
- `r = f([1]); r.append("x"); f([1])` → still the original, uncorrupted result

Trap answer: Slapping `@functools.lru_cache` on it: `TypeError: unhashable type: 'list'`. The second trap is a hand-rolled cache that returns the stored object itself: the first caller mutates it and every later caller gets the corrupted value.

Real-world bug it maps to: A cached feature dict or config object is mutated by one request handler; every subsequent request serves poisoned data until the process restarts.

Hints:

1. "Why does `lru_cache` need hashable arguments?"
2. "How can you turn a list or dict into something hashable without losing information?"
3. "What happens if you hand the caller the exact object stored in your cache?"

Reference solution:

```python
import copy
import functools
from collections import OrderedDict
from collections.abc import Callable, Hashable
from typing import Any, TypeVar

R = TypeVar("R")


def _freeze(value: Any) -> Hashable:
    """Convert JSON-like values into a hashable, type-tagged key."""
    if isinstance(value, dict):
        return ("dict", frozenset((k, _freeze(v)) for k, v in value.items()))
    if isinstance(value, (list, tuple)):
        return (type(value).__name__, tuple(_freeze(v) for v in value))
    if isinstance(value, (set, frozenset)):
        return ("set", frozenset(_freeze(v) for v in value))
    hash(value)                      # raise TypeError early for other unhashables
```

```
    return value


def memoize_by_value(maxsize: int = 128) -> Callable[[Callable[..., R]], Callable[..., R]]:
    """LRU-cache a function by argument *value*, returning defensive copies."""
    if maxsize < 1:
        raise ValueError("maxsize must be >= 1")

    def decorator(func: Callable[..., R]) -> Callable[..., R]:
        cache: OrderedDict[Hashable, R] = OrderedDict()
        stats = {"hits": 0, "misses": 0}

        @functools.wraps(func)
        def wrapper(*args: Any, **kwargs: Any) -> R:
            key = (_freeze(args), _freeze(kwargs))
            if key in cache:
                cache.move_to_end(key)
                stats["hits"] += 1
                return copy.deepcopy(cache[key])
            stats["misses"] += 1
            result = func(*args, **kwargs)
            cache[key] = copy.deepcopy(result)   # caller keeps `result`; cache keeps its own
copy
            if len(cache) > maxsize:
                cache.popitem(last=False)
            return result

        def cache_info() -> tuple[int, int, int]:
            return stats["hits"], stats["misses"], len(cache)

        wrapper.cache_info = cache_info       # type: ignore[attr-defined]
        return wrapper

    return decorator
```

Complexity: O(size of arguments) per call to build the key, plus O(size of result) for each deepcopy; O(maxsize) entries of space.

Tests:

```
calls = []

@memoize_by_value(maxsize=2)
def build_features(user_ids, options=None):
    calls.append(1)
    return {"users": list(user_ids), "opts": dict(options or {})}

r1 = build_features([1, 2], {"a": 1, "b": 2})
r2 = build_features([1, 2], {"b": 2, "a": 1})            # dict order doesn't matter
assert r1 == r2 and len(calls) == 1
assert build_features.cache_info() == (1, 1, 1)

r2["users"].append(999)                                  # mutate a returned value
assert build_features([1, 2], {"a": 1, "b": 2})["users"] == [1, 2]

build_features((1, 2), {"a": 1, "b": 2})                 # tuple != list
assert len(calls) == 2

build_features([3], None)                                # cache full: evicts the LRU (list)
entry
assert build_features.cache_info()[2] == 2
before = len(calls)
build_features([1, 2], {"a": 1, "b": 2})                 # evicted, so recomputed
assert len(calls) == before + 1

build_features([7], options={"x": 1}); build_features(options={"x": 1}, user_ids=[7])
assert build_features.__name__ == "build_features"
```

```
try:
    build_features([object.__new__(type("Opaque", (), {"__hash__": None}))])
    raise AssertionError("expected TypeError")
except TypeError:
    pass

try:
    functools.lru_cache(lambda xs: sum(xs))([1, 2])
    raise AssertionError("expected TypeError")
except TypeError:
    pass
```

Alternatives and tradeoffs: Key by `json.dumps(args, sort_keys=True)`: simple, but fails on sets/tuples/non-JSON types and conflates list with tuple. Convert arguments to tuples at call sites and use `lru_cache`: fastest (C implementation), but pushes the burden onto every caller. Return frozen results (tuples, `MappingProxyType`) instead of deep copies: cheaper, but leaks immutability into the API. Pick: freezing keys plus defensive copies for correctness; switch to immutable return types when profiling shows the copies matter.

Follow-ups:

- "Make it thread-safe." A `threading.Lock` around cache reads/writes, and call `func` outside the lock (see H-C02).
- "`1`, `1.0` and `True` hash equal. Same cache entry?" Yes, and `lru_cache(typed=False)` behaves the same; add `type(v).__name__` to the key if that matters.
- "Results are 500 MB NumPy arrays." Deep copies become the bottleneck: return read-only views (`arr.setflags(write=False)`) instead.

Scoring: Strong: hashable type-tagged keys, order-insensitive dicts, defensive copies, LRU. Acceptable: key freezing right but returns the cached object until asked about mutation. Red flag: `str(args)` as the key with no awareness of dict-order or type collisions.

MT-C02 · Decode lines from a chunked byte stream

Medium & Tricky · Coding · DE, BE, MLE, FS · Round 2 · UTF-8, incremental decoding, generators · 20 min

Problem: An S3 object, HTTP response, or socket delivers a text file as `bytes` chunks of arbitrary size. Write a generator that yields complete text lines without line terminators. Chunks may split a line *and* a multi-byte UTF-8 character anywhere. Support `\n` and `\r\n` (including a `\r` / `\n` pair split across chunks). A final line without a trailing newline is still yielded. Invalid UTF-8 must raise, not be silently replaced.

Signature: `def iter_lines(chunks: Iterable[bytes], encoding: str = "utf-8") -> Iterator[str]:`

Constraints: Total size may exceed memory; individual lines fit in memory.

Examples:

- `[b"id,amt\n1,\xe2\x82", b"\xb9500\n2,9"]` → `"id,amt"`, `"1,₹500"`, `"2,9"`
- `[b"a\r", b"\nb\r\n"]` → `"a"`, `"b"`
- `[]` → nothing

Trap answer: `for chunk in chunks: yield from chunk.decode().split("\n")`. This raises `UnicodeDecodeError` when `₹` (3 bytes) straddles a boundary, and emits broken half-lines at every chunk boundary.

Real-world bug it maps to: Streaming ingestion passes every ASCII test, then crashes in production only when a `₹`, `é`, or emoji happens to land on a chunk boundary, i.e. intermittently and data-dependent.

Hints:

1. "How many bytes is `₹` in UTF-8? What does `b'\xe2\x82'.decode()` do?"
2. "Look at `codecs.getincrementaldecoder`."
3. "Keep a buffer of the incomplete last line between chunks."

Reference solution:

```python
import codecs
from collections.abc import Iterable, Iterator


def iter_lines(chunks: Iterable[bytes], encoding: str = "utf-8") -> Iterator[str]:
    """Yield text lines (without terminators) from arbitrarily split byte chunks.

    Handles multi-byte characters and \\r\\n pairs split across chunks. A final
    unterminated line is yielded. Invalid bytes raise UnicodeDecodeError.
    """
    decoder = codecs.getincrementaldecoder(encoding)(errors="strict")
    pending = ""
    for chunk in chunks:
        pending += decoder.decode(chunk)
        *complete, pending = pending.split("\n")
        for line in complete:
            yield line.removesuffix("\r")
    pending += decoder.decode(b"", final=True)   # raises if a character is truncated
    if pending:
        yield pending.removesuffix("\r")
```

Complexity: O(total bytes) time for normal line lengths; O(longest line + chunk size) memory. (A pathological single enormous line makes the repeated `split` quadratic; see follow-ups.)

Tests:

```python
data = "id,amt\n1,₹500\n2,café\r\n3,😀\nlast".encode()
expected = ["id,amt", "1,₹500", "2,café", "3,😀", "last"]
for size in (1, 2, 3, 5, 7, len(data)):                    # every split position
    chunks = [data[i:i + size] for i in range(0, len(data), size)]
    assert list(iter_lines(chunks)) == expected, size

assert list(iter_lines([])) == []
assert list(iter_lines([b"a\r", b"\nb\r\n"])) == ["a", "b"]
assert list(iter_lines([b"x\n\n", b"y\n"])) == ["x", "", "y"]     # blank line kept
try:
    list(iter_lines([b"ok\n", b"\xe2\x82"]))                # truncated final character
    raise AssertionError("expected UnicodeDecodeError")
except UnicodeDecodeError:
    pass
try:
    [line for c in [b"1,\xe2\x82", b"\xb9\n"] for line in c.decode().split("\n")]
    raise AssertionError("naive version should fail")
except UnicodeDecodeError:
    pass
```

Alternatives and tradeoffs: Wrap the stream in `io.TextIOWrapper(io.BufferedReader(raw))`: it does incremental decoding and universal newlines for you and is the production answer when you have a file-like object. `codecs.iterdecode(chunks, "utf-8")` handles decoding but not line splitting. Pick: `TextIOWrapper` if a file-like object exists; this generator when all you have is an iterable of chunks (e.g. `requests`' `iter_content`, aiohttp, boto3 `iter_chunks`).

Follow-ups:

- "A 2 GB file with no newlines." Keep pending pieces in a list and scan only the new chunk for `\n`, or enforce a maximum line length and raise; unbounded lines are a DoS vector.
- "The source is UTF-16 or has a BOM." Use `utf-8-sig` to strip a BOM; UTF-16 needs the incremental decoder all the more (every character is multi-byte).
- "Make it async." `async for chunk in stream`: the same logic in an async generator.

Scoring: Strong: incremental decoder, cross-chunk buffer, `\r\n` split handled, final flush. Acceptable: buffers lines correctly but decodes per chunk until hint 1. Red flag: `errors="ignore"` to make the exception go away.

MT-C03 · Typed settings from environment variables

Medium & Tricky · Coding · All roles · Round 2 · runtime types vs hints, truthiness, parsing · 15–20 min

Problem: Load service settings from environment variables with prefix `APP_` into a frozen dataclass: `db_url: str` (required), `pool_size: int = 5`, `debug: bool = False`, `timeout_s: float = 30.0`. Values arrive as strings. Booleans accept `1/true/yes/on` and `0/false/no/off` (case-insensitive, surrounding whitespace ignored); anything else is an error. Collect *all* problems (missing required, bad values, unknown `APP_` keys such as typos) and raise one `ValueError` listing them.

Signature: `def load_settings(env: Mapping[str, str], prefix: str = "APP_") -> Settings:`

Constraints: Standard library only. Adding a field to `Settings` must not require touching the loader.

Examples:

- `{"APP_DB_URL": "postgres://x", "APP_DEBUG": "false", "APP_POOL_SIZE": " 8 "}` → `Settings(db_url='postgres://x', pool_size=8, debug=False, timeout_s=30.0)`
- `{"APP_DEBUG": "maybe", "APP_DEBGU": "1"}` → ValueError mentioning `APP_DB_URL`, `APP_DEBUG`, and `APP_DEBGU`

Trap answer: `debug = bool(os.environ.get("APP_DEBUG", False))`. `bool("false")` is True because any non-empty string is truthy. Second trap: `pool_size: int = os.environ["APP_POOL_SIZE"]` stays a string, since hints aren't enforced (ET-O06).

Real-world bug it maps to: `APP_DEBUG=false` in production turns debug mode *on*, exposing stack traces and internal settings to users.

Hints:

1. "What is `bool("false")`?"
2. "How can the loader discover field names and types without hardcoding them?" (`dataclasses.fields`, `typing.get_type_hints`)
3. "Why collect errors instead of raising at the first one?"

Reference solution:

```python
import typing
from collections.abc import Callable, Mapping
from dataclasses import MISSING, dataclass, fields


@dataclass(frozen=True)
class Settings:
    db_url: str
    pool_size: int = 5
    debug: bool = False
    timeout_s: float = 30.0


_TRUE = {"1", "true", "yes", "on"}
_FALSE = {"0", "false", "no", "off"}


def _parse_bool(raw: str) -> bool:
    value = raw.strip().lower()
    if value in _TRUE:
        return True
    if value in _FALSE:
        return False
    raise ValueError(f"not a boolean: {raw!r}")


_PARSERS: dict[type, Callable[[str], object]] = {
    bool: _parse_bool,
    int: lambda raw: int(raw.strip()),
```

```
    float: lambda raw: float(raw.strip()),
    str: lambda raw: raw,
}


def load_settings(env: Mapping[str, str], prefix: str = "APP_") -> Settings:
    """Build Settings from prefixed env vars, reporting every problem at once."""
    hints = typing.get_type_hints(Settings)
    values: dict[str, object] = {}
    errors: list[str] = []
    known: set[str] = set()
    for f in fields(Settings):
        key = prefix + f.name.upper()
        known.add(key)
        if key not in env:
            if f.default is MISSING and f.default_factory is MISSING:
                errors.append(f"{key}: required")
            continue
        try:
            values[f.name] = _PARSERS[hints[f.name]](env[key])
        except ValueError as exc:
            errors.append(f"{key}: {exc}")
    errors += [f"{k}: unknown setting" for k in sorted(env) if k.startswith(prefix) and k not
in known]
    if errors:
        raise ValueError("invalid settings: " + "; ".join(errors))
    return Settings(**values)
```

Complexity: O(fields + env size) time and space.

Tests:

```
s = load_settings({"APP_DB_URL": "postgres://x", "APP_DEBUG": "false", "APP_POOL_SIZE": " 8 "})
assert s == Settings(db_url="postgres://x", pool_size=8, debug=False, timeout_s=30.0)
assert load_settings({"APP_DB_URL": "d", "APP_DEBUG": " YES "}).debug is True
assert load_settings({"APP_DB_URL": "d", "HOME": "/root"}).pool_size == 5   # other vars
ignored
try:
    load_settings({"APP_DEBUG": "maybe", "APP_DEBGU": "1", "APP_TIMEOUT_S": "fast"})
    raise AssertionError("expected ValueError")
except ValueError as exc:
    msg = str(exc)
    assert all(k in msg for k in ("APP_DB_URL: required", "APP_DEBUG", "APP_DEBGU",
"APP_TIMEOUT_S"))
try:
    load_settings({"APP_DB_URL": "d", "APP_POOL_SIZE": "8.0"})
    raise AssertionError("expected ValueError")
except ValueError as exc:
    assert "APP_POOL_SIZE" in str(exc)
try:
    load_settings({})
except ValueError as exc:
    assert "APP_DB_URL: required" in str(exc)
assert bool("false") is True                                  # the trap, documented
```

Alternatives and tradeoffs: `pydantic-settings` (`BaseSettings`) does exactly this, plus `.env` files, secrets directories, and nested models; it's the production pick in a FastAPI codebase. `environs`/`python-decouple` are lighter. Pick: `pydantic-settings` when Pydantic is already a dependency; this stdlib version for tools that must stay dependency-free.

Follow-ups:

- "`int("1_000")` succeeds. Is that OK?" Python accepts underscores in numeric strings; decide explicitly whether config should.
- "`float("nan")` and `float("inf")` parse. Is that OK for a timeout?" No: add range validation (`timeout_s > 0` and finite) in `__post_init__`.

- "Secrets (`APP_DB_URL` has a password)?" Never include raw values in error messages or logs; this loader only echoes values for parse errors, so mask secret fields.

Scoring: Strong: explicit bool parsing, type-driven parsing, all errors collected, unknown keys flagged. Acceptable: correct with hardcoded fields. Red flag: `bool(raw)`, or trusts the dataclass hints to convert types.

Part 3a • HARD (H)

A stretch at four years: internals, concurrency, performance, non-trivial design. Output questions 8–12 minutes; coding tasks 25–35 minutes. A candidate at the bar is solid on most of these.

H-O01 · ThreadPoolExecutor.map ordering and failures

Hard · Output · DE, BE, MLE, FS · Round 4 · threads, futures, GIL · 8 min

Question: What does this print? (Sleep durations vary, but the output below does not depend on timing.)

```python
import time
from concurrent.futures import ThreadPoolExecutor

def fetch_price(sku):
    time.sleep({"A": 0.3, "B": 0.1, "C": 0.2, "D": 0.0}[sku])
    if sku == "C":
        raise TimeoutError(f"pricing timeout for {sku}")
    return f"{sku}:ok"

with ThreadPoolExecutor(max_workers=4) as pool:
    results = pool.map(fetch_price, ["A", "B", "C", "D"])
    try:
        for r in results:
            print(r)
    except TimeoutError as exc:
        print("failed:", exc)
print("done")
```

Expected output:

```
A:ok
B:ok
failed: pricing timeout for C
done
```

Explanation: `pool.map` submits all four calls immediately and returns an iterator that yields results *in input order*, blocking on each future's `.result()` in turn. D finishes first and B before A, but nothing is printed until A's result is available. When the iterator reaches C, `.result()` re-raises the exception from the worker thread in the caller's thread, which ends the `for` loop; D's result was computed but is never seen. Leaving the `with` calls `shutdown(wait=True)`, so "done" prints only after every thread has finished. On the GIL: `time.sleep` (like socket and file I/O) releases the GIL, so the four waits overlap and threads genuinely help. Pure-Python CPU work would not speed up, because only one thread executes bytecode at a time on the standard build. Takeaway: `map` = ordered results and fail-fast on iteration; use `submit` + `as_completed` for per-item error handling.

Follow-ups:

1. "Collect every result and every error instead of stopping at C." `futures = {pool.submit(fetch_price, s): s for s in skus}`, then iterate `as_completed(futures)` and call `f.exception()` or wrap `f.result()` in try/except. Completion order, so store results keyed by SKU.
2. "CPU-bound work instead?" `ProcessPoolExecutor`: arguments and results must be picklable, and the entry point needs an `if __name__ == "__main__":` guard because child processes re-import the main module under the `spawn`/`forkserver` start methods (3.14 changed the Linux default from `fork` to `forkserver`).
3. "Free-threaded Python?" 3.13 ships an experimental no-GIL build (PEP 703); 3.14 makes it officially supported but still optional (PEP 779). Most deployments still run the GIL build.

Scoring: Strong: exact output, input-ordered results, where the exception surfaces, why threads help here. Acceptable: gets the ordering right after thinking about futures. Red flag: prints in completion order (D first), or believes threads speed up CPU-bound Python.

H-O02 · A blocking call inside a coroutine

Hard · Output · BE, FS, MLE, DE · Round 4 · asyncio event loop, blocking, `gather` · 10 min

Question: What does this print?

```python
import asyncio
import time

async def fetch(name, steps):
    print(f"start {name}")
    for _ in range(steps):
        await asyncio.sleep(0)
    print(f"end {name}")
    return name

async def blocking_fetch(name):
    print(f"start {name}")
    time.sleep(0.05)
    print(f"end {name}")
    return name

async def main():
    results = await asyncio.gather(fetch("A", 2), blocking_fetch("B"), fetch("C", 1))
    print(results)

asyncio.run(main())
```

Expected output:

```
start A
start B
end B
start C
end C
end A
['A', 'B', 'C']
```

Explanation: `gather` wraps each coroutine in a Task and schedules them in argument order on a single-threaded event loop. A task runs until it hits an `await` that actually suspends; `asyncio.sleep(0)` suspends for exactly one loop iteration. Trace: A prints and suspends; B prints, then `time.sleep` blocks the *entire thread* (no other task can run for 50 ms), prints, and finishes without ever yielding; C prints and suspends; A resumes, hits its second `sleep(0)`, suspends; C resumes and finishes; A resumes and finishes. `gather` returns results in argument order, not completion order. Takeaway: one synchronous call inside a coroutine stalls every concurrent request on that loop; use `await asyncio.sleep`, async clients, or `await asyncio.to_thread(blocking_fn)`.

Follow-ups:

1. "In FastAPI, what's the difference between `def endpoint()` and `async def endpoint()`?" `def` endpoints run in a threadpool, so blocking calls are tolerable; `async def` endpoints run on the loop, so any blocking call inside them stalls all requests.
2. "How would you detect blocking in production?" `asyncio.run(main(), debug=True)` / `loop.slow_callback_duration` logs callbacks that take too long; latency spikes correlated across unrelated endpoints are the symptom.
3. "`gather(..., return_exceptions=True)`?" Exceptions are returned as values in the result list instead of propagating; without it, the first exception propagates but the other tasks keep running.

Scoring: Strong: exact interleaving, explains that B never yields and why ordering is deterministic. Acceptable: understands blocking, slips on A/C order. Red flag: believes `time.sleep` in a coroutine lets other tasks run.

H-O03 · yield from and a sub-generator's return value

Hard · Output · All roles · Round 4 · generators, `yield from`, `StopIteration.value` · 8 min

Question: What does this print?

```python
def parse_batch(lines):
    good = bad = 0
    for line in lines:
        if line.isdigit():
            good += 1
            yield int(line)
        else:
            bad += 1
    return {"good": good, "bad": bad}

def ingest(batches):
    totals = []
    for batch in batches:
        stats = yield from parse_batch(batch)
        totals.append(stats)
    return totals

gen = ingest([["1", "x", "3"], ["y"], ["5"]])
print(list(gen))
gen = ingest([["7"], ["z"]])
while True:
    try:
        print("value", next(gen))
    except StopIteration as stop:
        print("summary", stop.value)
        break
```

Expected output:

```
[1, 3, 5]
value 7
summary [{'good': 1, 'bad': 0}, {'good': 0, 'bad': 1}]
```

Explanation: A generator's `return value` doesn't yield anything; it raises `StopIteration(value)` when the generator finishes. `yield from sub` does three jobs: it forwards every value `sub` yields straight to the outer consumer (plus `send`/`throw`), and when `sub` finishes, the `yield from` *expression evaluates to* `sub`'s return value. So `stats` receives each batch's counts, and `ingest` returns the list of them. `list(gen)` consumes only yielded values and silently discards the final return value. The manual loop catches `StopIteration` and reads `.value`. Note the batch `["y"]` yields nothing, yet its stats are still collected. Takeaway: `yield from` is how generators compose, and return values are how they report summaries without polluting the data stream.

Follow-ups:

1. "Why not just `for x in parse_batch(batch): yield x`?" That forwards values but loses the return value and doesn't forward `send()`/`throw()`/`close()` to the sub-generator.
2. "What happens if a generator raises `StopIteration` explicitly inside its body?" Since 3.7 (PEP 479) it's converted to `RuntimeError`, preventing silent early termination.
3. "Where does this pattern show up for real?" Streaming parsers that report malformed-row counts, and the historical foundation of `async`/`await` (coroutines were built on `yield from`).

Scoring: Strong: all three lines and explains that `list()` drops the return value. Acceptable: gets values, needs a hint for the summary. Red flag: thinks `return` inside a generator yields the value.

H-O04 · `__slots__` on a high-volume event record

Hard · Output · All roles · Round 4 · `__slots__`, `dataclass(slots=True)`, attribute storage · 8 min

Question: What does this print?

```python
from dataclasses import dataclass

class ClickEvent:
    __slots__ = ("user_id", "url")

    def __init__(self, user_id, url):
        self.user_id = user_id
        self.url = url

@dataclass(slots=True)
class ViewEvent:
    user_id: int
    url: str

e = ClickEvent(1, "/home")
try:
    e.usr_id = 2
except AttributeError as exc:
    print("AttributeError:", exc)
print(hasattr(e, "__dict__"), hasattr(ViewEvent(1, "/"), "__dict__"))

class TaggedClick(ClickEvent):
    pass

t = TaggedClick(3, "/cart")
t.campaign = "diwali"
print(t.campaign, "__dict__" in dir(t))

try:
    class Broken:
        __slots__ = ("retries",)
        retries = 3
except ValueError as exc:
    print("ValueError:", exc)
```

Expected output (3.12):

```
AttributeError: 'ClickEvent' object has no attribute 'usr_id'
False False
diwali True
ValueError: 'retries' in __slots__ conflicts with class variable
```

(3.13+: the first message becomes `'ClickEvent' object has no attribute 'usr_id' and no __dict__ for setting new attributes`.)

Explanation: Normally instance attributes live in a per-instance `__dict__`, a hash table, which is flexible but costs memory for each object. `__slots__` tells the class to allocate fixed storage for exactly the named attributes, implemented as descriptors on the class, and to skip `__dict__`. Assigning an unknown name (here a typo, `usr_id`) therefore raises AttributeError instead of silently creating a new attribute. `dataclass(slots=True)` (3.10+) generates the same thing. A subclass that doesn't declare `__slots__` gets a `__dict__` again, so `TaggedClick` accepts arbitrary attributes and the memory saving is lost. A class variable with the same name as a slot is rejected, because both would be class attributes and the slot descriptor would be overwritten. Takeaway: `__slots__` saves memory and catches typos for millions of small objects, but every class in the hierarchy must cooperate.

Follow-ups:

1. "How much memory does it save?" Typically on the order of 50–60% for small objects (no per-instance dict); measure with `tracemalloc`, not `sys.getsizeof` (H-O05).
2. "What do you lose?" Dynamic attributes, `functools.cached_property` (it needs `__dict__`), and weak references

unless `"__weakref__"` is in the slots.

3. "Default values with slots?" Set them in `__init__` (or via the dataclass), not as class attributes, because of the conflict shown in the last line.

Scoring: Strong: all four lines, explains descriptors/no `__dict__` and the subclass pitfall. Acceptable: first two lines right, misses the subclass. Red flag: thinks `__slots__` is only an optimisation hint with no behavioural change.

H-O05 · Measuring memory with sys.getsizeof

Hard · Output · All roles · Round 4 · memory, `sys.getsizeof`, generators vs lists (CPython 64-bit) · 8 min

Question: What does this print on 64-bit CPython?

```python
import sys

small = [c for c in "abc"]
large = [c * 10_000 for c in "abc"]
print(sys.getsizeof(small) == sys.getsizeof(large))

payload = {"user": "u1", "embedding": [0.1] * 1536}
print(sys.getsizeof(payload) < sys.getsizeof(payload["embedding"]))

squares_list = [i * i for i in range(1_000_000)]
squares_gen = (i * i for i in range(1_000_000))
print(sys.getsizeof(squares_gen) < 1_000 < sys.getsizeof(squares_list))
print(sys.getsizeof(1) == sys.getsizeof(2 ** 100))
```

Expected output:

```
True
True
True
False
```

Explanation: `sys.getsizeof` is *shallow*: it reports the memory of the object's own structure, not of the objects it references. A list stores pointers (8 bytes each on 64-bit), so two 3-element lists built the same way have the same size whether they point at 1-character or 10,000-character strings. The dict holding a 1536-float embedding reports less than the list inside it, because the dict only counts its own hash table. A generator object is a small fixed-size frame (a couple of hundred bytes) no matter how many items it will produce, while the list of a million squares holds eight megabytes of pointers plus the int objects themselves, which `getsizeof` doesn't count. Python ints are arbitrary-precision, so `2**100` needs more storage than `1`. Takeaway: `getsizeof` answers "how big is this container's shell"; for real footprints use `tracemalloc`, `pympler.asizeof`, or process RSS.

Follow-ups:

1. "How would you measure the real memory of `payload`?" `tracemalloc` snapshots before and after building it, or `pympler.asizeof`; for NumPy arrays, `arr.nbytes`.
2. "Rough real size of 1,536 Python floats in a list vs a NumPy float32 array?" About 12 KB of pointers plus about 24 bytes per distinct float object (~49 KB total) vs 6 KB contiguous. Why embeddings belong in arrays.
3. "Does the generator keep its small size after being consumed?" Yes; generators hold only the current frame state.
4. "Why are both lists built with comprehensions?" List size also depends on *how* the list was built. In CPython 3.12, `["a", "b", "c"]` (all constants) is built by extending from a constant tuple and over-allocates (88 bytes), while `["a" * 10_000, ...]` (runtime expressions) is built at exact size (80 bytes), so two 3-element lists report different sizes. One more reason not to trust `getsizeof` for comparisons.

Scoring: Strong: all four and explains shallow measurement. Acceptable: three of four, fixes after the "shallow" hint. Red flag: uses `getsizeof` totals to justify a memory optimisation.

H-O06 · Circular import between orders and pricing

Hard · Output · All roles · Round 4 · import system, `sys.modules`, circular imports · 10 min

Question: The first block writes two real modules to a temp directory. What does this print?

```
import sys, tempfile, textwrap
from pathlib import Path

modules = {
    "orders": """
        print("orders: start")
        from pricing import price_for
        TAX_RATE = 0.18
        print("orders: end")
    """,
    "pricing": """
        print("pricing: start")
        from orders import TAX_RATE
        def price_for(sku):
            return 100.0 * (1 + TAX_RATE)
        print("pricing: end")
    """,
}
src = Path(tempfile.mkdtemp())
for name, code in modules.items():
    (src / f"{name}.py").write_text(textwrap.dedent(code))
sys.path.insert(0, str(src))
try:
    import orders
except ImportError as exc:
    print("ImportError:", str(exc).split(" (")[0])
print("orders" in sys.modules, "pricing" in sys.modules)
```

Expected output:

```
orders: start
pricing: start
ImportError: cannot import name 'TAX_RATE' from partially initialized module 'orders'
False False
```

(The full message continues with "(most likely due to a circular import) ()".)

Explanation: `import orders` creates an empty module object, registers it in `sys.modules` *before* executing it (that registry is the import cache), then runs the file top to bottom. Line 2 of `orders` imports `pricing`, which starts executing and immediately asks for `orders.TAX_RATE`. `sys.modules["orders"]` exists, so Python doesn't re-import it; but that module is only partially initialised and `TAX_RATE` hasn't been assigned yet, so `from orders import TAX_RATE` fails. Neither "end" line prints. Because both imports failed, Python removes both half-built modules from `sys.modules`, so a later import retries cleanly. Fixes: move shared constants into a third module (`config.py`) that both import; import lazily inside the function that needs it; or use `import orders` and access `orders.TAX_RATE` at call time (attribute access is deferred).

Follow-ups:

1. "Why does `import orders` + `orders.TAX_RATE` inside `price_for` work?" The name is looked up when the function is *called*, after both modules have finished executing.
2. "Run `python orders.py` directly. What's different?" The file runs as `__main__`; when `pricing` does `from orders import ...`, Python imports a *second* copy named `orders`, so top-level code runs twice. The `if __name__ == "__main__":` guard keeps script-only code out of the importable module.
3. "Why is module-level code executed only once per process?" Subsequent imports hit `sys.modules`; use `importlib.reload` to force it (rarely wise).

Scoring: Strong: exact output, explains `sys.modules` registration before execution, offers a structural fix. Acceptable:

knows "circular import" but not why a *partially* initialised module exists. Red flag: suggests wrapping the import in try/except as the fix.

H-C01 · Bounded-concurrency async fetcher

Hard · Coding · BE, FS, DE, MLE · Round 4 · asyncio, Semaphore, timeouts, cancellation · 30 min

Problem: Enrich 50,000 records by calling an async HTTP API. The provider allows at most `limit` concurrent requests. Write `fetch_all(items, fetch, limit, timeout)` that calls `await fetch(item)` for every item with at most `limit` in flight, applies a per-request timeout (measured from when the request starts, not while it waits for a slot), and returns one `Result` per item *in input order*, capturing exceptions per item instead of failing the whole batch. Cancelling the caller must cancel all work; cancellation must never be swallowed.

Signature: `async def fetch_all(items: Sequence[T], fetch: Callable[[T], Awaitable[R]], limit: int, timeout: float) -> list[Result[T, R]]:`

Constraints: `limit >= 1`. `Result` is a frozen dataclass with `item`, `value`, `error`.

Examples:

- 5 items, `limit=2` → never more than 2 `fetch` calls in flight; 5 results in input order
- An item whose fetch raises `ConnectionError` → its `Result.error` is that exception; others succeed
- A fetch that never completes → `Result.error` is a `TimeoutError`

Hints:

1. "Which asyncio primitive limits how many coroutines enter a block at once?"
2. "Where should the timeout start: before or after acquiring the slot?"
3. "`CancelledError` is a `BaseException` subclass. Which `except` clause should you use?"

Reference solution:

```python
import asyncio
from collections.abc import Awaitable, Callable, Sequence
from dataclasses import dataclass
from typing import Generic, TypeVar

T = TypeVar("T")
R = TypeVar("R")


@dataclass(frozen=True)
class Result(Generic[T, R]):
    item: T
    value: R | None = None
    error: Exception | None = None


async def fetch_all(
    items: Sequence[T],
    fetch: Callable[[T], Awaitable[R]],
    limit: int,
    timeout: float,
) -> list[Result[T, R]]:
    """Run fetch(item) for all items with at most `limit` in flight.

    Results are returned in input order. Per-item exceptions (including
    timeouts) are captured in Result.error; cancellation propagates.
    """
    if limit < 1:
        raise ValueError("limit must be >= 1")
    semaphore = asyncio.Semaphore(limit)

    async def run_one(item: T) -> Result[T, R]:
        async with semaphore:                        # wait for a slot first ...
```

```
            try:                                        # ... then start the timeout clock
                value = await asyncio.wait_for(fetch(item), timeout)
            except Exception as exc:                    # not BaseException: CancelledError must
propagate
                return Result(item, error=exc)
            return Result(item, value=value)

    return list(await asyncio.gather(*(run_one(item) for item in items)))
```

Complexity: O(n) tasks and results; at most `limit` concurrent fetches. Memory is O(n) because every item gets a task up front (see follow-ups).

Tests:

```
async def scenario() -> None:
    in_flight = peak = 0

    async def fetch(sku: str) -> str:
        nonlocal in_flight, peak
        in_flight += 1
        peak = max(peak, in_flight)
        try:
            for _ in range(3):
                await asyncio.sleep(0)
            if sku == "bad":
                raise ConnectionError("refused")
            if sku == "hang":
                await asyncio.Event().wait()          # never completes
            return sku.upper()
        finally:
            in_flight -= 1

    skus = ["a", "b", "bad", "c", "hang", "d"]
    results = await fetch_all(skus, fetch, limit=2, timeout=0.05)
    assert [r.item for r in results] == skus
    assert [r.value for r in results] == ["A", "B", None, "C", None, "D"]
    assert isinstance(results[2].error, ConnectionError)
    assert isinstance(results[4].error, TimeoutError)
    assert peak == 2 and in_flight == 0

    assert await fetch_all([], fetch, limit=3, timeout=1) == []
    single = await fetch_all(["x"], fetch, limit=1, timeout=1)
    assert single == [Result("x", "X", None)]

    task = asyncio.create_task(fetch_all(["hang"] * 3, fetch, limit=3, timeout=10))
    await asyncio.sleep(0.01)
    task.cancel()
    try:
        await task
        raise AssertionError("cancellation was swallowed")
    except asyncio.CancelledError:
        pass
    await asyncio.sleep(0)
    assert in_flight == 0

    try:
        await fetch_all(["a"], fetch, limit=0, timeout=1)
        raise AssertionError("expected ValueError")
    except ValueError:
        pass

asyncio.run(scenario())
```

Alternatives and tradeoffs: A fixed pool of `limit` worker tasks pulling from an `asyncio.Queue` keeps memory at O(limit) instead of O(n) tasks, better for millions of items or unbounded streams. `asyncio.TaskGroup` (3.11+) gives structured concurrency but cancels siblings on the first unhandled error, which is the opposite of per-item capture

unless each task catches its own errors (as here). Pick: semaphore + gather for up to tens of thousands of items; worker pool for large or streaming inputs.

Follow-ups:

- "The provider also enforces 100 requests/second." Add a token bucket (BE-C01) in front of the semaphore: concurrency and rate are different limits.
- "Retry failed items with backoff." Wrap `fetch` in an async retry (M-C01 adapted with `asyncio.sleep`); only retry idempotent calls and transient errors.
- "Version note." Since 3.11, `asyncio.TimeoutError` is an alias of the builtin `TimeoutError`; on 3.10 you must catch `asyncio.TimeoutError`.

Scoring: Strong: semaphore, timeout inside the slot, `except Exception`, ordered results, discusses O(n) tasks. Acceptable: correct but timeout wraps the semaphore wait, or catches `BaseException` and fixes it after hint 3. Red flag: `asyncio.run` inside the function, or sequential awaits in a loop.

H-C02 · Thread-safe TTL cache decorator

Hard · Coding · BE, MLE, DE · Round 4 · threads, locks, `OrderedDict`, `functools.wraps` · 30 min

Problem: Services call slow lookups (exchange rates, feature-flag configs, model metadata) from many threads. Write `ttl_cache(ttl, maxsize, clock)`: cache results for `ttl` seconds per argument combination; evict least-recently-used entries beyond `maxsize`; never cache exceptions; be safe under concurrent calls; and never hold the lock while the slow function runs. The clock is injectable for tests.

Signature: `def ttl_cache(ttl: float, maxsize: int = 256, clock: Callable[[], float] = time.monotonic) -> Callable[[Callable[P, R]], Callable[P, R]]:`

Constraints: Arguments are hashable; keyword argument order must not matter. Expose `cache_clear()`.

Examples:

- Call at t=0 and t=5 with `ttl=10` → one underlying call
- Call at t=0 and t=11 → two underlying calls
- A call that raises → nothing cached; the next call retries

Hints:

1. "Why `time.monotonic()` rather than `time.time()`?"
2. "What goes wrong if you hold the lock while the wrapped function makes a 2-second HTTP call?"
3. "What does `OrderedDict.move_to_end` give you?"

Reference solution:

```python
import functools
import threading
import time
from collections import OrderedDict
from collections.abc import Callable, Hashable
from typing import Any, ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")


def ttl_cache(
    ttl: float,
    maxsize: int = 256,
    clock: Callable[[], float] = time.monotonic,
) -> Callable[[Callable[P, R]], Callable[P, R]]:
    """Thread-safe LRU cache whose entries expire `ttl` seconds after being stored.

    The lock guards only the cache bookkeeping; the wrapped function runs
    outside it, so a slow call never blocks hits for other keys. Exceptions
```

```
    are not cached. Concurrent misses on the same key may compute twice.
    """
    if ttl <= 0 or maxsize < 1:
        raise ValueError("ttl must be > 0 and maxsize >= 1")

    def decorator(func: Callable[P, R]) -> Callable[P, R]:
        entries: OrderedDict[Hashable, tuple[float, Any]] = OrderedDict()
        lock = threading.Lock()

        @functools.wraps(func)
        def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
            key = (args, frozenset(kwargs.items()))
            with lock:
                entry = entries.get(key)
                if entry is not None:
                    expires_at, value = entry
                    if clock() < expires_at:
                        entries.move_to_end(key)
                        return value
                    del entries[key]                        # expired: drop it
            value = func(*args, **kwargs)                    # slow part, lock NOT held
            with lock:
                entries[key] = (clock() + ttl, value)
                entries.move_to_end(key)
                while len(entries) > maxsize:
                    entries.popitem(last=False)
            return value

        def cache_clear() -> None:
            with lock:
                entries.clear()

        wrapper.cache_clear = cache_clear                    # type: ignore[attr-defined]
        return wrapper

    return decorator
```

Complexity: O(1) average per hit or insert (dict + linked-list reordering), plus key construction O(number of arguments); O(maxsize) space.

Tests:

```
now = [0.0]
calls: list[str] = []

@ttl_cache(ttl=10, maxsize=2, clock=lambda: now[0])
def fx_rate(pair: str, *, source: str = "ecb") -> float:
    """Look up an FX rate."""
    calls.append(pair)
    if pair == "BAD":
        raise LookupError(pair)
    return {"USDINR": 83.1, "EURINR": 90.2, "GBPINR": 105.0}[pair]

assert fx_rate("USDINR") == 83.1
now[0] = 5
assert fx_rate("USDINR") == 83.1 and calls == ["USDINR"]            # hit within ttl
now[0] = 11
fx_rate("USDINR"); assert calls == ["USDINR", "USDINR"]              # expired -> recomputed
fx_rate("EURINR"); fx_rate("GBPINR")                                 # maxsize=2 evicts LRU
(USDINR)
fx_rate("USDINR"); assert calls.count("USDINR") == 3
assert fx_rate("EURINR", source="x") == fx_rate(source="x", pair="EURINR")   # kw order
irrelevant

for _ in range(2):
    try:
        fx_rate("BAD")
```

```
    except LookupError:
        pass
assert calls.count("BAD") == 2                                  # exceptions never cached
assert fx_rate.__name__ == "fx_rate"

fx_rate.cache_clear()
before = len(calls)
fx_rate("USDINR"); assert len(calls) == before + 1

counter = {"n": 0}
@ttl_cache(ttl=1000, maxsize=50)
def square(x: int) -> int:
    counter["n"] += 1
    return x * x

def hammer() -> None:
    for i in range(2000):
        assert square(i % 100) == (i % 100) ** 2

threads = [threading.Thread(target=hammer) for _ in range(8)]
for t in threads: t.start()
for t in threads: t.join()
assert counter["n"] >= 100                                      # no crash, results
correct

try:
    ttl_cache(ttl=0)
    raise AssertionError("expected ValueError")
except ValueError:
    pass
```

Alternatives and tradeoffs: `cachetools.TTLCache` plus `cachetools.cached(cache, lock=threading.Lock())` is the production answer; it also computes outside the lock. Single-flight (one in-flight computation per key, other callers wait on a `Future`) prevents a cache stampede when a hot key expires, at the cost of more complex code and a failure mode where waiters share one error. Redis/Memcached for caches shared across processes. Pick: `cachetools` in-process; add single-flight for hot keys backed by expensive calls; Redis when multiple replicas must share entries.

Follow-ups:

- "Why `monotonic` rather than `time.time()`?" Wall-clock time jumps (NTP sync, manual changes); a backwards jump would make entries live far too long.
- "Implement single-flight." A dict `key → Future` guarded by the lock: the first caller computes and sets the result; others `future.result()`.
- "Does this help in `async` code?" Not as-is: `threading.Lock` would block the loop; use an async variant with `asyncio.Lock` and awaitable results.

Scoring: Strong: lock scoped to bookkeeping only, monotonic clock, no exception caching, LRU eviction, names stampede risk. Acceptable: correct but holds the lock during the call, fixes it after hint 2. Red flag: no lock, or `time.time()` with no awareness of clock jumps.

H-C03 · External sort of a file larger than memory

Hard · Coding · DE, MLE, BE · Round 4 · memory, generators, `heapq.merge`, `contextlib.ExitStack`, `tempfile` · 35 min

Problem: Sort a 40 GB newline-delimited file on a machine with 4 GB RAM (e.g. to dedupe by key or prepare a merge join). Write `external_sort(input_path, output_path, max_lines_in_memory, key=None, tmp_dir=None)`: split the input into sorted runs of at most `max_lines_in_memory` lines written to temp files, then k-way merge the runs into the output. The sort must be stable. A final line without a trailing newline must not merge into the next line. All temp files are deleted, even on error. Return the number of lines written.

Signature: `def external_sort(input_path: str, output_path: str, max_lines_in_memory: int, key: Callable[[str], Any] | None = None, tmp_dir: str | None = None) -> int:`

Constraints: UTF-8 text; `key` receives the line *without* its newline. Lines fit in memory individually.

Examples:

- Lines `c, a, b` with `max_lines_in_memory=2` → output `a, b, c`; two runs were created and removed
- An empty file → empty output, returns 0
- `key=lambda l: l.split(",")[0]` with equal keys → original relative order preserved

Hints:

1. "What's the largest thing you're allowed to hold in memory at once?"
2. "Once each run is sorted, how do you combine k sorted streams without loading them?"
3. "How do you guarantee cleanup of k temp files and k open handles when something fails midway?"

Reference solution:

```python
import heapq
import os
import tempfile
from collections.abc import Callable
from contextlib import ExitStack
from itertools import islice
from typing import Any


def external_sort(
    input_path: str,
    output_path: str,
    max_lines_in_memory: int,
    key: Callable[[str], Any] | None = None,
    tmp_dir: str | None = None,
) -> int:
    """Stable external merge sort of a text file using bounded memory.

    Phase 1 writes sorted runs of at most `max_lines_in_memory` lines to temp
    files; phase 2 k-way merges them with heapq.merge (stable across runs, and
    each run is stably sorted). Temp files are always removed.
    """
    if max_lines_in_memory < 1:
        raise ValueError("max_lines_in_memory must be >= 1")
    line_key = None if key is None else (lambda line: key(line.rstrip("\n")))
    run_paths: list[str] = []
    try:
        with open(input_path, encoding="utf-8") as src:
            while chunk := list(islice(src, max_lines_in_memory)):
                chunk = [line if line.endswith("\n") else line + "\n" for line in chunk]
                chunk.sort(key=line_key)
                fd, path = tempfile.mkstemp(suffix=".run", dir=tmp_dir)
                run_paths.append(path)
                with os.fdopen(fd, "w", encoding="utf-8") as run:
                    run.writelines(chunk)
        written = 0
        with ExitStack() as stack, open(output_path, "w", encoding="utf-8") as out:
            runs = [stack.enter_context(open(p, encoding="utf-8")) for p in run_paths]
            for line in heapq.merge(*runs, key=line_key):
                out.write(line)
                written += 1
        return written
    finally:
        for path in run_paths:
            try:
                os.remove(path)
            except FileNotFoundError:
                pass
```

Complexity: O(n log n) comparisons overall (O(n log m) per run of size m, plus O(n log k) for the k-way merge); memory

O(max_lines_in_memory + k); disk I/O is two full passes.

Tests:

```
import random

with tempfile.TemporaryDirectory() as work:
    src, dst, runs = (os.path.join(work, n) for n in ("in.txt", "out.txt", "runs"))
    os.mkdir(runs)

    rng = random.Random(7)
    lines = [f"{rng.randint(0, 999):03d},row{i}" for i in range(1000)]
    with open(src, "w", encoding="utf-8") as f:
        f.write("\n".join(lines))                                  # no trailing newline
    n = external_sort(src, dst, max_lines_in_memory=64, tmp_dir=runs)
    with open(dst, encoding="utf-8") as f:
        out = f.read().splitlines()
    assert n == 1000 and out == sorted(lines)
    assert os.listdir(runs) == []                                   # temp runs cleaned up

    n = external_sort(src, dst, 50, key=lambda l: l.split(",")[0], tmp_dir=runs)
    with open(dst, encoding="utf-8") as f:
        out = f.read().splitlines()
    assert out == sorted(lines, key=lambda l: l.split(",")[0])     # stable by key

    open(src, "w").close()
    assert external_sort(src, dst, 10, tmp_dir=runs) == 0 and open(dst).read() == ""

    with open(src, "w", encoding="utf-8") as f:
        f.write("solo")
    assert external_sort(src, dst, 1, tmp_dir=runs) == 1 and open(dst).read() == "solo\n"

    with open(src, "w", encoding="utf-8") as f:
        f.write("b\na\nc\n")
    def exploding_key(line: str) -> str:
        if line == "c":
            raise RuntimeError("bad key")
        return line
    try:
        external_sort(src, dst, 2, key=exploding_key, tmp_dir=runs)
        raise AssertionError("expected RuntimeError")
    except RuntimeError:
        assert os.listdir(runs) == []                               # cleanup on failure too
```

Alternatives and tradeoffs: GNU `sort -S 2G -T /scratch --parallel=8` does this faster in C and is often the pragmatic answer. DuckDB (`COPY (SELECT * FROM read_csv(...) ORDER BY ...) TO ...`) spills to disk automatically. Spark's `sortBy` is an external sort across a cluster. Pick: `sort` or DuckDB on one machine; this implementation when you need custom keys inside a Python pipeline or can't shell out.

Follow-ups:

- "10,000 runs exceed the open-file limit (`ulimit -n`)." Multi-pass merge: merge runs in groups of, say, 256 into bigger runs, then merge those.
- "Dedupe by key while merging." `itertools.groupby` over the merged stream, since equal keys are now adjacent, keeping the first or last per group.
- "Speed it up." Larger runs (more memory per run), compress runs (gzip trades CPU for I/O), sort runs in parallel processes.

Scoring: Strong: bounded runs, `heapq.merge`, stability argued, newline edge case, guaranteed cleanup (`finally`/`ExitStack`). Acceptable: correct two-phase design with sloppy cleanup. Red flag: `sorted(open(path))`, or merging runs by loading them all back into memory.

Part 3b · HARD & TRICKY (HT)

Deep, subtle behaviour that separates strong candidates from good ones. These are differentiators, not requirements: partial credit with sound reasoning is a strong signal at 3–4 years.

HT-O01 · Cooperative super() across pipeline mixins

Hard & Tricky · Output · All roles · Round 4 · MRO, `super()`, cooperative inheritance · 10 min

Question: What does this print?

```python
class Step:
    def run(self, record):
        print("Step")
        return record

class Validate(Step):
    def run(self, record):
        print("Validate")
        return super().run(record)

class Enrich(Step):
    def run(self, record):
        print("Enrich")
        return super().run(dict(record, region="south"))

class Audit(Step):
    def run(self, record):
        print("Audit")
        return Step.run(self, record)

class Pipeline(Audit, Validate, Enrich):
    pass

print([c.__name__ for c in Pipeline.__mro__])
print(Pipeline().run({"id": 1}))

class OrderPipeline(Validate, Audit, Enrich):
    pass

print(OrderPipeline().run({"id": 2}))
```

Expected output:

```
['Pipeline', 'Audit', 'Validate', 'Enrich', 'Step', 'object']
Audit
Step
{'id': 1}
Validate
Audit
Step
{'id': 2}
```

Trap answer: `Audit, Validate, Enrich, Step` with `{'id': 1, 'region': 'south'}` for the first pipeline (assuming every mixin runs), or `Validate, Step` for the second (assuming `super()` means "my parent class").

Explanation: Python linearises the class hierarchy with C3 into the MRO: each class appears before its bases, and the order of bases in the class statement is preserved. `super()` does not mean "parent class": it means "the next class after the current one *in the MRO of the instance's type*". So `Validate`'s `super()` goes to `Enrich` in one hierarchy and to `Audit` in another, even though `Validate` only inherits from `Step`. Mechanism 1: C3 decides the order. Mechanism 2: `Audit` hard-codes `Step.run(self, ...)` instead of calling `super()`, which jumps straight to `Step` and cuts the chain. In `Pipeline`, `Audit` is first, so `Validate` and `Enrich` never run. In `OrderPipeline`, `Validate` delegates to `Audit`,

which again skips `Enrich`. Neither result has `region`. Fix: every class in a cooperative hierarchy calls `super()`, and all methods share a compatible signature (accept `**kwargs` if needed).

Real-world bug it maps to: A mixin-based pipeline or Django/DRF view stack where one mixin calls its base class directly: a validation, auth, or audit step silently stops executing when someone reorders the bases.

Follow-ups:

1. "Fix `Audit` to call `super()`. What does `Pipeline().run` print now?" Audit, Validate, Enrich, Step, returning `{'id': 1, 'region': 'south'}`.
2. "`class Bad(Step, Validate)`?" TypeError at class creation: "Cannot create a consistent method resolution order (MRO)", because `Step` would have to precede its own subclass.
3. "How does zero-argument `super()` know the class?" The compiler provides a `__class__` cell for methods that use `super`; `super()` combines it with the first argument (`self`).

Scoring: Strong: exact MRO and both runs; explains "next in the MRO" and the chain break. Acceptable: MRO right, misses that `Audit` cuts the chain. Red flag: believes `super()` always calls the direct parent.

HT-O02 · cached_property vs property on a model wrapper

Hard & Tricky · Output · All roles (MLE emphasis) · Round 4 · descriptors, properties, attribute precedence · 10 min

Question: What does this print?

```python
from functools import cached_property

class ModelHandle:
    loads = 0

    def __init__(self, path):
        self.path = path

    @cached_property
    def model(self):
        ModelHandle.loads += 1
        return f"model<{self.path}#{ModelHandle.loads}>"

    @property
    def version(self):
        return self.path.rsplit("-", 1)[-1]

h = ModelHandle("ranker-v3")
print(h.model, h.model, ModelHandle.loads)
print("model" in vars(h))
h.path = "ranker-v4"
print(h.model, h.version)
del h.model
print(h.model)
try:
    h.version = "v9"
except AttributeError as exc:
    print("AttributeError:", exc)
h.__dict__["version"] = "v9"
print(h.version)
```

Expected output:

```
model<ranker-v3#1> model<ranker-v3#1> 1
True
model<ranker-v3#1> v4
model<ranker-v4#2>
AttributeError: property 'version' of 'ModelHandle' object has no setter
v4
```

(3.10 wording: `can't set attribute 'version'`.)

Trap answer: `model<ranker-v4#2>` after changing `path` (expecting the cache to notice), and `v9` on the last line (expecting the instance dict to win).

Explanation: Both decorators are descriptors, but of different kinds. `property` defines `__get__` *and* `__set__`, which makes it a **data descriptor**; data descriptors on the class take precedence over the instance `__dict__`. So even after `h.__dict__["version"] = "v9"`, `h.version` still calls the getter and returns `v4`, and plain assignment fails because there's no setter. `cached_property` defines only `__get__`, a **non-data descriptor**: on first access it computes the value and stores it in the instance `__dict__` under the same name. From then on the instance dict wins over the non-data descriptor, so the function never runs again: that's the cache. It knows nothing about `path`, so the value is stale after `path` changes. `del h.model` removes the dict entry, and the next access recomputes. Takeaway: lookup order is data descriptor → instance dict → non-data descriptor/class attribute; `cached_property` invalidation is manual.

Real-world bug it maps to: A service hot-swaps a model by changing `handle.path`, but `handle.model` keeps serving the old model from `cached_property` until the process restarts; predictions silently come from the wrong version.

Follow-ups:

1. "Thread safety?" In 3.12 `cached_property` no longer takes a lock (the old per-class lock was removed), so two threads can compute the value concurrently; guard expensive loads with your own lock.
2. "Does it work with `__slots__`?" No: it needs an instance `__dict__` (TypeError at first access if there isn't one).
3. "Write a minimal non-data descriptor." A class with `__get__(self, obj, objtype=None)` and `__set_name__(self, owner, name)`; store the result in `obj.__dict__[name]`.

Scoring: Strong: all six lines; explains data vs non-data precedence. Acceptable: understands caching but gets the last line wrong. Red flag: can't explain what a descriptor is after the reveal.

HT-O03 · send/throw/close on a running aggregator

Hard & Tricky · Output · All roles · Round 4 · generator protocol, `send`, `throw`, `close`, `GeneratorExit` · 10 min

Question: What does this print?

```python
def running_total():
    total = 0.0
    try:
        while True:
            try:
                amount = yield total
            except ValueError as exc:
                print(f"skipped: {exc}")
                continue
            if amount is not None:
                total += amount
    finally:
        print(f"flush total={total}")

agg = running_total()
print(next(agg))
print(agg.send(100.0))
print(agg.send(50.5))
print(agg.throw(ValueError("bad amount")))
print(next(agg))
agg.close()
try:
    agg.send(1.0)
except StopIteration:
    print("StopIteration")
```

Expected output:

```
0.0
100.0
150.5
```

```
skipped: bad amount
150.5
150.5
flush total=150.5
StopIteration
```

Trap answer: Treating `throw` as fatal (the generator dies with ValueError), `next(agg)` returning `None`, and `close()` printing nothing.

Explanation: `next()` primes the generator: it runs to the first `yield`, which produces the current `total` (0.0). `send(v)` resumes the paused `yield` expression with value `v`, then runs to the next `yield`, whose value is `send`'s return value. `throw(exc)` raises `exc` *at the paused* `yield`; this generator catches ValueError, loops, and yields again, so `throw` returns 150.5 like any other resume. Plain `next()` is `send(None)`, and the `None` is ignored. `close()` raises `GeneratorExit` at the `yield`; it isn't caught by `except ValueError`, so the `finally` flushes and the generator ends. Any later `send` raises StopIteration. Takeaway: generators are resumable functions with a two-way channel, and `finally` is where they flush state on close.

Real-world bug it maps to: A generator-based batch writer that only flushes when its loop ends naturally loses the final partial batch when the consumer calls `close()` or the generator is garbage-collected; putting the flush in `finally` (or `except GeneratorExit`) fixes it.

Follow-ups:

1. "`agg.send(100.0)` before priming?" `TypeError: can't send non-None value to a just-started generator`.
2. "What if the `finally` block did `yield total`?" On `close()`, `RuntimeError: generator ignored GeneratorExit`. You may not yield after GeneratorExit.
3. "When do you still see this protocol today?" `contextlib.contextmanager` uses `throw`/`close` internally (M-O02); pytest yield-fixtures; streaming aggregators; historically, coroutines before `async`/`await`.

Scoring: Strong: all lines exact; explains priming, `throw` resuming at the `yield`, and `close` → GeneratorExit → finally. Acceptable: `send` semantics right, unsure about `throw`/`close`. Red flag: thinks `send` returns the sent value.

HT-O04 · TaskGroup failure, cancellation and except*

Hard & Tricky · Output · BE, FS, MLE, DE · Round 4 · asyncio cancellation, `TaskGroup`, exception groups (3.11+) · 12 min

Question: What does this print?

```python
import asyncio

async def fetch(name, steps, fail=False):
    try:
        for _ in range(steps):
            await asyncio.sleep(0)
        if fail:
            raise ConnectionError(f"{name} refused")
        print(f"{name} done")
        return name
    except asyncio.CancelledError:
        print(f"{name} cancelled")
        raise

async def main():
    try:
        async with asyncio.TaskGroup() as tg:
            fast = tg.create_task(fetch("inventory", 1))
            tg.create_task(fetch("payments", 2, fail=True))
            tg.create_task(fetch("shipping", 5))
    except* ConnectionError as group:
        print("handled:", [str(e) for e in group.exceptions])
    print(fast.result())
```

```
asyncio.run(main())
```

Expected output:

```
inventory done
shipping cancelled
handled: ['payments refused']
inventory
```

Trap answer: `shipping done` (gather-like "everyone finishes"), or expecting a plain `except ConnectionError:` to be the right handler.

Explanation: A `TaskGroup` provides structured concurrency: when any child task fails, the group cancels all remaining children, waits for them to finish, and then raises an `ExceptionGroup` containing every non-cancellation failure. Trace: every task starts and yields at `sleep(0)`; on the next round `inventory` completes; on the round after that `payments` raises ConnectionError; the group cancels `shipping`, which receives `CancelledError` at its current `await`, prints, and re-raises (correct behaviour); the group then raises `ExceptionGroup([ConnectionError])`. `except*` matches exceptions *inside* the group by type and binds a sub-group. A plain `except ConnectionError:` would **not** match, because the raised object is an `ExceptionGroup`. `fast` completed before the failure, so its result is still available. Takeaway: TaskGroup = all-or-nothing with guaranteed cleanup; handle its errors with `except*`.

Real-world bug it maps to: Someone migrates from `gather` to `TaskGroup` and keeps `except ConnectionError:` around it; the handler never fires and the whole request 500s. Or a task catches `CancelledError` without re-raising, so the group hangs waiting for a task that refuses to die.

Follow-ups:

1. "What if `shipping` swallowed `CancelledError` and returned?" It would stop early but report success; the group still raises the ExceptionGroup, but swallowing cancellation breaks timeouts and shutdown elsewhere. Rule: clean up, then re-raise.
2. "Same code with `asyncio.gather` (default)?" The ConnectionError propagates immediately to the awaiter, but `shipping` keeps running unsupervised in the background, printing `shipping done` later if the loop is still alive.
3. "Two tasks fail with different types?" One ExceptionGroup with both; each `except*` clause handles its matching subset, and unmatched exceptions re-raise as a smaller group.

Scoring: Strong: exact output; explains cancellation of siblings, re-raising CancelledError, and why `except*`. Acceptable: knows siblings get cancelled, unsure about exception groups. Red flag: catches `BaseException`/`CancelledError` broadly "to be safe".

HT-O05 · `__del__` timing and reference cycles (CPython-specific)

Hard & Tricky · Output · All roles · Round 4 · reference counting, cyclic GC, finalizers (CPython-specific) · 10 min

Question: What does this print on CPython?

```python
import gc

closed = []

class Connection:
    def __init__(self, name):
        self.name = name
        self.peer = None

    def __del__(self):
        closed.append(self.name)

gc.disable()
a = Connection("db-a")
alias = a
del a
print(closed)
```

```
del alias
print(closed)
b, c = Connection("db-b"), Connection("db-c")
b.peer, c.peer = c, b
del b, c
print(closed)
gc.collect()
print(sorted(closed))
gc.enable()
```

Expected output:

```
[]
['db-a']
['db-a']
['db-a', 'db-b', 'db-c']
```

Trap answer: `['db-a']` after `del a` ("del destroys the object"), and `['db-a', 'db-b', 'db-c']` right after `del b, c`.

Explanation: CPython frees an object the moment its reference count hits zero. `del a` removes one *name*, not the object; `alias` still refers to it, so nothing is finalised. `del alias` drops the count to zero, and `__del__` runs immediately. `b` and `c` reference each other, so after `del b, c` each still has a count of 1 from the other: reference counting alone can never free a cycle. The cyclic garbage collector finds unreachable cycles; it's disabled here to make timing deterministic, so the cycle survives until `gc.collect()`. Since 3.4 (PEP 442) cycles containing `__del__` are collected normally rather than parked in `gc.garbage`. Finalisation order within a cycle is unspecified, hence `sorted`. On PyPy (no refcounting) even `db-a` would be finalised at some later GC. Takeaway: never rely on `__del__` for releasing resources; use context managers or `weakref.finalize`.

Real-world bug it maps to: Connections or file handles closed in `__del__` "work" in tests but leak in production when an object ends up in a cycle (a parent/child back-reference, a bound-method callback, a traceback holding frames), exhausting the DB connection pool.

Follow-ups:

1. "Name a hidden source of cycles." Storing an exception in a local variable inside `except` (traceback → frame → locals → exception); bound methods registered as callbacks on the object's own attributes.
2. "How do you break the cycle without GC?" `weakref.ref` for back-references (child → parent).
3. "Is it ever right to call `gc.disable()`?" Rarely: some latency-sensitive services disable or tune (`gc.freeze()`, thresholds) to avoid pauses, accepting that cycles must be avoided or collected manually.

Scoring: Strong: all four lines; distinguishes names vs objects, refcount vs cyclic GC. Acceptable: gets the cycle behaviour after a hint about refcounts. Red flag: believes `del x` destroys the object.

HT-O06 · Status-code identity check (CPython-specific)

Hard & Tricky · Output · All roles · Round 4 · object identity, `is` vs `==`, small-int cache (CPython-specific) · 8 min

Question: What does this print on CPython?

```
def is_expected(status, expected):
    return status is expected

ok = 200
not_found = 404
print(is_expected(int("200"), ok), is_expected(int("404"), not_found))
print(int("404") == not_found)

registry = {}
def canonical(code):
    return registry.setdefault(code, code)

first = canonical(int("404"))
```

```
print(canonical(int("404")) is first)
print(None is None, [] is [], ok is not None)
```

Expected output:

```
True False
True
True
True False True
```

Trap answer: `True True` ("same number, same object") or `False False` ("`int()` always makes a new object").

Explanation: `==` compares values via `__eq__`; `is` compares identity, meaning the same object in memory. CPython pre-allocates the integers −5 through 256 and reuses those objects everywhere, so `int("200")` returns the cached object that the constant `200` also refers to: `is` is True. `404` is outside the cache, so `int("404")` builds a fresh object at runtime and `is` is False, although `==` is True. (Values are built with `int()` so constant folding can't merge them.) `setdefault` returns the object already stored in the dict, so a registry can canonicalise equal values into one object. `None` is a singleton, which is why `is None` is the correct test; `[] is []` compares two distinct new lists. Takeaway: use `is` only for singletons (`None`, `True`/`False`, sentinels) and `==` for values; the small-int cache is an implementation detail that makes the bug pass tests with 200 and fail with 404.

Real-world bug it maps to: `if response.status is 200:` passes every happy-path test, but `if status is 404:` or `if retries is max_retries:` (with values above 256) silently never matches in production. CPython 3.8+ emits `SyntaxWarning: "is" with a literal` for the direct form.

Follow-ups:

1. "Strings?" CPython interns some strings (identifier-like literals), so `is` on strings can also "work by accident"; same rule applies.
2. "When is identity the right test for non-singletons?" Sentinel objects (`_MISSING = object()`), cycle detection (`id()`-keyed visited sets), and checking whether a function returned the same cached object.
3. "Does `is` work for `float('nan')`?" `x is x` is True but `x == x` is False for NaN; containers use identity first, so `nan in [nan]` is True for the same object.

Scoring: Strong: exact output; explains the cache as CPython-specific and gives the singleton rule. Acceptable: knows `is` vs `==` but can't explain why 200 works. Red flag: uses `is` for value comparison and defends it.

HT-C01 · Async worker pool with graceful shutdown

Hard & Tricky · Coding · BE, FS, DE · Round 4 · `asyncio.Queue`, cancellation, graceful shutdown · 35 min

Problem: A service consumes jobs (webhook deliveries, emails, enrichment calls) from an in-process queue with N async workers. On deploy it receives SIGTERM and must shut down gracefully: stop accepting new jobs, let queued and in-flight jobs finish for up to `grace` seconds, then cancel whatever is still running. Afterwards the caller must know which jobs completed, failed, were interrupted mid-flight, and were never started. Handler exceptions must not kill a worker. Cancellation must never be swallowed.

Signature:

```
class WorkerPool:
    def __init__(self, handler: Callable[[Any], Awaitable[Any]], workers: int = 4) -> None
    def start(self) -> None
    def submit(self, job: Any) -> None            # RuntimeError once shutdown has begun
    async def shutdown(self, grace: float) -> list[Any]   # returns never-started jobs
    # attributes: results, errors (list of (job, exc)), interrupted
```

Constraints: One event loop. Standard library only.

Examples:

- 10 quick jobs, 3 workers, `shutdown(grace=1)` → 10 results, nothing interrupted or left over
- A handler that raises for one job → that `(job, exc)` in `errors`; workers keep going

- Jobs that block forever, 2 workers, 5 jobs, `grace=0.05` → 2 interrupted, 3 returned as never started

Trap answer: Workers wrapped in `except BaseException:` (or bare `except:`) "so they never die". That catches `CancelledError`, so `task.cancel()` does nothing and shutdown hangs forever. The second trap is cancelling workers immediately and losing queued jobs silently.

Real-world bug it maps to: Rolling deploys either hang until Kubernetes SIGKILLs the pod (cancellation swallowed) or drop in-flight webhooks (no drain), and nobody knows which jobs were lost.

Hints:

1. "What does `queue.join()` wait for, and how do `task_done()` calls feed it?"
2. "How do you bound the drain phase in time?"
3. "Where can a worker record that it was interrupted, without swallowing the cancellation?"

Reference solution:

```python
import asyncio
from collections.abc import Awaitable, Callable
from typing import Any


class WorkerPool:
    """Fixed-size pool of asyncio workers with bounded graceful shutdown."""

    def __init__(self, handler: Callable[[Any], Awaitable[Any]], workers: int = 4) -> None:
        if workers < 1:
            raise ValueError("workers must be >= 1")
        self._handler = handler
        self._size = workers
        self._queue: asyncio.Queue[Any] = asyncio.Queue()
        self._tasks: list[asyncio.Task[None]] = []
        self._closing = False
        self.results: list[Any] = []
        self.errors: list[tuple[Any, Exception]] = []
        self.interrupted: list[Any] = []

    def start(self) -> None:
        self._tasks = [asyncio.create_task(self._worker()) for _ in range(self._size)]

    def submit(self, job: Any) -> None:
        if self._closing:
            raise RuntimeError("pool is shutting down")
        self._queue.put_nowait(job)

    async def _worker(self) -> None:
        while True:
            job = await self._queue.get()
            try:
                self.results.append(await self._handler(job))
            except asyncio.CancelledError:
                self.interrupted.append(job)   # record, then let cancellation propagate
                raise
            except Exception as exc:           # handler failure: keep the worker alive
                self.errors.append((job, exc))
            finally:
                self._queue.task_done()

    async def shutdown(self, grace: float) -> list[Any]:
        """Stop intake, drain for up to `grace` seconds, cancel the rest.

        Returns jobs that were still queued (never started).
        """
        self._closing = True
        try:
            await asyncio.wait_for(self._queue.join(), timeout=grace)
        except TimeoutError:
```

```
            pass
        for task in self._tasks:
            task.cancel()
        await asyncio.gather(*self._tasks, return_exceptions=True)
        never_started = []
        while not self._queue.empty():
            never_started.append(self._queue.get_nowait())
            self._queue.task_done()
        return never_started
```

Complexity: O(jobs) total work; O(queued jobs) memory; shutdown takes at most `grace` plus the time for cancelled handlers to unwind.

Tests:

```
async def scenario() -> None:
    async def quick(job: int) -> int:
        await asyncio.sleep(0)
        if job == 4:
            raise ValueError("bad payload")
        return job * 10

    pool = WorkerPool(quick, workers=3)
    pool.start()
    for j in range(10):
        pool.submit(j)
    leftover = await pool.shutdown(grace=1)
    assert leftover == [] and pool.interrupted == []
    assert sorted(pool.results) == [j * 10 for j in range(10) if j != 4]
    assert [(job, type(e)) for job, e in pool.errors] == [(4, ValueError)]
    assert all(t.done() for t in pool._tasks)
    try:
        pool.submit(99)
        raise AssertionError("expected RuntimeError")
    except RuntimeError:
        pass

    async def stuck(job: int) -> None:
        await asyncio.Event().wait()                    # never finishes

    pool = WorkerPool(stuck, workers=2)
    pool.start()
    for j in range(5):
        pool.submit(j)
    leftover = await pool.shutdown(grace=0.05)
    assert sorted(pool.interrupted) == [0, 1] and leftover == [2, 3, 4]
    assert all(t.cancelled() for t in pool._tasks)       # cancellation not swallowed

    empty = WorkerPool(quick, workers=1)
    empty.start()
    assert await empty.shutdown(grace=0.1) == []

    try:
        WorkerPool(quick, workers=0)
        raise AssertionError("expected ValueError")
    except ValueError:
        pass

asyncio.run(scenario())
```

Alternatives and tradeoffs: 3.13 adds `asyncio.Queue.shutdown()`, which makes `put` raise `QueueShutDown` and wakes blocked getters, replacing the `_closing` flag and making "stop intake" race-free. `asyncio.TaskGroup` supervises workers but cancels all of them on the first unhandled error, which is wrong for a pool that must survive handler failures unless each handler catches its own errors. Out-of-process queues (Celery, RQ, Arq, SQS) survive crashes and redeliver unacknowledged jobs, which in-process queues cannot. Pick: this pattern for in-process background work;

a durable broker when losing a job is unacceptable.

Follow-ups:

- "Wire it to SIGTERM." `loop.add_signal_handler(signal.SIGTERM, lambda: asyncio.create_task(pool.shutdown(25)))`, with `grace` smaller than the orchestrator's termination grace period (Kubernetes default 30 s).
- "Interrupted jobs must not be lost." Persist them (or the never-started list) to a durable queue before exiting, and make handlers idempotent because a job may have partially run.
- "Backpressure?" `asyncio.Queue(maxsize=...)` with `await queue.put()` so producers slow down instead of buffering unboundedly.

Scoring: Strong: bounded drain with `join` + timeout, cancel + gather, re-raises CancelledError, accounts for every job. Acceptable: correct drain, weak accounting of interrupted vs never-started. Red flag: `except BaseException` in the worker, or `shutdown` that just cancels immediately.

HT-C02 · Nested transactions with savepoints

Hard & Tricky · Coding · BE, DE, FS · Round 4 · context managers, exceptions, `sqlite3` transactions · 30 min

Problem: Service code calls helpers that each want "a transaction", and helpers call each other. Write a `TransactionManager` whose `transaction()` context manager opens a real transaction at the outermost level and a `SAVEPOINT` when nested. On a clean exit, commit (outer) or release the savepoint (inner). On an exception, roll back only that level and **re-raise**. If a caller catches an inner failure, the outer transaction must still be able to commit its other work.

Signature:

```
class TransactionManager:
    def __init__(self, conn: sqlite3.Connection) -> None
    @contextmanager
    def transaction(self) -> Iterator[sqlite3.Connection]
```

Constraints: Use `sqlite3` from the standard library with explicit transaction control. Savepoint names must be unique per nesting level.

Examples:

- Outer inserts A; inner inserts B and raises; caller catches; outer inserts C → table has A, C
- Outer raises after inner succeeded → table empty
- Three levels, all clean → everything committed

Trap answer: Letting every level issue `BEGIN` (SQLite raises "cannot start a transaction within a transaction"), or a nested block that swallows its exception after rolling back, so callers think the helper succeeded. Also: `ROLLBACK TO SAVEPOINT x` *without* `RELEASE`, which leaves the savepoint on the stack.

Real-world bug it maps to: An "audit log write" helper fails inside a larger order transaction and rolls back the entire order (no savepoints); or it swallows its error and the order commits without its audit row.

Hints:

1. "How does the manager know whether it's the outermost level?"
2. "What does `ROLLBACK TO SAVEPOINT` leave behind?"
3. "Why must the context manager re-raise after rolling back?"

Reference solution:

```
import sqlite3
from collections.abc import Iterator
from contextlib import contextmanager


class TransactionManager:
```

```
    """Nestable transactions: BEGIN at depth 0, SAVEPOINTs below it."""

    def __init__(self, conn: sqlite3.Connection) -> None:
        conn.isolation_level = None      # we issue BEGIN/COMMIT ourselves
        self._conn = conn
        self._depth = 0
        self._counter = 0

    @contextmanager
    def transaction(self) -> Iterator[sqlite3.Connection]:
        outermost = self._depth == 0
        if outermost:
            self._conn.execute("BEGIN")
        else:
            self._counter += 1
            savepoint = f"sp_{self._counter}"
            self._conn.execute(f"SAVEPOINT {savepoint}")
        self._depth += 1
        try:
            yield self._conn
        except BaseException:
            self._depth -= 1
            if outermost:
                self._conn.execute("ROLLBACK")
            else:
                self._conn.execute(f"ROLLBACK TO SAVEPOINT {savepoint}")
                self._conn.execute(f"RELEASE SAVEPOINT {savepoint}")
            raise
        else:
            self._depth -= 1
            if outermost:
                self._conn.execute("COMMIT")
            else:
                self._conn.execute(f"RELEASE SAVEPOINT {savepoint}")
```

Complexity: O(1) extra statements per level; O(nesting depth) state.

Tests:

```
def fresh() -> tuple[sqlite3.Connection, TransactionManager]:
    conn = sqlite3.connect(":memory:")
    conn.execute("CREATE TABLE audit (event TEXT NOT NULL)")
    return conn, TransactionManager(conn)

def events(conn: sqlite3.Connection) -> list[str]:
    return [r[0] for r in conn.execute("SELECT event FROM audit ORDER BY rowid")]

conn, tm = fresh()
propagated = []
with tm.transaction() as c:
    c.execute("INSERT INTO audit VALUES ('A')")
    try:
        with tm.transaction() as inner:
            inner.execute("INSERT INTO audit VALUES ('B')")
            raise LookupError("helper failed")
    except LookupError:
        propagated.append("inner")               # re-raised; the caller chose to continue
    c.execute("INSERT INTO audit VALUES ('C')")
assert propagated == ["inner"]
assert events(conn) == ["A", "C"] and not conn.in_transaction

conn, tm = fresh()
try:
    with tm.transaction() as c:
        with tm.transaction() as inner:
            inner.execute("INSERT INTO audit VALUES ('B')")
        raise RuntimeError("outer failed")
```

```
except RuntimeError:
    propagated.append("outer")
assert propagated == ["inner", "outer"]
assert events(conn) == [] and not conn.in_transaction

conn, tm = fresh()
with tm.transaction() as c1:
    with tm.transaction() as c2:
        with tm.transaction() as c3:
            c3.execute("INSERT INTO audit VALUES ('deep')")
assert events(conn) == ["deep"]

conn, tm = fresh()
try:
    with tm.transaction() as c:
        c.execute("INSERT INTO audit VALUES (NULL)")   # constraint violation
except sqlite3.IntegrityError:
    pass
with tm.transaction() as c:                             # depth reset: usable again
    c.execute("INSERT INTO audit VALUES ('after')")
assert events(conn) == ["after"]
```

Alternatives and tradeoffs: SQLAlchemy's `session.begin_nested()` and Django's nested `transaction.atomic()` implement exactly this and are what you'd use in those stacks. Python 3.12's `sqlite3.Connection.autocommit` attribute (PEP 249-compliant control) is the modern alternative to `isolation_level=None`. Pick: the framework's nested-transaction API when you have one; this pattern for raw drivers.

Follow-ups:

- "Make it safe across threads." Connections are not meant to be shared; give each thread its own connection and manager (e.g. `threading.local`), or depth tracking breaks.
- "Postgres?" Same statements; also consider that a failed statement aborts the whole Postgres transaction unless you roll back to a savepoint, which is why ORMs wrap risky statements in savepoints.
- "Async?" Same shape with `@asynccontextmanager` and an async driver; the depth counter must be per-connection, not per-event-loop.

Scoring: Strong: BEGIN vs SAVEPOINT by depth, ROLLBACK TO + RELEASE, re-raises, depth reset on both paths. Acceptable: works but forgets RELEASE after rollback, or doesn't reset depth on error. Red flag: swallows exceptions inside the context manager.

HT-C03 · Per-instance memoization that doesn't leak

Hard & Tricky · Coding · MLE, BE, DE · Round 4 · memory, weak references, GC, caching · 30 min

Problem: A feature extractor class has an expensive method `embed(self, text)`. Decorating it with `functools.lru_cache` caused a memory leak: the process creates thousands of short-lived extractor instances per hour and none are ever freed. Write a method decorator `instance_memo` that caches per instance, never keeps an instance alive, frees an instance's cache when the instance is garbage-collected, and keeps caches of *equal-but-distinct* instances separate (instances may define `__eq__` and even be unhashable). Expose `live_caches()` for monitoring.

Signature: `def instance_memo(method: Callable[Concatenate[S, P], R]) -> Callable[Concatenate[S, P], R]:`

Constraints: Method arguments are hashable. Instances support weak references (no `__slots__` without `__weakref__`). Single-threaded use.

Examples:

- `x.embed("a")` twice → computed once
- `x` and `y` compare equal, `x.embed("a")` and `y.embed("a")` → computed twice
- `del x` → `live_caches()` drops by one

Trap answer: `@functools.lru_cache` on the method: the cache key includes `self`, so the cache holds a strong

reference to every instance until evicted (never, with `maxsize=None`). The second trap is `WeakKeyDictionary[self]`, which fails for unhashable instances and merges caches of equal instances, because it uses `__hash__`/`__eq__`.

Real-world bug it maps to: A long-running inference or ETL worker grows by gigabytes a day because each request-scoped object is pinned by a class-level method cache; restarts become the "fix".

Hints:

1. "What exactly is in the key that `lru_cache` builds for a method call?"
2. "How can you run code when an object is garbage-collected without holding a strong reference to it?"
3. "Why might `WeakKeyDictionary` merge two different instances?"

Reference solution:

```python
import functools
import weakref
from collections.abc import Callable
from typing import Any, Concatenate, ParamSpec, TypeVar

S = TypeVar("S")
P = ParamSpec("P")
R = TypeVar("R")


def instance_memo(method: Callable[Concatenate[S, P], R]) -> Callable[Concatenate[S, P], R]:
    """Memoize a method per instance without keeping instances alive.

    Caches are keyed by id(instance); weakref.finalize drops the cache when
    the instance is collected (before its id can be reused). Equal but
    distinct instances get separate caches; unhashable instances work.
    """
    caches: dict[int, dict[Any, R]] = {}

    @functools.wraps(method)
    def wrapper(self: S, *args: P.args, **kwargs: P.kwargs) -> R:
        cache = caches.get(id(self))
        if cache is None:
            cache = caches[id(self)] = {}
            weakref.finalize(self, caches.pop, id(self), None)   # no strong ref to self
        key = (args, frozenset(kwargs.items()))
        if key not in cache:
            cache[key] = method(self, *args, **kwargs)
        return cache[key]

    wrapper.live_caches = lambda: len(caches)                    # type: ignore[attr-defined]
    return wrapper
```

Complexity: O(1) average per call beyond key construction; memory proportional to live instances × their distinct calls, released when instances die.

Tests:

```python
import gc

computed: list[tuple[str, str]] = []

class Extractor:
    __hash__ = None                                  # unhashable on purpose

    def __init__(self, name: str) -> None:
        self.name = name

    def __eq__(self, other: object) -> bool:
        return isinstance(other, Extractor)          # all extractors "equal"

    @instance_memo
    def embed(self, text: str, *, dim: int = 4) -> tuple[str, int]:
```

```
        computed.append((self.name, text))
        return (text.upper(), dim)

x, y = Extractor("x"), Extractor("y")
assert x.embed("a") == ("A", 4) and x.embed("a") == ("A", 4)
assert computed == [("x", "a")]
y.embed("a")
assert computed == [("x", "a"), ("y", "a")]           # equal instances, separate caches
x.embed(text="a", dim=8); x.embed(dim=8, text="a")
assert computed.count(("x", "a")) == 2                 # kwargs order irrelevant
assert Extractor.embed.live_caches() == 2

probe = weakref.ref(x)
del x
gc.collect()
assert probe() is None and Extractor.embed.live_caches() == 1   # freed, cache dropped
del y
gc.collect()
assert Extractor.embed.live_caches() == 0

class Leaky:
    @functools.lru_cache(maxsize=None)
    def embed(self, text: str) -> str:
        return text

leaky = Leaky()
leaky.embed("a")
leaky_probe = weakref.ref(leaky)
del leaky
gc.collect()
assert leaky_probe() is not None                       # lru_cache keeps the instance alive
assert Extractor.embed.__name__ == "embed"
```

Alternatives and tradeoffs: Store the cache on the instance (`self.__dict__["_embed_cache"]`, as `cached_property` does): it dies with the instance, needs no global dict, and is the simplest option when instances have a `__dict__`. `weakref.WeakKeyDictionary` is fine when instances hash by identity. `lru_cache` on a module-level function taking only the data (not `self`) works when the result doesn't depend on instance state. Pick: an instance attribute cache when possible; this id + `finalize` design when you can't touch the instance's attributes (slots with `__weakref__`, third-party classes).

Follow-ups:

- "Why is `id()` reuse not a problem here?" `finalize` callbacks run during the object's deallocation, before its memory (and therefore its id) can be reused.
- "Are `x.embed("a")` and `x.embed(text="a")` the same entry?" No: positional and keyword spellings produce different keys (as with `lru_cache`). Normalise with `inspect.signature(method).bind(...)` if that matters.
- "Bound the per-instance cache size." Replace the inner dict with an `OrderedDict` LRU (MT-C01 / H-C02 pattern).
- "What if the cached value references `self`?" It creates a cycle (cache → value → self), freed only by the cyclic GC, and `finalize` then runs at GC time rather than immediately. Avoid storing `self`-referencing values.

Scoring: Strong: explains the `lru_cache` leak mechanism, uses weak references correctly, handles equal/unhashable instances, proves collection in a test. Acceptable: `WeakKeyDictionary` solution and knows its limits when asked. Red flag: "just call `cache_clear()` periodically".

Part 4 · Role-wise section

Each role gets a calibration note, the master-bank questions an interviewer must ask, four role-exclusive questions written in full, and role-specific red flags. Role-exclusive questions may use the role's libraries. Versions used for verification: pandas 2.2.3 (baseline) and pandas 3.0.2, NumPy 2.x, Pydantic 2.13, FastAPI 0.14x, Jinja2 3.1.

DE · Data Engineer

Calibration (3–4 years): Writes memory-safe pipelines, thinks about reruns and partial failures before being asked, and never silently drops rows. Should reason about Spark's execution model (lazy evaluation, shuffles, skew) without needing a cluster, and treat timestamps as UTC with explicit offsets.

Must-ask master-bank IDs: ET-O01 (money rounding), ET-O03 (iterator exhaustion), ET-O05 (shared defaults), E-C01 (batching), MT-O04 (mutation during iteration), MT-O05 (groupby), MT-C02 (chunked decoding), M-C03 (task DAG), H-C03 (external sort), H-O01 (threads for I/O).

DE-O01 · Comparing naive and aware timestamps

Medium & Tricky · Output · DE · Round 3 · datetime, timezones, DST · 7 min

Question: What does this print? (IST is UTC+05:30; US daylight saving starts on 8 March 2026.)

```python
from datetime import datetime, timedelta, timezone
from zoneinfo import ZoneInfo

event = datetime.fromisoformat("2026-03-08T01:30:00+05:30")
cutoff = datetime(2026, 3, 8, 1, 0)
print(event == cutoff)
try:
    print(event > cutoff)
except TypeError as exc:
    print("TypeError:", exc)
print(event.astimezone(timezone.utc).isoformat())

ny = ZoneInfo("America/New_York")
start = datetime(2026, 3, 7, 12, 0, tzinfo=ny)
end = start + timedelta(days=1)
print(end.isoformat(), end - start)
print(end.astimezone(timezone.utc) - start.astimezone(timezone.utc))
```

Expected output:

```
False
TypeError: can't compare offset-naive and offset-aware datetimes
2026-03-07T20:00:00+00:00
2026-03-08T12:00:00-04:00 1 day, 0:00:00
23:00:00
```

Trap answer: `True/False` for the comparisons without error, and `1 day, 0:00:00` for both differences. Tempting because both datetimes "look like" 8 March.

Explanation: A naive datetime has no offset: it could be any timezone, so Python refuses to *order* it against an aware one (TypeError), but `==` between naive and aware simply returns False without raising: the silent half of the same problem. `astimezone(timezone.utc)` converts the instant: 01:30 IST is 20:00 UTC on the previous day. Arithmetic between two aware datetimes sharing the *same* `tzinfo` object is wall-clock arithmetic: adding one day gives 12:00 the next day, now in EDT (−04:00), and subtracting them reports 1 day. Converting both to UTC first reveals the real elapsed time: 23 hours, because the clocks sprang forward. Takeaway: store and compute in UTC, convert to local time only for display, and reject naive timestamps at ingestion.

Real-world bug it maps to: A daily incremental load filters `event_time > last_watermark` with a naive watermark: either the job crashes, or with `==`-based dedup it silently double-loads. Daily aggregates across DST boundaries report 23- or 25-hour "days" as 24.

Follow-ups:

1. "`datetime.utcnow()`?" Returns a *naive* datetime in UTC and is deprecated since 3.12; use `datetime.now(timezone.utc)`.
2. "Parsing `2026-09-24T10:00:00Z`?" `fromisoformat` accepts the `Z` suffix since 3.11; on 3.10 it raises ValueError.

3. "In Spark?" `spark.sql.session.timeZone` controls how timestamps render and parse; set it to UTC for pipelines.

Scoring: Strong: all five lines, including the silent `==` and the 23 hours. Acceptable: gets the TypeError and UTC conversion, misses the DST subtlety. Red flag: stores local times without offsets and sees no problem.

DE-O02 · csv.DictReader on malformed rows

Medium · Output · DE · Round 3 · CSV quoting, ragged rows · 6 min

Question: What does this print?

```
import csv
import io

raw = io.StringIO(
    'order_id,customer,amount\n'
    '1,"Rao, Priya",120.50\n'
    '2,"Line one\nline two",80\n'
    '3,Mehta\n'
    '4,Iyer,99,EXTRA\n'
)
reader = csv.DictReader(raw)
for row in reader:
    print(row)
print(reader.line_num)
```

Expected output:

```
{'order_id': '1', 'customer': 'Rao, Priya', 'amount': '120.50'}
{'order_id': '2', 'customer': 'Line one\nline two', 'amount': '80'}
{'order_id': '3', 'customer': 'Mehta', 'amount': None}
{'order_id': '4', 'customer': 'Iyer', 'amount': '99', None: ['EXTRA']}
6
```

Explanation: The csv module implements quoting, so a comma inside quotes is data and a quoted newline is part of the field; a naive `line.split(",")` breaks on both. Every value is a string: no type inference. Short rows are padded with `restval` (default `None`) rather than rejected. Extra fields are collected into a list under the `restkey` key (default `None`) rather than rejected. Neither raises, so malformed rows flow silently downstream unless you check. `line_num` counts *physical source lines* read (6: header + 5 lines, because record 2 spans two lines), not records. When reading real files, open them with `newline=""` so the csv module sees the raw line endings inside quoted fields. Takeaway: validate row shape explicitly (`None in row` or `None in row.values()`), route bad rows to a quarantine table, and never assume `line_num == record number`.

Follow-ups:

1. "How do you make ragged rows fail loudly?" Check `None in row` (extras) and `None in row.values()` (missing) per row and raise or dead-letter with the line number.
2. "`pandas.read_csv` on this file?" Row 4 raises a tokenizing error by default; `on_bad_lines="warn"/"skip"` changes that, and "skip" silently drops data.
3. "Encoding surprises?" Excel exports often use `utf-8-sig` (BOM) or cp1252; a BOM glued to the first header makes `row["order_id"]` a KeyError.

Scoring: Strong: all five lines, including `None: ['EXTRA']` and `line_num`. Acceptable: quoting right, unsure about ragged rows. Red flag: parses CSV with `split(",")` in their own answer.

DE-C01 · SCD Type 2 merge for a customer dimension

Hard · Coding · DE · Round 3 · SCD2, idempotent reruns, validation · 35 min

Problem: Maintain a Type 2 slowly changing dimension for customers. Tracked attributes: `name`, `city`. Given the current dimension rows and a full snapshot from the source for an effective date:

- new customers get a current row;
- changed customers get their current row closed (`valid_to = effective`) and a new current row;
- unchanged and absent customers are left alone (no deletes).

The merge must be **idempotent**: re-running the same snapshot for the same date changes nothing. It must be atomic: on any validation error, raise without partial output. Reject snapshots with duplicate customers, dimensions with two current rows for one customer, and changes whose effective date isn't after the current version's `valid_from`. Return the new dimension sorted by `(customer_id, valid_from)` plus counts.

Signature: `def scd2_merge(dimension: list[DimRow], snapshot: list[dict[str, str]], effective: date) -> tuple[list[DimRow], dict[str, int]]:`

Constraints: Up to 10^6 dimension rows (fits in memory for this exercise). `DimRow` is a frozen dataclass; `valid_to is None` means current.

Examples:

- Empty dimension + snapshot of 2 customers → 2 inserted
- c1 moved city → old row closed on the effective date, new row current; counts `updated=1`
- Same snapshot, same date, applied twice → second run reports only `unchanged`

Hints:

1. "Index the current rows by customer first."
2. "What makes a rerun a no-op?"
3. "Where can you detect every error before producing any output?"

Reference solution:

```python
from dataclasses import dataclass, replace
from datetime import date

TRACKED = ("name", "city")


@dataclass(frozen=True)
class DimRow:
    customer_id: str
    name: str
    city: str
    valid_from: date
    valid_to: date | None = None          # None = current version

    @property
    def is_current(self) -> bool:
        return self.valid_to is None


def scd2_merge(
    dimension: list[DimRow], snapshot: list[dict[str, str]], effective: date
) -> tuple[list[DimRow], dict[str, int]]:
    """Apply a full source snapshot to an SCD Type 2 dimension.

    Pure function: inputs are not mutated; on any validation error nothing is
    returned. Re-running the same snapshot for the same date is a no-op.
    """
    current: dict[str, DimRow] = {}
    history: list[DimRow] = []
    for row in dimension:
        if not row.is_current:
            history.append(row)
        elif row.customer_id in current:
            raise ValueError(f"two current rows for {row.customer_id}")
        else:
            current[row.customer_id] = row
```

```
    stats = {"inserted": 0, "updated": 0, "unchanged": 0}
    seen: set[str] = set()
    for record in snapshot:
        cid = record["customer_id"]
        if cid in seen:
            raise ValueError(f"duplicate customer in snapshot: {cid}")
        seen.add(cid)
        existing = current.get(cid)
        if existing is None:
            current[cid] = DimRow(cid, record["name"], record["city"], effective)
            stats["inserted"] += 1
        elif all(getattr(existing, f) == record[f] for f in TRACKED):
            stats["unchanged"] += 1
        else:
            if effective <= existing.valid_from:
                raise ValueError(f"{cid}: effective {effective} is not after
{existing.valid_from}")
            history.append(replace(existing, valid_to=effective))
            current[cid] = DimRow(cid, record["name"], record["city"], effective)
            stats["updated"] += 1

    rows = history + list(current.values())
    rows.sort(key=lambda r: (r.customer_id, r.valid_from))
    return rows, stats
```

Complexity: O(n log n) for n dimension rows (the final sort), O(s) for s snapshot records; O(n + s) space.

Tests:

```
d1, d2, d3 = date(2026, 1, 1), date(2026, 2, 1), date(2026, 3, 1)
snap1 = [{"customer_id": "c1", "name": "Asha", "city": "Pune"},
         {"customer_id": "c2", "name": "Ravi", "city": "Hyderabad"}]
dim, stats = scd2_merge([], snap1, d1)
assert stats == {"inserted": 2, "updated": 0, "unchanged": 0} and all(r.is_current for r in
dim)

snap2 = [{"customer_id": "c1", "name": "Asha", "city": "Bengaluru"},
         {"customer_id": "c3", "name": "Meera", "city": "Chennai"}]      # c2 absent: untouched
dim2, stats = scd2_merge(dim, snap2, d2)
assert stats == {"inserted": 1, "updated": 1, "unchanged": 0}
assert [(r.customer_id, r.city, r.valid_from, r.valid_to) for r in dim2] == [
    ("c1", "Pune", d1, d2), ("c1", "Bengaluru", d2, None),
    ("c2", "Hyderabad", d1, None), ("c3", "Chennai", d2, None)]

again, stats = scd2_merge(dim2, snap2, d2)                              # idempotent rerun
assert again == dim2 and stats == {"inserted": 0, "updated": 0, "unchanged": 2}

assert scd2_merge([], [], d1) == ([], {"inserted": 0, "updated": 0, "unchanged": 0})

bad_inputs = [
    (dim2, snap2 + [snap2[0]], d3),                                      # duplicate in
snapshot
    (dim2 + [DimRow("c3", "Meera", "Delhi", d1)], [], d3),               # two current rows
    (dim2, [{"customer_id": "c1", "name": "Asha", "city": "Goa"}], d1),  # effective in the
past
]
for dimension, snapshot, eff in bad_inputs:
    try:
        scd2_merge(dimension, snapshot, eff)
        raise AssertionError("expected ValueError")
    except ValueError:
        pass
assert len(dim2) == 4                                                    # inputs never mutated
```

Alternatives and tradeoffs: In a warehouse or lakehouse this is a `MERGE INTO` (Delta Lake, Snowflake, BigQuery): one statement to close changed rows and one to insert new versions, or dbt snapshots, which implement exactly this

pattern. Comparing a hash of tracked columns (`md5(concat_ws('|', ...))`) instead of column-by-column is cheaper for wide dimensions, but beware NULL handling in the concatenation. Pick: `MERGE`/dbt snapshots in production; this pure function for understanding and for unit-testing the logic.

Follow-ups:

- "The source sends deletes." Close the current row (and optionally add an `is_deleted` flag) for customers absent from a *full* snapshot; never infer deletes from an incremental feed.
- "A correction arrives for a past date (late-arriving data)." Either reject (as here) or insert the version into history and re-close adjacent versions: much harder, which is why many teams only support forward changes.
- "Do this in Spark on 500 M rows." Join snapshot to current rows on `customer_id` (shuffle), compute change flags, union closed + new + untouched; broadcast the snapshot if it's small; watch skew on hot keys.

Scoring: Strong: idempotent by construction, validates before emitting, handles absent customers, sorts deterministically, mentions `MERGE`/dbt. Acceptable: correct inserts/updates, needs prompting on idempotency or validation. Red flag: mutates input rows in place, or a rerun creates duplicate versions.

DE-C02 · Sessionise clickstream events

Medium · Coding · DE · Round 3 · sessionisation, windowing, timezone-aware parsing · 20 min

Problem: Clickstream events arrive as `(user_id, iso_timestamp)` pairs from several regions, unsorted, with explicit offsets (`Z`, `+05:30`, `-04:00`). A session ends when a user is inactive for *more than* 30 minutes. Return sessions as `(user_id, start_utc_iso, end_utc_iso, event_count)` sorted by user then start. Naive timestamps must be rejected with `ValueError`.

Signature: `def sessionise(events: list[tuple[str, str]], gap: timedelta = timedelta(minutes=30)) -> list[tuple[str, str, str, int]]:`

Constraints: Up to 10^6 events, fits in memory. A gap of exactly 30 minutes stays in the same session.

Examples:

- u1 at 10:00Z, 10:20Z, 11:00Z → two sessions: (10:00–10:20, 2 events) and (11:00–11:00, 1)
- u1 at `2026-09-24T15:30:00+05:30` (10:00Z) and `2026-09-24T10:30:00Z` → one session of 2 events
- A naive timestamp → ValueError

Hints:

1. "Normalise every timestamp to UTC first. Why?"
2. "Group per user, then sort within the user."
3. "Compare each event with the *previous* event, not the session start."

Reference solution:

```python
from collections import defaultdict
from datetime import datetime, timedelta, timezone


def sessionise(
    events: list[tuple[str, str]], gap: timedelta = timedelta(minutes=30)
) -> list[tuple[str, str, str, int]]:
    """Split each user's events into sessions separated by inactivity > gap.

    Timestamps must carry an offset; all output is in UTC ISO 8601.
    """
    per_user: dict[str, list[datetime]] = defaultdict(list)
    for user_id, raw in events:
        ts = datetime.fromisoformat(raw)                  # 'Z' suffix supported on 3.11+
        if ts.tzinfo is None:
            raise ValueError(f"naive timestamp for {user_id}: {raw!r}")
        per_user[user_id].append(ts.astimezone(timezone.utc))
```

```
    sessions: list[tuple[str, str, str, int]] = []
    for user_id in sorted(per_user):
        times = sorted(per_user[user_id])
        start = prev = times[0]
        count = 0
        for ts in times:
            if ts - prev > gap:
                sessions.append((user_id, start.isoformat(), prev.isoformat(), count))
                start, count = ts, 0
            count += 1
            prev = ts
        sessions.append((user_id, start.isoformat(), prev.isoformat(), count))
    return sessions
```

Complexity: O(n log n) for sorting within users; O(n) space.

Tests:

```
events = [
    ("u1", "2026-09-24T11:00:00Z"),
    ("u2", "2026-09-24T09:00:00-04:00"),                  # 13:00Z
    ("u1", "2026-09-24T15:30:00+05:30"),                  # 10:00Z
    ("u1", "2026-09-24T10:20:00Z"),
    ("u1", "2026-09-24T11:30:00Z"),                       # exactly 30 min after 11:00: same
session
    ("u1", "2026-09-24T12:00:01Z"),                       # 30 min 1 s: new session
]
assert sessionise(events) == [
    ("u1", "2026-09-24T10:00:00+00:00", "2026-09-24T10:20:00+00:00", 2),
    ("u1", "2026-09-24T11:00:00+00:00", "2026-09-24T11:30:00+00:00", 2),
    ("u1", "2026-09-24T12:00:01+00:00", "2026-09-24T12:00:01+00:00", 1),
    ("u2", "2026-09-24T13:00:00+00:00", "2026-09-24T13:00:00+00:00", 1),
]
assert sessionise([]) == []
assert sessionise([("u9", "2026-01-01T00:00:00Z")]) == [
    ("u9", "2026-01-01T00:00:00+00:00", "2026-01-01T00:00:00+00:00", 1)]
dupes = [("u1", "2026-01-01T00:00:00Z")] * 3                          # duplicate events
assert sessionise(dupes)[0][3] == 3
try:
    sessionise([("u1", "2026-09-24T10:00:00")])
    raise AssertionError("expected ValueError")
except ValueError:
    pass
```

Alternatives and tradeoffs: pandas: sort by user and time, `diff()` within `groupby`, `(diff > gap).cumsum()` as a session id, then aggregate. Vectorised, but the whole frame must fit in memory. Spark/SQL: `LAG(ts) OVER (PARTITION BY user ORDER BY ts)`, flag gaps, running `SUM` of flags as session id; Spark 3.2+ also has `session_window`. Streaming (Flink, Spark Structured Streaming): session windows with watermarks for late events. Pick: SQL/Spark window functions at scale; this pure-Python version for small data or tests.

Follow-ups:

- "Events arrive as an unbounded stream." Keep per-user open sessions; close one when watermark − last_event > gap; late events beyond the watermark go to a correction path.
- "Duplicate events (same user, same timestamp)?" Decide explicitly: dedupe on an event ID before sessionising, or count duplicates (as here).
- "One user has 40% of all events (bot)." Skew: in Spark that partition becomes a straggler; salt the key or filter bots upstream.

Scoring: Strong: UTC normalisation, per-user sort, gap vs previous event, boundary (exactly 30 min) handled, naive rejected. Acceptable: correct but compares to session start or needs the UTC hint. Red flag: sorts timestamps as strings with mixed offsets.

DE red flags: loads whole files with `read()`/`readlines()` "because it's easier"; no answer to "what happens if the job

fails halfway and reruns?"; drops malformed rows without counting or quarantining them; uses naive datetimes or string comparison for timestamps; thinks `df.count()` in Spark is free, or can't say which operations cause a shuffle; uses floats for money.

DS · Data Scientist

Calibration (3–4 years): Vectorises by default, distrusts results that look too good, and can explain every number in their output: where the NaNs went, what the dtype is, and whether a feature could have seen the future. Knows that pandas 3.0 changes copy semantics and string dtypes.

Must-ask master-bank IDs: ET-O01 (float rounding), ET-O03 (exhaustion), E-O05 (sorting), M-O05 (dataclasses), MT-O01 (late binding), MT-O05 (groupby semantics), MT-C01 (memoize with unhashable args), H-O05 (memory measurement).

DS-O01 · Broadcasting and dtype casting in feature scaling

Medium & Tricky · Output · DS, MLE · Round 3 · NumPy broadcasting, dtypes, in-place casting · 7 min

Question: What does this print? (Two stores × three products.)

```python
import numpy as np

prices = np.array([[90, 240, 60], [120, 240, 90]])
means = prices.mean(axis=0)
print(prices.dtype, means.shape, (prices - means).shape)
print((prices - means).tolist())
try:
    prices - prices.mean(axis=1)
except ValueError as exc:
    print("ValueError:", str(exc).strip())
print((prices - prices.mean(axis=1)[:, np.newaxis]).tolist())
try:
    prices -= means
except TypeError as exc:
    print(type(exc).__name__)
```

Expected output:

```
int64 (3,) (2, 3)
[[-15.0, 0.0, -15.0], [15.0, 0.0, 15.0]]
ValueError: operands could not be broadcast together with shapes (2,3) (2,)
[[-40.0, 110.0, -70.0], [-30.0, 90.0, -60.0]]
UFuncTypeError
```

Trap answer: Row-centering "just works" like column-centering, and `prices -= means` quietly produces floats (or truncated ints). Tempting because `prices - means` worked a moment earlier.

Explanation: Broadcasting aligns shapes from the *right*: `(2, 3)` with `(3,)` treats the vector as one row and repeats it down, so column-centering works. Row means have shape `(2,)`, and `(2, 3)` vs `(2,)` compares 3 with 2 on the trailing axis, which is incompatible, hence ValueError. `[:, np.newaxis]` (or `keepdims=True`) reshapes it to `(2, 1)`, which broadcasts across columns. The integer array is `int64` (NumPy 2 uses int64 by default on Windows too). `prices - means` creates a new float array, but `prices -= means` must write floats into the existing int array, which the `same_kind` casting rule forbids, so NumPy raises `UFuncTypeError` (a TypeError subclass) instead of silently truncating. Takeaway: check shapes and use `keepdims=True`; cast explicitly (`prices.astype(np.float64)`) before in-place math.

Real-world bug it maps to: Normalising per row with a `(n,)` vector happens to work when the matrix is square, subtracting the wrong statistic from every cell, and nothing crashes until the shapes change.

Follow-ups:

1. "When would a mismatched broadcast *not* raise?" When the shapes happen to be compatible, e.g. a square `(3, 3)` matrix minus a `(3,)` vector of row means silently subtracts it from each row as if it were column means.
2. "`np.array([1, 2, 3]) / 2` dtype?" float64: true division always produces floats.
3. "Memory of `prices - means` on a (10^7, 100) float64 array?" A full 8 GB temporary; use in-place ops on float arrays or `out=` arguments.

Scoring: Strong: all five lines, reasons from right-aligned shapes, explains the casting rule. Acceptable: gets broadcasting,

unsure about the in-place error. Red flag: fixes shape errors by transposing until it runs.

DS-O02 · Chained assignment under 2.2 vs Copy-on-Write

Hard & Tricky · Output · DS, DE · Round 3 · views vs copies, SettingWithCopy, Copy-on-Write · 10 min

Question: What does this print on **pandas 2.2** (default settings)? Then: what changes on pandas 3.0?

```
import warnings
import pandas as pd

warnings.simplefilter("ignore")
df = pd.DataFrame({"region": ["N", "S", "N"], "sales": [100, 200, 300]})
df[df["region"] == "N"]["sales"] = 0
print(df["sales"].tolist())

north = df[df["region"] == "N"]
north["sales"] = -1
print(df["sales"].tolist(), north["sales"].tolist())

sales = df["sales"]
sales.iloc[0] = 999
print(df["sales"].tolist())

df.loc[df["region"] == "N", "sales"] = 0
print(df["sales"].tolist())
```

Expected output (pandas 2.2):

```
[100, 200, 300]
[100, 200, 300] [-1, -1]
[999, 200, 300]
[0, 200, 0]
```

Pandas 3.0 (Copy-on-Write is the default): line 3 becomes `[100, 200, 300]`; the other lines are unchanged. Warnings also differ: 2.2 emits `SettingWithCopyWarning`, and 3.0 emits `ChainedAssignmentError` for line 1. (Enable `pd.options.mode.copy_on_write = True` on 2.2 to get 3.0 behaviour today.)

Trap answer: `[0, 200, 0]` on line 1 ("I set the N rows to 0"), and `[100, 200, 300]` on line 3 in 2.2. Tempting because both lines read like "modify df".

Explanation: Boolean indexing (`df[mask]`) always creates a copy. Line 1 is chained assignment: `df[mask]` builds a temporary copy, then `["sales"] = 0` modifies that temporary, which is immediately discarded, so `df` never changes. `north` is likewise a copy; assigning into it leaves `df` alone. Line 3 is the version-dependent one: in 2.2, selecting a single column returns a Series that can share memory with the DataFrame, so writing through `sales.iloc[0]` mutates `df` (a "spooky action at a distance" bug). Under Copy-on-Write every indexing result *behaves* as a copy: data is shared lazily and copied on the first write, so `df` is untouched. `.loc[mask, col] = value` is a single operation on `df` itself and works in both versions. Takeaway: always write with one `.loc`/`.iloc` call on the object you mean to modify; never rely on views.

Real-world bug it maps to: A cleaning step "sets outliers to NaN" through chained indexing, silently does nothing, and the model trains on raw outliers. Or, in 2.2, a helper modifies a Series extracted from a shared DataFrame and corrupts the source data for every later step.

Follow-ups:

1. "What else does pandas 3.0 change by default?" String columns get the dedicated `str` dtype (PyArrow-backed when available) instead of `object`; code checking `dtype == object` breaks (see DS-O03).
2. "`df2 = df; df2["x"] = 1`?" Plain aliasing: both names are the same object in every version. CoW is about indexing results, not assignment.
3. "Does CoW make pandas slower?" Usually faster/leaner: fewer defensive copies, and copies happen only on the first write.

Scoring: Strong: all four lines, names chained assignment, explains the 3.0 change on line 3. Acceptable: knows chained assignment fails, unsure about the Series view. Red flag: silences `SettingWithCopyWarning` globally as the fix.

DS-O03 · NaN in groupby keys and aggregations

Medium · Output · DS · Round 3 · NaN semantics, groupby, dtypes · 6 min

Question: What does this print on pandas 2.2?

```python
import numpy as np
import pandas as pd

df = pd.DataFrame({
    "segment": ["gold", "silver", None, "gold", "silver"],
    "spend": [100.0, np.nan, 50.0, 300.0, 80.0],
})
g = df.groupby("segment")["spend"]
print(g.mean().to_dict())
print(g.size().to_dict(), g.count().to_dict())
print(df.groupby("segment", dropna=False)["spend"].sum().to_dict())
print(np.nan == np.nan, df["spend"].mean(), df["spend"].to_numpy().mean())
print(df["segment"].dtype)
```

Expected output (pandas 2.2):

```
{'gold': 200.0, 'silver': 80.0}
{'gold': 2, 'silver': 2} {'gold': 2, 'silver': 1}
{'gold': 400.0, 'silver': 80.0, nan: 50.0}
False 132.5 nan
object
```

(pandas 3.0: identical except the last line prints `str`, the new default string dtype.)

Explanation: `groupby` drops rows whose *key* is missing by default (`dropna=True`), so the `None` segment and its 50.0 vanish from lines 1–2 without any warning. Aggregations skip missing *values*: silver's mean is 80/1, not (NaN + 80)/2. `size()` counts rows including NaN values, while `count()` counts non-null values, so the gap between them is your missing-data rate. `dropna=False` keeps the missing key as its own group (`nan`). `NaN != NaN` by IEEE 754 definition, which is why you test with `isna()`. pandas `mean()` skips NaN (`skipna=True`), while NumPy's `mean()` propagates it. Takeaway: always check `size()` vs `count()` and whether keys contained nulls before trusting a grouped metric.

Follow-ups:

1. "Revenue by segment sums to less than total revenue. Why?" Null segment keys dropped by `groupby`; `dropna=False` or fill with `"unknown"` first.
2. "`df[df["spend"] == np.nan]`?" Always empty; use `df["spend"].isna()`.
3. "What changes with the pandas 3.0 `str` dtype for missing values?" Missing strings are represented consistently as NaN in the new dtype; code comparing to `None` or checking `dtype == object` must be updated.

Scoring: Strong: all five lines and explains key-dropping vs value-skipping. Acceptable: misses the dropped None group. Red flag: surprised that NaN ≠ NaN.

DS-C01 · Leakage-free rolling features per customer

Medium · Coding · DS, MLE · Round 3 · groupby/shift/rolling, train–serve leakage · 20 min

Problem: For a churn model, add two features to a transactions DataFrame (`customer_id`, `ts`, `amount`, rows unsorted): `avg_prev_3`, the mean of the customer's previous up to 3 transactions **strictly before** the current one (NaN if none), and `txn_count_before`, the number of earlier transactions. The current row must never contribute to its own features. Return a new DataFrame in the original row order with the original index. Raise `ValueError` if the index isn't unique.

Signature: `def add_history_features(df: pd.DataFrame, window: int = 3) -> pd.DataFrame:`

Constraints: pandas 2.2 and 3.0. Up to 10^7 rows. Ties in `ts` for a customer keep input order.

Examples:

- c1 transactions at t1=10, t2=20, t3=30 → `avg_prev_3` = NaN, 10.0, 15.0; `txn_count_before` = 0, 1, 2
- A customer with one transaction → NaN, 0

Trap answer: `df.groupby("customer_id")["amount"].rolling(3).mean()`. It includes the current row (label leakage, so the feature correlates with the target), returns a MultiIndex that doesn't align back to `df`, and ignores time order on unsorted input.

Real-world bug it maps to: Offline AUC of 0.95 that collapses to 0.6 in production, because at serving time the current transaction's amount isn't known yet when the prediction is made.

Hints:

1. "Which rows may the feature for row i see?"
2. "What does `shift(1)` do inside a group?"
3. "How do you get results back in the original order?"

Reference solution:

```python
import numpy as np
import pandas as pd


def add_history_features(df: pd.DataFrame, window: int = 3) -> pd.DataFrame:
    """Add per-customer features computed only from strictly earlier rows.

    avg_prev_3: mean of up to `window` previous amounts (NaN if none).
    txn_count_before: number of earlier transactions.
    Output keeps the input's row order and index.
    """
    if not df.index.is_unique:
        raise ValueError("index must be unique to align features back")
    out = df.copy()
    ordered = out.sort_values(["customer_id", "ts"], kind="stable")
    by_customer = ordered.groupby("customer_id", sort=False)["amount"]
    out["avg_prev_3"] = by_customer.transform(
        lambda s: s.shift(1).rolling(window, min_periods=1).mean()
    )                                                  # aligns on index -> original order
    out["txn_count_before"] = ordered.groupby("customer_id", sort=False).cumcount()
    return out
```

Complexity: O(n log n) for the sort; the grouped rolling pass is O(n). O(n) extra memory for the sorted copy.

Tests:

```python
tx = pd.DataFrame(
    {"customer_id": ["c1", "c2", "c1", "c1", "c2", "c3", "c1"],
     "ts":          [20,   5,    10,   30,   6,    1,    40],
     "amount":      [20.0, 7.0,  10.0, 30.0, 9.0,  4.0,  100.0]},
    index=[101, 102, 103, 104, 105, 106, 107],
)
out = add_history_features(tx)
assert list(out.index) == [101, 102, 103, 104, 105, 106, 107]
got = [None if np.isnan(v) else v for v in out["avg_prev_3"].tolist()]
# c1 by time: 10, 20, 30, 100 -> NaN, 10, 15, 20 ; c2: 7, 9 -> NaN, 7 ; c3 -> NaN
assert got == [10.0, None, None, 15.0, 7.0, None, 20.0]
assert out["txn_count_before"].tolist() == [1, 0, 0, 2, 1, 0, 3]
assert "avg_prev_3" not in tx.columns                    # input not modified

single = add_history_features(pd.DataFrame({"customer_id": ["x"], "ts": [1], "amount": [5.0]}))
assert np.isnan(single["avg_prev_3"].iloc[0]) and single["txn_count_before"].iloc[0] == 0

empty = add_history_features(pd.DataFrame({"customer_id": [], "ts": [], "amount": []}))
assert len(empty) == 0 and {"avg_prev_3", "txn_count_before"} <= set(empty.columns)
```

```
try:
    add_history_features(tx.set_axis([1, 1, 2, 3, 4, 5, 6]))
    raise AssertionError("expected ValueError")
except ValueError:
    pass
```

Alternatives and tradeoffs: Time-based windows (`rolling("30D", on="ts", closed="left")`) express "previous 30 days" rather than "previous 3 rows", and `closed="left"` excludes the current timestamp, which is what you want when several transactions share a timestamp. A SQL window (`AVG(amount) OVER (PARTITION BY customer ORDER BY ts ROWS BETWEEN 3 PRECEDING AND 1 PRECEDING)`) is the same logic in the warehouse, and a feature store computes it point-in-time correctly for both training and serving. The `transform(lambda ...)` runs Python per group; for millions of groups, `groupby().shift()` followed by `groupby().rolling()` avoids the lambda. Pick: SQL/feature store for production features; this pandas version for exploration, matching the serving definition exactly.

Follow-ups:

- "How do you prove there's no leakage?" Recompute the feature for a row using only data with `ts < row.ts` and assert equality; also check that train/validation splits are by time, not random.
- "Two transactions at the same timestamp." Decide: exclude same-timestamp rows (time-based `closed="left"`) so simultaneous events don't see each other.
- "Serving must match training." Compute the online feature from the same definition (feature store, or shared SQL), not a re-implementation.

Scoring: Strong: `shift(1)` before rolling, stable sort, index alignment, articulates leakage. Acceptable: correct values after the hint about `shift`. Red flag: includes the current row and calls it fine "because it's a rolling mean".

DS red flags: iterates rows with `iterrows()` for numeric work; random train/test split on time-series data; can't explain where NaNs went in an aggregate; silences all warnings instead of understanding them; unseeded randomness in reported results; accepts a surprisingly high metric without checking for leakage.

MLE · Machine Learning Engineer

Calibration (3–4 years): Bridges notebooks and production. Knows why the model was fast in the notebook and slow in the API: event-loop blocking, per-request model loading, Python loops over vectors, copies of large arrays. Makes runs reproducible and memory predictable.

Must-ask master-bank IDs: E-C01 (batching), ET-O04 (hashing), MT-C01 (caching with unhashable args), H-O01 (thread pools), H-O02 (blocking the loop), H-O05 (memory), HT-O02 (cached_property), HT-C03 (cache leaks).

MLE-O01 · In-place normalisation through NumPy views

Medium & Tricky · Output · MLE, DS · Round 3 · views vs copies, memory, contiguity · 7 min

Question: What does this print?

```python
import numpy as np

embeddings = np.array([[3.0, 4.0], [0.0, 2.0], [6.0, 8.0]])
batch = embeddings[:2]
batch /= np.linalg.norm(batch, axis=1, keepdims=True)
print(embeddings.tolist())
print(np.shares_memory(batch, embeddings))

picked = embeddings[[0, 2]]
picked *= 0
print(embeddings[2].tolist(), np.shares_memory(picked, embeddings))

flat = embeddings.T.reshape(-1)
flat[0] = -1.0
print(embeddings[0, 0], embeddings.T.flags["C_CONTIGUOUS"])
```

Expected output:

```
[[0.6, 0.8], [0.0, 1.0], [6.0, 8.0]]
True
[6.0, 8.0] False
0.6 False
```

Trap answer: The original `embeddings` stays `[[3.0, 4.0], ...]` after normalising `batch`, and `picked *= 0` zeroes rows 0 and 2. Tempting because `batch` and `picked` look like "new variables".

Explanation: Basic slicing (`a[:2]`, `a[::2]`) returns a **view**: a new array header pointing into the same buffer, so the in-place `/=` rewrote the first two rows of `embeddings`. Fancy (integer-list or boolean) indexing returns a **copy**, so zeroing `picked` leaves the original untouched. `.T` is a view with swapped strides (no longer C-contiguous); `reshape(-1)` can't express that layout as a flat view, so it returns a copy and writing to `flat` has no effect. The rule "reshape returns a view when possible" makes this layout-dependent. `np.shares_memory` is the reliable check. Takeaway: in-place operations on slices mutate the source, and whether you got a view depends on the indexing type and memory layout.

Real-world bug it maps to: A preprocessing step normalises "a batch" in place, silently modifying the embedding cache shared by the whole service, so every later similarity score uses doubly-normalised or corrupted vectors.

Follow-ups:

1. "How do you make a slice safe to modify?" `.copy()`, or compute out-of-place (`batch = batch / norms`).
2. "How do you prevent accidental writes to a shared array?" `arr.setflags(write=False)`; in-place ops then raise.
3. "Why do views matter for memory?" Slicing a 10 GB array is free, but holding a small view keeps the whole 10 GB buffer alive.

Scoring: Strong: all four lines, explains view vs copy rules and contiguity. Acceptable: knows slices are views, misses the reshape case. Red flag: believes every assignment from indexing copies.

MLE-O02 · Seeding random, NumPy and generators

Medium · Output · MLE, DS · Round 3 · reproducibility, RNG state · 5 min

Question: What does this print?

```python
import random
import numpy as np

np.random.seed(0)
random.seed(42); a = np.random.rand(3)
random.seed(42); b = np.random.rand(3)
print(np.array_equal(a, b))

rng1 = np.random.default_rng(42)
rng2 = np.random.default_rng(42)
print(np.array_equal(rng1.random(3), rng2.random(3)))

np.random.seed(7); x = np.random.rand(2)
np.random.seed(7); _ = np.random.default_rng(0).random(5); y = np.random.rand(2)
print(np.array_equal(x, y))

children = np.random.default_rng(123).spawn(2)
print(children[0].random() == children[1].random())
```

Expected output:

```
False
True
True
False
```

Explanation: Python's `random` module and NumPy's legacy global generator (`np.random.*`) are separate RNGs with separate state, so `random.seed` does nothing for NumPy: `a` and `b` are consecutive draws from NumPy's stream. Two `default_rng(42)` Generators are independent objects with identical seeds, so they produce identical sequences. A `Generator` has its own state and doesn't touch the global one, so drawing from it between the two `np.random.rand` calls doesn't disturb `y`. `spawn` (NumPy 1.25+) derives statistically independent child streams, ideal for parallel workers that must not share a sequence. Takeaway: seed every RNG you use (Python, NumPy, framework, CUDA), prefer passing explicit `Generator` objects over global state, and give each worker its own spawned stream.

Follow-ups:

1. "What else must be seeded in PyTorch training?" `torch.manual_seed`, CUDA seeds, `DataLoader` worker seeds (`worker_init_fn/generator`), plus `torch.use_deterministic_algorithms(True)` since some GPU kernels are non-deterministic.
2. "`multiprocessing` with `fork` and the global NumPy RNG?" Every child inherits the same state and generates identical "random" augmentations; spawn per-worker generators.
3. "Is `PYTHONHASHSEED` related?" It controls str/bytes hash randomisation (set iteration order), another source of run-to-run differences.

Scoring: Strong: all four lines, recommends explicit Generators. Acceptable: three of four. Red flag: believes `random.seed` seeds everything.

MLE-C01 · FastAPI inference service with startup model loading

Hard · Coding · MLE · Round 3 · model serving, Pydantic v2 validation, sync vs async endpoints · 35 min

Problem: Serve a (stand-in) linear model over HTTP.

- Load the model **once per process at startup**, not per request and not at import time, so tests can inject a fake loader.
- `POST /predict` takes `{"features": [f1, f2, f3, f4]}` and returns `{"score": float, "model_version": str}`. Exactly 4 finite floats; NaN/Infinity rejected with 422.

- `POST /predict/batch` accepts 1–256 items.
- `GET /healthz` reports the model version.
- Inference is CPU-bound and must not block the event loop.

Signature: `def create_app(loader: Callable[[], LinearModel] = load_model) -> FastAPI:`

Constraints: FastAPI + Pydantic v2. The model object is read-only after loading.

Examples:

- `{"features": [1, 2, 3, 4]}` → `{"score": 5.6, "model_version": "ranker-v3"}`
- `{"features": [1, 2, 3]}` → 422
- Body `{"features": [NaN, 1, 2, 3]}` (Python's JSON parser accepts `NaN`) → 422

Hints:

1. "Where does FastAPI let you run code before the first request?" (lifespan)
2. "`def` or `async def` for a CPU-bound endpoint?"
3. "Does Pydantic accept NaN for a `float` field by default? And once you reject it, what does FastAPI's default 422 response put in the body?"

Reference solution:

```python
from collections.abc import AsyncIterator, Callable
from contextlib import asynccontextmanager
from dataclasses import dataclass
from typing import Annotated

from fastapi import FastAPI, Request
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse
from pydantic import BaseModel, Field

FiniteFloat = Annotated[float, Field(allow_inf_nan=False)]


@dataclass(frozen=True)
class LinearModel:
    version: str
    weights: tuple[float, ...]
    bias: float

    def predict(self, features: list[float]) -> float:
        return sum(w * x for w, x in zip(self.weights, features, strict=True)) + self.bias


def load_model() -> LinearModel:
    """Stand-in for an expensive load (joblib/ONNX/torch) from disk or a registry."""
    return LinearModel(version="ranker-v3", weights=(0.5, -1.0, 2.0, 0.25), bias=0.1)


class PredictRequest(BaseModel):
    features: list[FiniteFloat] = Field(min_length=4, max_length=4)


class PredictResponse(BaseModel):
    score: float
    model_version: str


class BatchRequest(BaseModel):
    items: list[PredictRequest] = Field(min_length=1, max_length=256)


def create_app(loader: Callable[[], LinearModel] = load_model) -> FastAPI:
    @asynccontextmanager
```

```
    async def lifespan(app: FastAPI) -> AsyncIterator[None]:
        app.state.model = loader()          # once per worker process, before traffic
        yield
        app.state.model = None              # release on shutdown

    app = FastAPI(lifespan=lifespan)

    @app.exception_handler(RequestValidationError)
    async def validation_error(request: Request, exc: RequestValidationError) -> JSONResponse:
        # FastAPI's default handler echoes the offending input back to the client.
        # For a NaN input that response can't be rendered (Starlette's JSON encoder
        # uses allow_nan=False), so the client would get a 500 instead of a 422.
        # Returning loc/msg/type only also avoids echoing sensitive payloads.
        errors = [{"loc": list(e["loc"]), "msg": e["msg"], "type": e["type"]} for e in
exc.errors()]
        return JSONResponse(status_code=422, content={"detail": errors})

    @app.get("/healthz")
    def healthz(request: Request) -> dict[str, str]:
        return {"status": "ok", "model_version": request.app.state.model.version}

    # Plain `def`: FastAPI runs these in a threadpool, so CPU work doesn't
    # block the event loop that serves every other request.
    @app.post("/predict", response_model=PredictResponse)
    def predict(body: PredictRequest, request: Request) -> PredictResponse:
        model: LinearModel = request.app.state.model
        return PredictResponse(score=model.predict(body.features), model_version=model.version)

    @app.post("/predict/batch", response_model=list[PredictResponse])
    def predict_batch(body: BatchRequest, request: Request) -> list[PredictResponse]:
        model: LinearModel = request.app.state.model
        return [PredictResponse(score=model.predict(i.features), model_version=model.version)
                for i in body.items]

    return app
```

Complexity: O(d) per prediction for d features (O(n·d) per batch); the model is loaded once per process.

Tests:

```
import math
from fastapi.testclient import TestClient

loads: list[int] = []

def counting_loader() -> LinearModel:
    loads.append(1)
    return load_model()

app = create_app(counting_loader)
assert loads == []                                  # nothing loaded at import/create
time
with TestClient(app) as client:                     # `with` runs the lifespan
    r = client.post("/predict", json={"features": [1, 2, 3, 4]})
    assert r.status_code == 200
    assert math.isclose(r.json()["score"], 5.6) and r.json()["model_version"] == "ranker-v3"
    client.post("/predict", json={"features": [0, 0, 0, 0]})
    assert loads == [1]                             # loaded exactly once

    assert client.post("/predict", json={"features": [1, 2, 3]}).status_code == 422
    nan_body = '{"features": [NaN, 1, 2, 3]}'
    r = client.post("/predict", content=nan_body, headers={"content-type": "application/json"})
    assert r.status_code == 422
    assert client.post("/predict", json={"features": ["x", 1, 2, 3]}).status_code == 422
    assert client.post("/predict", json={"features": ["1", 2, 3, 4]}).status_code == 200   #
lax mode coerces
```

```
    r = client.post("/predict/batch", json={"items": [{"features": [1, 2, 3, 4]}] * 3})
    assert r.status_code == 200 and len(r.json()) == 3
    assert client.post("/predict/batch", json={"items": []}).status_code == 422
    assert client.get("/healthz").json() == {"status": "ok", "model_version": "ranker-v3"}
assert app.state.model is None                          # released on shutdown
```

Alternatives and tradeoffs: `async def` endpoint + `await asyncio.to_thread(model.predict, ...)`: explicit and equivalent, useful when the endpoint also awaits async I/O. A process pool (or multiple Uvicorn/Gunicorn workers) is needed when inference holds the GIL for long, because threads only help if the library releases the GIL (NumPy, ONNX Runtime, and PyTorch mostly do). Dedicated servers (Triton, TorchServe, BentoML, Ray Serve) add dynamic batching and GPU scheduling. Pick: this pattern for light CPU models; multiple workers for GIL-bound Python; a model server when you need GPU batching.

Follow-ups:

- "Why not load the model at module import?" Import-time side effects slow every test and CLI, can't be injected, and with `--preload` + fork can break libraries holding threads or GPU contexts.
- "Four workers, each loading a 2 GB model." 8 GB of RAM; consider memory-mapped weights, a shared model server, or fewer workers with threads.
- "Hot-swap to a new version without downtime." Load the new model in the background, then atomically replace `app.state.model` (a reference swap), or do a rolling deploy.
- "Why the custom validation handler?" Found while verifying this bank: FastAPI's default 422 body includes the rejected `input`, and a NaN input makes that body unrenderable (Starlette's JSON encoder refuses NaN), so the client gets a 500. A candidate who discovers this by testing the NaN case is showing exactly the production instinct this round looks for.

Scoring: Strong: lifespan loading, sync `def` (or `to_thread`) with a clear reason, strict validation including NaN (and tests that the NaN case really returns 422), testable via injected loader. Acceptable: works but loads per request or at import time, fixed after hint 1. Red flag: `async def` endpoint calling a heavy CPU `predict` directly and claiming async makes it faster.

MLE-C02 · Top-k cosine similarity search

Medium · Coding · MLE, DS · Round 3 · embeddings, vectorised NumPy, argpartition, normalisation · 20 min

Problem: Brute-force semantic search over an in-memory embedding matrix (the baseline to compare an ANN index against). Return the `k` rows most similar to the query by cosine similarity as `(row_index, similarity)` pairs, highest first, **ties broken by lower index** (including ties at the k-th boundary, so results are deterministic). Zero-norm rows (failed embeddings) are never returned. Reject shape mismatches and a zero query with `ValueError`. Clamp similarities to [−1, 1].

Signature: `def top_k_cosine(query: np.ndarray, corpus: np.ndarray, k: int) -> list[tuple[int, float]]:`

Constraints: corpus up to 10^6 × 1024. No Python loops over rows. k may exceed the number of valid rows.

Examples:

- corpus rows `[1,0]`, `[0,1]`, `[1,1]`, `[0,0]`, `[2,0]`, `[-1,0]`, query `[1,0]`, `k=3` → `[(0, 1.0), (4, 1.0), (2, 0.7071…)]`
- same with `k=1` → `[(0, 1.0)]` (tie at the boundary → lower index)
- `k=0` → `[]`

Hints:

1. "Cosine similarity of all rows at once: one matrix–vector product divided by norms."
2. "Full sort is O(n log n). How do you get the top k in O(n)?"
3. "`argpartition` doesn't order the k results and breaks ties arbitrarily. How do you make ties deterministic?"

Reference solution:

```python
import numpy as np


def top_k_cosine(query: np.ndarray, corpus: np.ndarray, k: int) -> list[tuple[int, float]]:
    """Return the k corpus rows most similar to `query` by cosine similarity.

    Highest similarity first; ties (including at the k-th boundary) go to the
    lower row index. Zero-norm rows are excluded.
    """
    query = np.asarray(query, dtype=np.float64)
    corpus = np.asarray(corpus, dtype=np.float64)
    if corpus.ndim != 2 or query.shape != (corpus.shape[1],):
        raise ValueError(f"shape mismatch: query {query.shape}, corpus {corpus.shape}")
    q_norm = np.linalg.norm(query)
    if q_norm == 0:
        raise ValueError("query vector has zero norm")
    norms = np.linalg.norm(corpus, axis=1)
    valid = np.flatnonzero(norms > 0)
    k = min(k, valid.size)
    if k <= 0:
        return []
    sims = np.clip(corpus[valid] @ query / (norms[valid] * q_norm), -1.0, 1.0)
    kth_best = np.partition(sims, sims.size - k)[sims.size - k]     # O(n) selection
    candidates = np.flatnonzero(sims >= kth_best)                    # everything tied with k-
th
    order = np.lexsort((valid[candidates], -sims[candidates]))[:k]   # sim desc, index asc
    chosen = candidates[order]
    return [(int(valid[i]), float(sims[i])) for i in chosen]
```

Complexity: O(n·d) for the matrix–vector product, dominating; O(n) selection plus O(c log c) for the c boundary candidates (≈ k). Memory O(n) for similarities (plus a copy of the valid rows; see follow-ups).

Tests:

```python
corpus = np.array([[1, 0], [0, 1], [1, 1], [0, 0], [2, 0], [-1, 0]], dtype=float)
res = top_k_cosine(np.array([1.0, 0.0]), corpus, 3)
assert [i for i, _ in res] == [0, 4, 2] and np.isclose(res[2][1], 1 / np.sqrt(2))
assert top_k_cosine(np.array([1.0, 0.0]), corpus, 1) == [(0, 1.0)]          # boundary tie
assert [i for i, _ in top_k_cosine(np.array([1.0, 0.0]), corpus, 99)] == [0, 4, 2, 1, 5]
assert top_k_cosine(np.array([1.0, 0.0]), corpus, 0) == []
assert top_k_cosine(np.array([1.0, 0.0]), np.zeros((0, 2)), 5) == []
for bad_query, bad_corpus in ((np.zeros(2), corpus), (np.ones(3), corpus)):
    try:
        top_k_cosine(bad_query, bad_corpus, 2)
        raise AssertionError("expected ValueError")
    except ValueError:
        pass

rng = np.random.default_rng(0)
big = rng.normal(size=(2000, 16))
q = rng.normal(size=16)
brute = sorted(range(2000), key=lambda i: (-(big[i] @ q) / (np.linalg.norm(big[i]) *
np.linalg.norm(q)), i))
assert [i for i, _ in top_k_cosine(q, big, 10)] == brute[:10]
```

Alternatives and tradeoffs: Pre-normalise the corpus once at load time (then cosine = dot product) to halve per-query work; store it as float32 to halve memory, accepting tiny ranking differences. `np.argsort(-sims)[:k]` is simpler but O(n log n). ANN indexes (FAISS, HNSW via hnswlib, pgvector, a vector DB) trade exactness for sub-linear search. Pick: pre-normalised float32 brute force up to a few million vectors (it's exact and often fast enough); ANN beyond that, with this function as the recall baseline.

Follow-ups:

- "`corpus[valid]` copies 4 GB on every query." Filter zero rows once at load time and keep a pre-normalised matrix, so each query is one `@` and no copies.

- "Batch of 64 queries?" One matrix–matrix product `Q @ C.T` (BLAS-efficient), then `argpartition` along axis 1.
- "Why clamp to [−1, 1]?" Floating-point error can yield 1.0000000002, which breaks `arccos` and threshold logic.

Scoring: Strong: fully vectorised, O(n) selection, deterministic ties, handles zero vectors, knows to pre-normalise. Acceptable: vectorised with full argsort. Red flag: Python loop computing similarity row by row.

MLE red flags: loads the model inside the request handler; `async def` everywhere "for speed" with CPU-bound work inside; pickles models without version pinning or awareness that unpickling untrusted data executes code; seeds only one of several RNGs; measures memory with `sys.getsizeof`; no batching story for inference or preprocessing.

BE · Python Backend Engineer

Calibration (3–4 years): Builds correct, testable services and understands concurrency and failure modes beyond the happy path: retries and idempotency, transactions and rollbacks, rate limits, where to patch in tests, and why one blocking call hurts every request.

Must-ask master-bank IDs: MT-O02 (mutable default leak), M-O06 (patch where looked up), M-O04 (exception chaining), M-C01 (retry decorator), H-O02 (blocking the loop), H-C01 (bounded async), HT-O04 (TaskGroup), HT-C02 (nested transactions).

BE-O01 · Pydantic v2 coercion, strict mode and dumping

Medium · Output · BE, FS · Round 3 · Pydantic v2 validation and serialisation · 7 min

Question: What does this print?

```python
from datetime import datetime
from decimal import Decimal
from pydantic import BaseModel, ConfigDict, ValidationError

class Order(BaseModel):
    order_id: int
    amount: Decimal
    placed_at: datetime
    tags: list[str] = []

class StrictOrder(Order):
    model_config = ConfigDict(strict=True, extra="forbid")

o = Order(order_id="42", amount="19.99", placed_at="2026-09-24T10:00:00Z", coupon="X")
print(o.order_id, repr(o.amount), o.placed_at.utcoffset())
print("coupon" in o.model_dump(), type(o.model_dump()["amount"]).__name__)
print(o.model_dump(mode="json"))
try:
    StrictOrder(order_id="42", amount=Decimal("1"), placed_at=o.placed_at, coupon="X")
except ValidationError as exc:
    print(exc.error_count(), [e["type"] for e in exc.errors()])
a = Order(order_id=1, amount=1, placed_at=o.placed_at)
b = Order(order_id=2, amount=1, placed_at=o.placed_at)
a.tags.append("vip")
print(b.tags)
```

Expected output:

```
42 Decimal('19.99') 0:00:00
False Decimal
{'order_id': 42, 'amount': '19.99', 'placed_at': '2026-09-24T10:00:00Z', 'tags': []}
2 ['int_type', 'extra_forbidden']
[]
```

Explanation: Pydantic v2 validates in **lax mode** by default: `"42"` is coerced to `int`, `"19.99"` to `Decimal`, and the ISO string with `Z` to an aware UTC datetime. Unknown fields are **ignored** by default (`extra="ignore"`), so `coupon` silently disappears: convenient, but it also hides client typos. `model_dump()` returns Python objects (Decimal stays Decimal); `model_dump(mode="json")` returns JSON-safe types: Decimal becomes a string (no float precision loss) and datetimes become ISO 8601 with `Z`. Strict mode rejects the string-to-int coercion (`int_type`) and `extra="forbid"` rejects the unknown field; errors are collected, not raised one at a time. Unlike a plain function default (MT-O02), Pydantic copies mutable field defaults per instance, so `b.tags` is independent. Takeaway: know which fields should be strict (IDs, money) and whether unknown fields should fail.

Follow-ups:

1. "Strict for one field only?" `order_id: Annotated[int, Strict()]` or `StrictInt`.
2. "Why does `amount` serialise as a string in JSON mode?" Decimal → JSON number would round-trip through float on

most clients; a string preserves exact value. Document it in the API contract.

3. "Flask/Django equivalents?" Marshmallow schemas or Django REST Framework serializers; FastAPI uses Pydantic models directly for request bodies and `response_model`.

Scoring: Strong: all five lines including dropped `coupon` and per-instance defaults. Acceptable: coercion right, surprised by `extra` or JSON-mode Decimal. Red flag: assumes Pydantic v1 behaviour (`.dict()`, silent float for Decimal) without noticing.

BE-O02 · sqlite3 transactions with the connection context manager

Medium & Tricky · Output · BE, DE · Round 3 · transactions, rollback, connection lifecycle · 8 min

Question: What does this print?

```python
import os
import sqlite3
import tempfile

path = os.path.join(tempfile.mkdtemp(), "shop.db")
writer = sqlite3.connect(path)
writer.execute("CREATE TABLE payments (id INTEGER PRIMARY KEY, amount INTEGER NOT NULL)")
with writer:
    writer.execute("INSERT INTO payments VALUES (1, 100)")
try:
    with writer:
        writer.execute("INSERT INTO payments VALUES (2, 200)")
        writer.execute("INSERT INTO payments VALUES (1, 999)")
except sqlite3.IntegrityError as exc:
    print("IntegrityError:", exc)
writer.execute("INSERT INTO payments VALUES (3, 300)")
print(writer.in_transaction)
reader = sqlite3.connect(path)
print(reader.execute("SELECT id FROM payments ORDER BY id").fetchall())
writer.close()
print(reader.execute("SELECT id FROM payments ORDER BY id").fetchall())
```

Expected output:

```
IntegrityError: UNIQUE constraint failed: payments.id
True
[(1,)]
[(1,)]
```

Trap answer: Payment 2 is saved (it succeeded before the failure), payment 3 is visible to the reader, and `writer.close()` saves it. Tempting because each `execute` "ran fine".

Explanation: `with conn:` is a *transaction* scope, not a connection scope: it commits on success, rolls back on exception, and does **not** close the connection. The failing duplicate key rolls back the whole block, including payment 2. Outside a `with`, the sqlite3 module (legacy transaction control, the 3.12 default) implicitly opens a transaction before an INSERT, so payment 3 is pending (`in_transaction` is True). Another connection can't see uncommitted data. `close()` does **not** commit, so the pending insert is discarded. Takeaway: make transaction boundaries explicit (`with conn:` or explicit commit) and treat `close()` without commit as a rollback.

Real-world bug it maps to: A script or worker "writes" rows, exits without committing, and the data never appears; or a partial batch is expected to persist but the whole transaction rolled back on one bad row.

Follow-ups:

1. "Python 3.12's `autocommit` attribute?" PEP 249-compliant control: `autocommit=False` means an implicit transaction is always open and `commit()`/`rollback()` start a new one; `autocommit=True` runs each statement in its own transaction. The legacy `isolation_level` behaviour remains the default for now.
2. "Why use `with closing(sqlite3.connect(...)) as conn, conn:`?" `closing` closes the connection and the inner `conn` manages the transaction.

3. "Isolation?" SQLite serialises writers; readers see the last committed state (WAL mode lets readers proceed during a write). Postgres defaults to READ COMMITTED.

Scoring: Strong: all four lines and states that `with conn` doesn't close. Acceptable: rollback right, thinks `close()` commits. Red flag: believes each `execute` autocommits.

BE-C01 · Per-client token-bucket rate limiter

Hard · Coding · BE, FS · Round 3 · rate limiting, locks, injectable clocks · 30 min

Problem: Implement an in-process rate limiter for API keys: each key gets a bucket holding up to `capacity` tokens, refilled continuously at `rate` tokens per second; a request costing `cost` tokens is allowed only if enough tokens are available. Provide `allow(key, cost=1)` and `retry_after(key, cost=1)` (seconds until allowed, for the `Retry-After` header). It must be thread-safe, and the clock injectable. A cost larger than capacity is a programming error.

Signature: `class TokenBucketLimiter:` with `__init__(self, rate: float, capacity: float, clock: Callable[[], float] = time.monotonic)`, `allow(self, key: str, cost: float = 1.0) -> bool`, `retry_after(self, key: str, cost: float = 1.0) -> float`.

Constraints: Up to 10^5 active keys; O(1) per call.

Examples:

- `rate=2, capacity=5`: 5 immediate calls allowed, the 6th denied; `retry_after` ≈ 0.5 s
- After 1 s, 2 more calls allowed
- A new key starts with a full bucket

Hints:

1. "Do you need a background thread to add tokens?"
2. "Where must the clock be read relative to the lock?"
3. "What should happen to tokens after a long idle period?"

Reference solution:

```python
import threading
import time
from collections.abc import Callable


class TokenBucketLimiter:
    """Per-key token buckets refilled lazily at `rate` tokens/s, bursting to `capacity`."""

    def __init__(self, rate: float, capacity: float,
                 clock: Callable[[], float] = time.monotonic) -> None:
        if rate <= 0 or capacity <= 0:
            raise ValueError("rate and capacity must be > 0")
        self._rate = rate
        self._capacity = capacity
        self._clock = clock
        self._buckets: dict[str, tuple[float, float]] = {}   # key -> (tokens, last_refill)
        self._lock = threading.Lock()

    def _available(self, key: str, now: float) -> float:
        tokens, last = self._buckets.get(key, (self._capacity, now))
        return min(self._capacity, tokens + max(0.0, now - last) * self._rate)

    def _check_cost(self, cost: float) -> None:
        if not 0 < cost <= self._capacity:
            raise ValueError(f"cost must be in (0, {self._capacity}]")

    def allow(self, key: str, cost: float = 1.0) -> bool:
        self._check_cost(cost)
        with self._lock:
            now = self._clock()              # read under the lock: per-key time never runs
```

```
backwards
            tokens = self._available(key, now)
            allowed = tokens >= cost
            self._buckets[key] = (tokens - cost if allowed else tokens, now)
            return allowed

    def retry_after(self, key: str, cost: float = 1.0) -> float:
        """Seconds until `cost` tokens are available (0.0 if available now)."""
        self._check_cost(cost)
        with self._lock:
            tokens = self._available(key, self._clock())
            return 0.0 if tokens >= cost else (cost - tokens) / self._rate
```

Complexity: O(1) time per call; O(active keys) memory.

Tests:

```
now = [100.0]
limiter = TokenBucketLimiter(rate=2, capacity=5, clock=lambda: now[0])
assert [limiter.allow("key-a") for _ in range(6)] == [True] * 5 + [False]
assert limiter.retry_after("key-a") == 0.5
assert limiter.allow("key-b")                                    # independent keys
now[0] += 1.0
assert [limiter.allow("key-a") for _ in range(3)] == [True, True, False]
now[0] += 1000
assert sum(limiter.allow("key-a") for _ in range(10)) == 5       # capped at capacity, no
hoarding
assert limiter.retry_after("key-a", cost=2) == 1.0
now[0] += 1.0
assert limiter.allow("key-a", cost=2) and limiter.retry_after("key-a") == 0.5

for bad in (0, 6):
    try:
        limiter.allow("key-a", cost=bad)
        raise AssertionError("expected ValueError")
    except ValueError:
        pass

frozen = TokenBucketLimiter(rate=1, capacity=10, clock=lambda: 0.0)
results: list[bool] = []
def hit() -> None:
    results.append(frozen.allow("shared"))
threads = [threading.Thread(target=hit) for _ in range(50)]
for t in threads: t.start()
for t in threads: t.join()
assert results.count(True) == 10                                 # no double-spend under
contention
```

Alternatives and tradeoffs: A sliding-window log (store timestamps per key) is exact but O(requests) memory; a sliding-window counter approximates it cheaply; a fixed window is simplest but allows 2× bursts at window edges. For multiple API replicas, the state must be shared: Redis with a Lua script (atomic read-refill-consume) or a gateway (NGINX, Envoy, Kong). Pick: token bucket, in Redis once you have more than one process.

Follow-ups:

- "Wire it into FastAPI." A dependency that calls `allow(api_key)` and raises `HTTPException(429, headers={"Retry-After": str(math.ceil(retry_after))})`.
- "10^7 keys, most idle." Evict buckets that have refilled to capacity (they're equivalent to "no entry"), e.g. during a periodic sweep or an LRU bound.
- "Why a monotonic clock?" Wall-clock adjustments backwards would freeze refills; forwards would grant free tokens.

Scoring: Strong: lazy refill, cap at capacity, lock around read-modify-write including the clock read, Retry-After, distributed story. Acceptable: correct single-threaded, adds the lock when asked. Red flag: background thread per key, or a check-then-act race outside the lock.

BE-C02 · Idempotent order-creation endpoint with dependency injection

Hard · Coding · BE, FS · Round 3 · FastAPI dependency injection, status codes, idempotency, testing · 30 min

Problem: `POST /orders` creates an order from `{"sku": str, "quantity": int (1–100)}`. Clients retry on timeouts, so the endpoint requires an `Idempotency-Key` header (8–64 chars). Rules:

- first request with a key → `201 Created` with the new order;
- the same key and the same payload → `200 OK` with the *same* order (no duplicate);
- the same key with a different payload → `409 Conflict`;
- a missing or invalid key or body → `422`.

The store is injected via a dependency so tests can replace it, and it must be safe under concurrent requests.

Signature: `@app.post("/orders") def create_order(order: OrderIn, response: Response, store: Annotated[OrderStore, Depends(get_store)], idempotency_key: Annotated[str, Header(min_length=8, max_length=64)]) -> OrderOut`

Constraints: FastAPI + Pydantic v2; in-memory store for the exercise.

Examples:

- key `k-000001`, `{"sku": "TV-55", "quantity": 1}` → 201, `order_id="ord_1"`
- same again → 200, `order_id="ord_1"`
- same key, quantity 2 → 409

Hints:

1. "Where does the check-then-create race live, and what protects it?"
2. "How does FastAPI let you change the status code of a successful response?"
3. "How do you swap the store in tests without patching?"

Reference solution:

```python
import itertools
import threading
from typing import Annotated

from fastapi import Depends, FastAPI, Header, HTTPException, Response, status
from pydantic import BaseModel, Field


class OrderIn(BaseModel):
    sku: str = Field(min_length=1, max_length=64)
    quantity: int = Field(gt=0, le=100)


class OrderOut(BaseModel):
    order_id: str
    sku: str
    quantity: int


class IdempotencyConflict(Exception):
    pass


class OrderStore:
    """In-memory store. In production: a table with UNIQUE(idempotency_key)."""

    def __init__(self) -> None:
        self._lock = threading.Lock()
        self._ids = itertools.count(1)
        self._by_key: dict[str, tuple[OrderIn, OrderOut]] = {}

    def create_or_get(self, key: str, order: OrderIn) -> tuple[OrderOut, bool]:
```

```
        """Return (order, created). Atomic check-and-insert per key."""
        with self._lock:
            if key in self._by_key:
                original, existing = self._by_key[key]
                if original != order:
                    raise IdempotencyConflict(key)
                return existing, False
            created = OrderOut(order_id=f"ord_{next(self._ids)}", **order.model_dump())
            self._by_key[key] = (order, created)
            return created, True


_store = OrderStore()


def get_store() -> OrderStore:
    return _store


app = FastAPI()


@app.post("/orders", response_model=OrderOut, status_code=status.HTTP_201_CREATED)
def create_order(
    order: OrderIn,
    response: Response,
    store: Annotated[OrderStore, Depends(get_store)],
    idempotency_key: Annotated[str, Header(min_length=8, max_length=64)],
) -> OrderOut:
    try:
        result, created = store.create_or_get(idempotency_key, order)
    except IdempotencyConflict:
        raise HTTPException(status.HTTP_409_CONFLICT,
                            "Idempotency-Key reused with a different payload") from None
    if not created:
        response.status_code = status.HTTP_200_OK
    return result
```

Complexity: O(1) per request; O(stored keys) memory (expire keys after 24–48 h in production).

Tests:

```
from concurrent.futures import ThreadPoolExecutor
from fastapi.testclient import TestClient

test_store = OrderStore()
app.dependency_overrides[get_store] = lambda: test_store
client = TestClient(app)
body = {"sku": "TV-55", "quantity": 1}

r1 = client.post("/orders", json=body, headers={"Idempotency-Key": "k-000001"})
assert r1.status_code == 201 and r1.json() == {"order_id": "ord_1", "sku": "TV-55", "quantity":
1}
r2 = client.post("/orders", json=body, headers={"Idempotency-Key": "k-000001"})
assert r2.status_code == 200 and r2.json() == r1.json()
r3 = client.post("/orders", json={**body, "quantity": 2}, headers={"Idempotency-Key": "k-
000001"})
assert r3.status_code == 409
assert client.post("/orders", json=body).status_code == 422                          # header
missing
assert client.post("/orders", json=body, headers={"Idempotency-Key": "short"}).status_code ==
422
assert client.post("/orders", json={"sku": "TV-55", "quantity": 0},
                   headers={"Idempotency-Key": "k-000002"}).status_code == 422
assert client.post("/orders", json={"sku": "TV-55"},
                   headers={"Idempotency-Key": "k-000003"}).status_code == 422
```

```python
def fire(_: int) -> tuple[int, str]:
    r = client.post("/orders", json={"sku": "PHONE", "quantity": 3},
                    headers={"Idempotency-Key": "k-concurrent"})
    return r.status_code, r.json()["order_id"]
with ThreadPoolExecutor(max_workers=8) as pool:
    outcomes = list(pool.map(fire, range(16)))
assert sorted(code for code, _ in outcomes) == [200] * 15 + [201]
assert len({oid for _, oid in outcomes}) == 1                                    # one
order only
assert _store._by_key == {}                                                      #
override used
app.dependency_overrides.clear()
```

Alternatives and tradeoffs: Database-enforced idempotency: `INSERT ... ON CONFLICT (idempotency_key) DO NOTHING RETURNING ...`, then `SELECT` on conflict. Survives restarts and works across replicas, which the in-memory store doesn't. Stripe-style: store the full response (status + body) keyed by idempotency key and replay it verbatim. Pick: DB unique constraint + stored response in production; the lock-based store here demonstrates the semantics and is enough for tests.

Follow-ups:

- "A request with the key is still in progress when the retry arrives." Store a "pending" marker and return `409` or `425 Too Early` (or block briefly) until the first completes.
- "Why 201 vs 200 vs 409 vs 422?" 201 = resource created; 200 = replay of an existing result; 409 = conflict with current state; 422 = semantically invalid input (FastAPI's default for validation errors; some APIs use 400).
- "Flask/Django equivalent of `Depends`?" Flask: app factories and extensions via `current_app`/`g`; Django: middleware or service objects; testing then relies more on `mock.patch` (M-O06).

Scoring: Strong: atomic check-and-insert, correct status codes, payload comparison for 409, DI override in tests, DB-backed production story. Acceptable: works single-threaded, adds the lock when asked about concurrency. Red flag: no idempotency awareness, or `mock.patch` of module globals as the only testing strategy.

BE red flags: blocking calls inside `async def` handlers; catches `Exception` and returns 200 with an error message; hand-rolled password hashing or crypto (should reach for `bcrypt`/`argon2-cffi`/`passlib` and `secrets`); retries non-idempotent writes without keys; logs secrets or full request bodies; patches where the function is defined instead of where it's looked up.

FS · Python Full Stack Engineer

Calibration (3–4 years): Owns a feature end to end: everything BE expects, plus anticipating what the frontend does with the API's output: JSON types that don't round-trip, timestamps that must be UTC ISO 8601, pagination that stays correct while data changes, and output that's safe to render.

Must-ask master-bank IDs: E-O01 (formatting), ET-O01 (money rounding), ET-O02 (string methods), ET-C02 (Unicode normalisation), M-O06 (patching), MT-O02 (mutable defaults), H-O02 (blocking the loop), HT-O04 (TaskGroup).

FS-O01 · json.dumps on datetimes, Decimals, sets and NaN

Medium · Output · FS, BE · Round 3 · JSON serialisation edge cases · 6 min

Question: What does this print?

```python
import json
from datetime import datetime, timezone
from decimal import Decimal

print(json.dumps({"ok": True, "missing": None, "ratio": float("nan")}))
for value in (datetime(2026, 9, 24, tzinfo=timezone.utc), Decimal("19.99"), {"a"}):
    try:
        json.dumps(value)
    except TypeError as exc:
        print("TypeError:", exc)
try:
    json.dumps(float("inf"), allow_nan=False)
except ValueError as exc:
    print("ValueError:", exc)
print(json.dumps({1: "a", True: "b"}), json.dumps({"price": 1e16, "id": 12345678901234567890}))
print(json.dumps({"name": "Café"}), json.dumps({"name": "Café"}, ensure_ascii=False))
```

Expected output:

```
{"ok": true, "missing": null, "ratio": NaN}
TypeError: Object of type datetime is not JSON serializable
TypeError: Object of type Decimal is not JSON serializable
TypeError: Object of type set is not JSON serializable
ValueError: Out of range float values are not JSON compliant: inf
{"1": "b"} {"price": 1e+16, "id": 12345678901234567890}
{"name": "Caf\u00e9"} {"name": "Café"}
```

(The ValueError message wording varies across versions; the stable part is "Out of range float values are not JSON compliant".)

Explanation: `json.dumps` maps `True`/`None` to `true`/`null`, but by default it also emits `NaN`/`Infinity`, which are **not valid JSON**, so browsers' `JSON.parse` throws on them; `allow_nan=False` makes Python raise instead. `datetime`, `Decimal`, and `set` have no JSON representation, so you must choose one (ISO 8601 string, string or number, list). In the dict literal `{1: "a", True: "b"}`, `True == 1` and `hash(True) == hash(1)`, so they are the *same key*; the later value wins, and JSON object keys are always strings. Python writes big ints exactly, but JavaScript numbers are doubles, so `12345678901234567890` loses precision in the browser (anything above 2^53). `ensure_ascii=True` (default) escapes non-ASCII as `\u00e9`, which is valid but bulky. Takeaway: define the wire format explicitly: ISO 8601 UTC strings, Decimals as strings, IDs above 2^53 as strings, and no NaN.

Follow-ups:

1. "How does FastAPI handle these?" `jsonable_encoder`/Pydantic serialise datetimes to ISO 8601 and Decimals per the field type; you still decide the contract for NaN and large IDs.
2. "Custom encoder?" `json.dumps(obj, default=fn)` where `fn` converts unknown types, or subclass `JSONEncoder`; `orjson` handles datetime and dataclasses natively and is much faster.
3. "A float price like 0.1 + 0.2?" Serialises as `0.30000000000000004`; send money as integer minor units or decimal strings.

Scoring: Strong: all seven lines, and explains NaN validity and the 2^53 limit. Acceptable: knows the TypeErrors, misses `True == 1` or NaN. Red flag: thinks JSON supports dates.

FS-O02 · Offset pagination under concurrent inserts

Medium · Output · FS, BE · Round 3 · offset vs cursor pagination · 6 min

Question: A feed shows newest posts first, 3 per page. What does this print?

```python
def page_offset(ids, page, size=3):
    return ids[(page - 1) * size : page * size]

def page_cursor(ids, before=None, size=3):
    return [i for i in ids if before is None or i < before][:size]

posts = list(range(10, 0, -1))                  # post ids, newest first
p1 = page_offset(posts, 1)
posts.insert(0, 11)                             # new post while user reads page 1
print(p1, page_offset(posts, 2))

posts = list(range(10, 0, -1))
p1 = page_offset(posts, 1)
posts.remove(9)                                 # a post on page 1 is deleted
print(p1, page_offset(posts, 2))

posts = list(range(10, 0, -1))
c1 = page_cursor(posts)
posts.insert(0, 11)
posts.remove(9)
print(c1, page_cursor(posts, before=c1[-1]))
```

Expected output:

```
[10, 9, 8] [8, 7, 6]
[10, 9, 8] [6, 5, 4]
[10, 9, 8] [7, 6, 5]
```

Explanation: Offset pagination addresses rows by *position*. A new post at the top shifts every row down one, so page 2 starts at what used to be position 2: post 8 is shown twice. A deletion shifts rows up, so post 7 falls into the gap and is never shown. Cursor (keyset) pagination addresses rows by *value*: "give me posts older than the last one I saw" (`id < 8`), which is unaffected by inserts above or deletes elsewhere. In SQL that's `WHERE (created_at, id) < (:t, :id) ORDER BY created_at DESC, id DESC LIMIT n`, served by an index instead of scanning and discarding `OFFSET` rows. Takeaway: use cursor pagination for feeds and anything that changes while being read; offset pagination only for small, static result sets or "jump to page N" UIs.

Follow-ups:

1. "Why include `id` in the cursor, not just `created_at`?" Timestamps collide; the tie-breaker makes the ordering total so no row is skipped or repeated (FS-C01).
2. "What does the UI lose with cursors?" Jumping to an arbitrary page number and showing total pages cheaply.
3. "Performance of `OFFSET 100000`?" The database still reads and discards 100,000 rows; keyset pagination is O(page size) with an index.

Scoring: Strong: all three lines and explains position vs value addressing. Acceptable: sees the duplicate, misses the skip on delete. Red flag: "just use offset, duplicates are rare".

FS-O03 · Template autoescaping and XSS

Medium · Output · FS · Round 3 · Jinja2 autoescaping, `Markup`, XSS contexts · 6 min

Question: What does this print?

```python
from jinja2 import Environment
```

```python
from markupsafe import Markup, escape

comment = '<script>alert("pwned")</script> & more'
raw_env = Environment()
safe_env = Environment(autoescape=True)
print(raw_env.from_string("<p>{{ c }}</p>").render(c=comment))
print(safe_env.from_string("<p>{{ c }}</p>").render(c=comment))
print(safe_env.from_string("<p>{{ c|safe }}</p>").render(c=comment))
print(safe_env.from_string("<a title='{{ t }}'>x</a>").render(t="' onmouseover='steal()"))
print(safe_env.from_string("<a href='{{ u }}'>profile</a>").render(u="javascript:alert(1)"))
print(escape("<b>") + Markup("<i>ok</i>"))
```

Expected output:

```
<p><script>alert("pwned")</script> & more</p>
<p>&lt;script&gt;alert(&#34;pwned&#34;)&lt;/script&gt; &amp; more</p>
<p><script>alert("pwned")</script> & more</p>
<a title='&#39; onmouseover=&#39;steal()'>x</a>
<a href='javascript:alert(1)'>profile</a>
&lt;b&gt;<i>ok</i>
```

Explanation: A bare `jinja2.Environment()` does **not** autoescape, so user input is rendered as live HTML (stored XSS). With autoescaping, `<`, `>`, `&`, `"` and `'` become entities, which also neutralises attribute break-outs like the `title` injection. `|safe` (or wrapping in `Markup`) tells the engine "this is already trusted HTML" and disables escaping for that value; using it on user input reintroduces XSS. Autoescaping is HTML-*text* escaping only: it does nothing about dangerous URL schemes, so `javascript:alert(1)` in an `href` still executes on click; URLs need scheme allow-listing (`http/https/mailto`). `escape()` returns `Markup`, and `Markup + Markup` concatenation doesn't double-escape. Takeaway: autoescape on by default (Flask and Django do this for HTML templates), never `|safe` user data, validate URLs, and add a Content-Security-Policy as defence in depth.

Follow-ups:

1. "Values inserted into a `<script>` block?" Use `|tojson` in Jinja (Flask) or `json_script` in Django; HTML escaping is the wrong encoding for JavaScript context.
2. "The frontend is React and the API returns JSON. Is XSS gone?" Mostly, since React escapes text, unless someone uses `dangerouslySetInnerHTML` or puts user URLs in `href`. Validate server-side regardless.
3. "User-supplied Markdown?" Render, then sanitise the HTML with an allow-list sanitiser (e.g. `nh3`/`bleach`) before marking it safe.

Scoring: Strong: all six lines, and flags the `javascript:` URL as still dangerous. Acceptable: knows autoescape, misses the URL-context gap. Red flag: `|safe` as the fix for "HTML entities showing up in the page".

FS-C01 · Cursor-paginated feed endpoint

Hard · Coding · FS, BE · Round 3 · keyset pagination, opaque cursors, UTC ISO 8601, FastAPI · 35 min

Problem: Implement `GET /feed?limit=20&cursor=...` returning `{"items": [...], "next_cursor": str | null}`.

- Order: `created_at` descending, then `id` descending (a total order; timestamps can collide).
- The cursor is opaque to clients: URL-safe base64 of the last item's `(created_at, id)`.
- A malformed cursor → `400`; `limit` outside 1–100 → `422`.
- Timestamps must be timezone-aware on input and always serialised as UTC ISO 8601 with `Z`, whatever offset they were stored with.
- Inserting new posts between page fetches must not cause duplicates or skips.

Signature: `@app.get("/feed", response_model=FeedPage) def feed(posts: Annotated[list[Post], Depends(get_posts)], limit: Annotated[int, Query(ge=1, le=100)] = 20, cursor: str | None = None) -> FeedPage`

Constraints: In-memory list for the exercise (production: an indexed query). Post IDs are unique ints.

Examples:

- 5 posts, `limit=2` → pages of 2, 2, 1; the last page has `next_cursor: null`
- A post stored as `2026-09-24T15:30:00+05:30` → serialised `"2026-09-24T10:00:00Z"`
- `cursor=not-base64!!` → 400

Hints:

1. "Why does ordering by `created_at` alone break pagination?"
2. "How do you know whether there's a next page without a COUNT query?" (fetch `limit + 1`)
3. "Which Pydantic type rejects naive datetimes?"

Reference solution:

```python
import base64
import binascii
import json
from datetime import datetime, timezone
from typing import Annotated

from fastapi import Depends, FastAPI, HTTPException, Query
from pydantic import AwareDatetime, BaseModel, field_serializer


class Post(BaseModel):
    id: int
    created_at: AwareDatetime                    # naive timestamps rejected at the boundary
    text: str

    @field_serializer("created_at")
    def _as_utc(self, value: datetime) -> str:
        return value.astimezone(timezone.utc).isoformat().replace("+00:00", "Z")


class FeedPage(BaseModel):
    items: list[Post]
    next_cursor: str | None


def encode_cursor(post: Post) -> str:
    payload = {"t": post.created_at.astimezone(timezone.utc).isoformat(), "id": post.id}
    raw = json.dumps(payload, separators=(",", ":")).encode()
    return base64.urlsafe_b64encode(raw).decode().rstrip("=")


def decode_cursor(cursor: str) -> tuple[datetime, int]:
    try:
        raw = base64.urlsafe_b64decode(cursor + "=" * (-len(cursor) % 4))
        payload = json.loads(raw)
        created_at = datetime.fromisoformat(payload["t"])
        post_id = int(payload["id"])
    except (binascii.Error, ValueError, KeyError, TypeError, UnicodeDecodeError) as exc:
        raise ValueError("malformed cursor") from exc
    if created_at.tzinfo is None:
        raise ValueError("cursor timestamp must be timezone-aware")
    return created_at, post_id


POSTS: list[Post] = []


def get_posts() -> list[Post]:
    return POSTS


app = FastAPI()
```

```python
@app.get("/feed", response_model=FeedPage)
def feed(
    posts: Annotated[list[Post], Depends(get_posts)],
    limit: Annotated[int, Query(ge=1, le=100)] = 20,
    cursor: str | None = None,
) -> FeedPage:
    ordered = sorted(posts, key=lambda p: (p.created_at, p.id), reverse=True)
    if cursor is not None:
        try:
            bound = decode_cursor(cursor)
        except ValueError:
            raise HTTPException(status_code=400, detail="invalid cursor") from None
        ordered = [p for p in ordered if (p.created_at, p.id) < bound]
    page = ordered[: limit + 1]                    # one extra row tells us if there's more
    items = page[:limit]
    next_cursor = encode_cursor(items[-1]) if len(page) > limit else None
    return FeedPage(items=items, next_cursor=next_cursor)
```

Complexity: O(n log n) per request here (sort + filter over the in-memory list); in production O(limit) with an index on `(created_at DESC, id DESC)`.

Tests:

```python
from fastapi.testclient import TestClient

def make(pid: int, ts: str) -> Post:
    return Post(id=pid, created_at=datetime.fromisoformat(ts), text=f"post {pid}")

data = [make(1, "2026-09-24T09:00:00Z"), make(2, "2026-09-24T10:00:00Z"),
        make(3, "2026-09-24T15:30:00+05:30"),                        # also 10:00Z: tie with id
2
        make(4, "2026-09-24T11:00:00Z"), make(5, "2026-09-24T12:00:00Z")]
app.dependency_overrides[get_posts] = lambda: data
client = TestClient(app)

seen, cursor, pages = [], None, 0
while True:
    params = {"limit": 2} | ({"cursor": cursor} if cursor else {})
    body = client.get("/feed", params=params).json()
    seen += [p["id"] for p in body["items"]]
    pages += 1
    if pages == 1:
        data.append(make(6, "2026-09-24T13:00:00Z"))                 # new post after page 1
    cursor = body["next_cursor"]
    if cursor is None:
        break
assert seen == [5, 4, 3, 2, 1] and pages == 3                         # no dupes, no skips

first = client.get("/feed", params={"limit": 10}).json()
assert [p["id"] for p in first["items"]] == [6, 5, 4, 3, 2, 1] and first["next_cursor"] is None
assert next(p for p in first["items"] if p["id"] == 3)["created_at"] == "2026-09-24T10:00:00Z"

assert client.get("/feed", params={"cursor": "not-base64!!"}).status_code == 400
bad = base64.urlsafe_b64encode(b'{"t":"2026-09-24T10:00:00","id":1}').decode()
assert client.get("/feed", params={"cursor": bad}).status_code == 400   # naive cursor
assert client.get("/feed", params={"limit": 0}).status_code == 422
assert client.get("/feed", params={"limit": 101}).status_code == 422
app.dependency_overrides[get_posts] = lambda: []
assert client.get("/feed").json() == {"items": [], "next_cursor": None}
try:
    Post(id=9, created_at=datetime(2026, 1, 1), text="naive")
    raise AssertionError("naive datetime accepted")
except ValueError:
    pass
app.dependency_overrides.clear()
```

Alternatives and tradeoffs: Offset pagination is simpler and supports page numbers, but duplicates and skips under writes (FS-O02) and gets slower with depth. A signed cursor (HMAC) stops clients from forging cursors, which matters if the cursor embeds filters or tenant IDs. Encoding only the last `id` works when IDs are monotonic with time (e.g. Snowflake/ULID IDs), which simplifies the cursor. Pick: keyset pagination with an opaque (optionally signed) cursor for feeds; offset only for admin tables.

Follow-ups:

- "The SQL for this?" `WHERE (created_at, id) < (:t, :id) ORDER BY created_at DESC, id DESC LIMIT :limit + 1` with a composite index; row-value comparison works in Postgres and SQLite (3.15+).
- "Bidirectional pagination (prev page)?" A second cursor with reversed comparison and ordering, then reverse the page before returning.
- "The frontend shows 'posted 5 hours ago' in the user's timezone." The API sends UTC ISO 8601; the browser converts with `Intl.DateTimeFormat`. Never send pre-formatted local times.

Scoring: Strong: total ordering with tie-breaker, `limit + 1`, opaque validated cursor, 400 vs 422 distinction, UTC `Z` serialisation, aware-only timestamps. Acceptable: keyset works but orders by timestamp alone, fixed after hint 1. Red flag: offset pagination, or a cursor that is the raw page number.

FS red flags: returns datetimes without offsets or in server-local time; floats for money in JSON; offset pagination for infinite scroll; `|safe`/`dangerouslySetInnerHTML` on user input; validation only on the client; CORS `allow_origins=["*"]` combined with credentials; unbounded file uploads (no size limit or content-type validation, trusting the client-supplied filename).

Part 5 · Round-wise section

Round definitions

Round	Duration	Purpose	Mix	Pass bar (3–4 years)
R1 · Screening	30–40 min	Fluency; baseline mental model of names, objects, control flow	4–5 output questions from E and ET, plus 1 E coding task	Clears most E; reasons sensibly through ET even when falling for a trap
R2 · Core Python deep dive	60 min	Depth in the language: OOP, functional tools, iterators, decorators, exceptions	4 output questions from M and MT, plus 1–2 M/MT coding tasks	Clears M; at least half of MT with correct reasoning
R3 · Role practical	60–75 min	Can they do the actual job?	2 role-exclusive output questions + 1 role-exclusive coding task (M or H)	Working solution, handles edge cases, discusses tradeoffs unprompted
R4 · Advanced & production	60 min	Seniority signal: concurrency, performance, internals, debugging	2–3 questions from H and HT + 1 code-review exercise + design follow-ups ("how does this behave at 100× data?")	Solid on H; partial HT credit is a strong signal

Theory questions (Part 6) slot into any round as warm-ups, bridges between coding questions, or quick depth probes: roughly 2–4 per round, matched to the round's tier.

Role × round matrix

Primary picks are listed first; items in *(alt: …)* are drop-in substitutes of the same tier and length, for re-interviews or when a candidate has clearly seen a question before.

Round	DE	DS	MLE	BE	FS
R1 output	E-O02, E-O06, ET-O01, ET-O03, ET-O05 *(alt: E-O03)*	E-O05, E-O06, ET-O01, ET-O03, ET-O06 *(alt: E-O02)*	E-O01, E-O06, ET-O03, ET-O04, ET-O06 *(alt: E-O04)*	E-O02, E-O03, ET-O02, ET-O04, ET-O06 *(alt: ET-O05)*	E-O01, E-O02, ET-O01, ET-O02, ET-O04 *(alt: E-O04)*
R1 coding	E-C01 *(alt: ET-C01)*	E-C02 *(alt: ET-C03)*	E-C01 *(alt: E-C02)*	E-C03 *(alt: ET-C01)*	E-C03 *(alt: ET-C02)*
R1 theory	TH-GEN-01, TH-NUM-02	TH-NUM-01, TH-SRT-02	TH-HSH-01, TH-TYP-02	TH-EXC-01, TH-MUT-02	TH-STR-01, TH-HSH-02
R2 output	M-O02, M-O04, MT-O04, MT-O05 *(alt: MT-O06)*	M-O03, M-O05, MT-O01, MT-O05 *(alt: MT-O04)*	M-O01, M-O05, MT-O02, MT-O05 *(alt: MT-O01)*	M-O01, M-O06, MT-O02, MT-O06 *(alt: MT-O03)*	M-O03, M-O06, MT-O01, MT-O03 *(alt: MT-O02)*
R2 coding	M-C03 + MT-C02 *(alt: M-C02)*	M-C02 *(alt: MT-C01)*	MT-C01 *(alt: M-C01)*	M-C01 *(alt: MT-C03)*	M-C01 *(alt: MT-C03)*
R2 theory	TH-CTX-03, TH-COL-04	TH-CLO-04, TH-OOP-03	TH-DEC-03, TH-FN-04	TH-TST-03, TH-SCO-04	TH-MOD-03, TH-EXC-04
R3 output	DE-O01, DE-O02	DS-O01, DS-O02 *(alt: DS-O03)*	MLE-O01, MLE-O02 *(alt: DS-O01)*	BE-O01, BE-O02	FS-O01, FS-O02 *(alt: FS-O03)*
R3 coding	DE-C01 *(alt: DE-C02)*	DS-C01 *(alt: MLE-C02)*	MLE-C01 *(alt: MLE-C02)*	BE-C02 *(alt: BE-C01)*	FS-C01 *(alt: BE-C02)*
R3 theory	TH-DE-03, TH-DE-06	TH-DS-03, TH-DS-05	TH-MLE-03, TH-MLE-06	TH-BE-03, TH-BE-06	TH-FS-03, TH-FS-06

R4 H/HT	H-O01, H-O05, HT-O05 *(alt: H-O06)*	H-O03, H-O05, HT-O02 *(alt: HT-O06)*	H-O01, H-O02, HT-O02 *(alt: HT-O03)*	H-O02, H-O06, HT-O04 *(alt: HT-O01)*	H-O02, HT-O04, HT-O06 *(alt: H-O04)*
R4 code review	CR-C (H-C03)	CR-A (MT-C01)	CR-B (H-C02)	CR-D (H-C01)	CR-E (HT-C02)
R4 design follow-ups	H-C03 or HT-C01 at 100× data	HT-C03 memory at scale	H-C01 / HT-C03 for serving	HT-C01 graceful shutdown	HT-C02 across replicas
R4 theory	TH-CON-05, TH-MEM-06	TH-MEM-05, TH-GEN-06	TH-ASY-05, TH-CON-06	TH-ASY-06, TH-DM-05	TH-ASY-05, TH-OOP-06

Round 4 code-review exercises

Hand the candidate the reference solution with the bug injected (don't say where), and give them 10–12 minutes to review it as if it were a pull request. Every injected bug below was applied to the verified reference solution and checked against its tests. Two of the five still pass the tests, which is deliberate: those check whether the candidate reviews beyond "the tests are green".

ID	Base	Injected bug	Tests	What a strong reviewer says
CR-A	MT-C01	Cache hit returns `cache[key]` instead of `copy.deepcopy(cache[key])`	Fail	"Callers get the stored object; one caller's mutation poisons every later hit." Suggests defensive copies or immutable results.
CR-B	H-C02	`func(*args, **kwargs)` moved *inside* the first `with lock:` block	**Pass**	"The lock is held during the slow call: one slow key serialises every cache hit and miss in the process." Also asks about stampedes and the clock.
CR-C	H-C03	Newline normalisation line removed from run creation	Fail	"A final line without `\n` gets glued to the next line after merging." Asks for an edge-case test with no trailing newline.
CR-D	H-C01	`except Exception` changed to `except BaseException`	**Pass**	"This captures `CancelledError/KeyboardInterrupt` as per-item results: cancellation and shutdown semantics break." Proposes a test where `fetch` raises a `BaseException` subclass and must propagate.
CR-E	HT-C02	`raise` removed after the nested rollback	Fail	"Exceptions are suppressed: the caller's code continues and the outer transaction commits partial work." Connects it to M-O02.

Scoring for code review: Strong: finds the bug, explains the production consequence, proposes the test that would have caught it. Acceptable: finds it with a pointer to the right function. Red flag: comments only on style or naming, or trusts green tests as proof of correctness.

Interviewer guide

Hint policy

- **When to hint:** after about one-third of the question's time budget with no forward progress, or when the candidate is confidently heading down a wrong path that will consume the rest of the slot. Don't hint while they're productively reasoning, even if slowly.
- **How:** use the graded hints in order, one at a time; wait for the candidate to act on each before giving the next. For output questions, the equivalent hint is "trace it line by line out loud" and then naming the mechanism area ("think about when defaults are evaluated").
- **Cost:** each hint caps the achievable score for that question: no hints → up to 4; one hint → up to 3; two hints → up to 3 if they then finish independently, otherwise 2; three hints or a walked-through solution → up to 2. A hint never lowers the score below what the candidate's reasoning earned.

Scoring scale (per question)

Score	Meaning
4 · Strong	Correct, explains the mechanism unprompted, raises tradeoffs or edge cases
3 · Acceptable	Correct or nearly so; explanation needs a nudge; minor slips corrected independently
2 · Weak	Partially correct, relies heavily on hints, or right answer with a wrong explanation
1 · Red flag	Wrong with no recovery, or shows a misconception that would cause production bugs

Round score = weighted average of question scores. Weights by tier: E 1.0, ET 1.0, M 1.5, MT 1.5, H 2.0, HT 2.0, role-exclusive 2.0 (or its stated tier's weight, whichever is higher). Theory questions are scored on the same 1–4 scale at weight 0.5; they calibrate, they don't decide.

Overall calibration for 3–4 years

Expected: clears all E and M, most ET, about half of MT, and at least one H. HT questions are differentiators, not requirements: a hire can score 2 on every HT; a 3–4 on even one HT, with sound reasoning, is a strong seniority signal. In Round 3, the coding task matters more than the output questions: a working, tested solution with edge cases beats perfect trivia recall.

When output and reasoning disagree

- **Wrong output, good reasoning:** score on the reasoning. If the candidate names the right mechanism and their error is a tracing slip (miscounted an index, forgot one print), give 3; if they reason to the correct mechanism *after* seeing the output and explain it cleanly, give 3 on M/MT and 2 on E/ET (fluency questions are supposed to be fast). This is the most hireable pattern in the bank: it's how good engineers debug.
- **Right output, can't explain:** probe once ("why?", then "what would change if…?" using a follow-up). If they still can't explain, cap at 2: a memorised answer to a classic is exactly what the disguised scenarios are designed to detect. If the follow-up variant is also correct, they understand more than they can articulate; give 3 and note communication as a separate signal.
- **Right output, wrong explanation:** score 2 and record the misconception. It will cause a bug in a slightly different situation.

Running the loop

- Keep a per-candidate log of question IDs used, so re-interviews use the *(alt: …)* column.
- Don't reuse the worked examples from the original brief (ET-O00, M-C00); they're public within your team.
- Calibrate panels quarterly: have two interviewers independently score the same recorded answers for a handful of IDs and compare.

Part 6 · Theory and technical questions

Verbal questions for warm-ups, bridges between coding questions, and fast depth probes. Every core topic and every role has one question per tier, so an interviewer can pick by category and difficulty:

Suffix	Tier	Use it for
-01	Easy (E)	Round 1 warm-up; one concept, one or two sentences expected
-02	Easy & Tricky (ET)	Round 1; the common wrong answer is listed as the trap
-03	Medium (M)	Round 2; expects mechanism plus a practical consequence
-04	Medium & Tricky (MT)	Round 2; two mechanisms interact
-05	Hard (H)	Round 4 (or Round 3 for role questions); internals, design, tradeoffs
-06	Hard & Tricky (HT)	Round 4 differentiator; partial answers are fine

Scoring uses the same 1–4 scale as the rest of the bank at weight 0.5 (see Part 5). The answers below are the key points to listen for, not a script: a candidate who reaches them in their own words gets full credit. Behaviour that the answers state as fact was checked on CPython 3.12.

Core Python topics

TH-DM · Data model and object identity

TH-DM-01 · Easy · *What three things does every Python object have, and what is a variable?*

> Every object has an identity (`id()`, fixed for its lifetime), a type (which decides what operations it supports), and a value. A variable is a *name bound to an object*, not a box holding a value; several names can refer to the same object, and assignment only rebinds names.

TH-DM-02 · Easy & Tricky · *Does `x is y` always imply `x == y`?*

> **Trap:** "Yes, the same object must equal itself."
>
> Not always. `==` calls `__eq__`, which can return anything: `nan = float("nan")` gives `nan is nan` True but `nan == nan` False. Containers check identity before equality, so `nan in [nan]` is True for the same object even though the comparison is False. (NumPy arrays return element-wise results from `==`, which is another reason `is` and `==` are different questions.)

TH-DM-03 · Medium · *What's the difference between `__repr__` and `__str__`, and which one do containers use?*

> `__repr__` is the unambiguous developer representation (ideally something that could recreate the object); `__str__` is the user-facing text. `print(obj)` and `f"{obj}"` use `str`; `f"{obj!r}"`, the REPL, and **containers printing their elements** use `repr` (hence the quotes in `['a']`). If only `__repr__` is defined, `str()` falls back to it, so define `__repr__` first.

TH-DM-04 · Medium & Tricky · *Are Python arguments passed by value or by reference?*

> **Trap:** "By reference for lists, by value for ints."
>
> Neither: it's call by object reference (call by sharing). The parameter is a new name bound to the caller's object. Mutating that object is visible to the caller; *rebinding* the parameter is not. Ints only look "by value" because they're immutable: `n += 1` rebinds the local name to a new int.

TH-DM-05 · Hard · *How does Python evaluate `a + b` when `a` and `b` have different types?*

> It calls `type(a).__add__(a, b)`. If that returns `NotImplemented`, it tries `type(b).__radd__(b, a)`. If `b`'s type is a subclass of `a`'s type that overrides `__radd__`, the reflected method is tried *first*. If both return `NotImplemented`, TypeError. So operator methods should `return NotImplemented` for unknown types rather than raise, or they block the other operand's chance to handle it.

TH-DM-06 · Hard & Tricky · *Can two different objects have the same `id()`?*

> **Trap:** "No, ids are unique forever."
>
> `id()` is unique only among objects alive *at the same time*. In CPython it's the memory address, which is reused after an object is freed, so `id([]) == id([])` can be True: the first list dies before the second is allocated. Storing `id()`s of temporary objects as keys (e.g. in a cache or "seen" set) is a bug; keep a reference or use `weakref.finalize` to clean up (see HT-C03).

TH-MUT · Mutability, aliasing and copying

TH-MUT-01 · Easy · *Which built-in types are mutable, and why does mutability matter for dict keys?*

> Immutable: `int`, `float`, `complex`, `bool`, `str`, `bytes`, `tuple`, `frozenset`, `None`. Mutable: `list`, `dict`, `set`, `bytearray`, most user classes. Dict keys and set members must be hashable, and a hash must not change while the object is stored, which in practice means immutable values.

TH-MUT-02 · Easy & Tricky · *`orders = orders.sort()`: what's in `orders` now?*

> **Trap:** "The sorted list."
>
> `None`. By convention, methods that mutate in place return `None` (`list.sort`, `list.append`, `list.reverse`, `dict.update`, `random.shuffle`), precisely so you can't confuse them with functions that return a new object (`sorted()`, `reversed()`, `a | b`).

TH-MUT-03 · Medium · *Shallow vs deep copy: how do you make each, and when do you need a deep copy?*

> Shallow: `copy.copy(x)`, `list(x)`, `x[:]`, `x.copy()`, `{**d}`. A new outer container shares the inner objects. Deep: `copy.deepcopy(x)` recursively copies, handles cycles via a memo dict, and can be customised with `__deepcopy__`. You need deep copies when nested mutable parts must be independent (configs, nested dicts from JSON, state snapshots). It's slow on large structures, so prefer immutable data or rebuilding only the parts you change.

TH-MUT-04 · Medium & Tricky · *`grid = [[0] * 3] * 3; grid[0][0] = 1`. What is `grid`?*

> **Trap:** "Only the first row changes."
>
> `[[1, 0, 0], [1, 0, 0], [1, 0, 0]]`. The inner `[0] * 3` is fine (ints are immutable), but the outer `* 3` repeats a *reference* to the same inner list three times. Fix: `[[0] * 3 for _ in range(3)]`, which evaluates a new inner list per iteration.

TH-MUT-05 · Hard · *How do you make instances effectively immutable, and what does it cost?*

> Options: `@dataclass(frozen=True)` (assignment raises `FrozenInstanceError`; `__post_init__` must use `object.__setattr__`), `typing.NamedTuple`, `__slots__` with read-only properties, `types.MappingProxyType` for read-only dict views. Immutability must be deep to mean anything: a frozen dataclass holding a list is still mutable through the list. Benefits: hashable, safe to share across threads and as defaults, easy reasoning. Cost: "changes" create new objects (`dataclasses.replace`), which matters in hot loops.

TH-MUT-06 · Hard & Tricky · `t = ([1, 2],); t[0] += [3]`. *What happens?*

> **Trap:** "Either it works, or it fails and nothing changes."
>
> Both happen. `t[0] += [3]` runs `t[0].__iadd__([3])`, which extends the list in place and returns it, *then* attempts `t[0] = result`, which raises `TypeError: 'tuple' object does not support item assignment`. After the exception, `t` is `([1, 2, 3],)`. It's a good probe of whether someone understands augmented assignment as "compute, then store".

TH-SCO · Scoping (LEGB, global, nonlocal)

TH-SCO-01 · Easy · *What does LEGB stand for?*

> The lookup order for reading a name: Local, Enclosing function scopes, Global (module), Builtins. Assignment inside a function creates a local name unless the name is declared `global` or `nonlocal`.

TH-SCO-02 · Easy & Tricky · *Do `if` blocks and `for` loops create a new scope?*

> **Trap:** "Yes, like in C or Java."
>
> No. Only modules, classes, functions (including lambdas) and comprehensions create scopes. A loop variable still exists after the loop with its last value, and a name assigned only inside an `if` branch that didn't run is simply unbound (NameError or UnboundLocalError when read).

TH-SCO-03 · Medium · *When do you need `global` or `nonlocal`?*

> Only when *rebinding* a name in an outer scope: `global x` for the module scope, `nonlocal x` for the nearest enclosing function scope (it can't target module scope). Mutating an outer object (`cache[key] = v`, `items.append(v)`) needs no declaration, because the name isn't rebound.

TH-SCO-04 · Medium & Tricky · *A module defines `list = load_orders()` at the top. What can break, and where?*

> **Trap:** "Only that line is affected."
>
> The module-level name `list` now shadows the builtin for the *whole module*, since the global scope is searched before builtins. Any later `list(x)` in any function of that module raises `TypeError: 'list' object is not callable`, often far from the offending line. Common culprits: `id`, `type`, `input`, `sum`, `max`, `dict`, `file`; `from module import *` can shadow names too.

TH-SCO-05 · Hard · *How do names work inside a class body?*

> The class body runs once as its own scope, and the names it defines become class attributes. Methods cannot see those names unqualified; they must use `self.name` or `ClassName.name`, because function scopes skip the class scope when resolving names. The same rule applies to comprehensions in a class body: only the outermost iterable is evaluated in class scope, so `[base + i for i in range(3)]` with class-level `base` raises NameError.

TH-SCO-06 · Hard & Tricky · *After `except ValueError as err:`, can you still use `err` after the block?*

> **Trap:** "Yes, like any assigned variable."
>
> No. Python implicitly deletes the `as` target at the end of the handler (`del err`) to break the cycle exception → traceback → frame → `err`, which would keep every local in that frame alive. Reading `err` afterwards raises NameError (UnboundLocalError in a function), and a pre-existing variable with the same name is gone too. Save it to another name inside the block if you need it later.

TH-CLO · Closures and late binding

TH-CLO-01 · Easy · *What is a closure?*

> An inner function that references variables of an enclosing function and keeps them alive after the outer function returns. The captured variables live in cells (`fn.__closure__`). Decorators, callback factories, and function-based counters are closures.

TH-CLO-02 · Easy & Tricky · *Do closures capture values when they're created?*

> **Trap:** "Yes, they snapshot the value."
>
> They capture *variables*, and read the current value when called (late binding). Lambdas created in a loop all see the loop variable's final value (MT-O01). Bind at creation time with a default argument, `functools.partial`, or a factory function.

TH-CLO-03 · Medium · *Closure vs a class with `__call__` vs `functools.partial`: when would you use each?*

> A closure for small private state and decorators. A class when the state should be inspectable, resettable, or needs several methods (and it pickles by default). `partial` to pre-bind arguments of an existing function; it exposes `.func`, `.args`, `.keywords`, and pickles if the function does, which matters for multiprocessing.

TH-CLO-04 · Medium & Tricky · *`def make_counter(): count = 0; def inc(): count += 1; return count; return inc`. What happens when you call the counter?*

> **Trap:** "It returns 1, 2, 3..."
>
> `UnboundLocalError`. `count += 1` is an assignment, so the compiler makes `count` local to `inc`, and the read happens before any local assignment. Fix with `nonlocal count`, or keep state in a mutable object. Two mechanisms: closures read outer variables fine, but assignment changes the scope decision.

TH-CLO-05 · Hard · *Why do lambdas and nested functions fail with multiprocessing?*

> `pickle` serialises functions *by reference*: module plus qualified name, looked up again on the other side. Lambdas and nested functions can't be found that way, so you get `PicklingError/AttributeError`. With the `spawn/forkserver` start methods (default on macOS and Windows, and on Linux since 3.14), everything sent to workers must pickle. Use top-level functions (optionally wrapped with `partial`), or `cloudpickle`, which Spark, Dask and Ray use to serialise closures by value.

TH-CLO-06 · Hard & Tricky · *Can a closure cause a memory leak?*

> **Trap:** "No, closures are just functions."
>
> Yes. A closure keeps every captured variable alive for as long as the function object lives. A callback registered on a long-lived object, or a lambda stored in a cache, that captured a DataFrame or a model keeps that object in memory indefinitely. Inspect with `fn.__closure__[i].cell_contents`; capture only the small values you need, or use `weakref` for large or owning references.

TH-FN · Functions and arguments

TH-FN-01 · Easy · *What do `*args` and `**kwargs` collect?*

> `*args` collects extra positional arguments into a tuple; `**kwargs` collects extra keyword arguments into a dict. At a call site, `*seq` and `**mapping` do the reverse: they spread a sequence into positionals and a mapping into keywords.

TH-FN-02 · Easy & Tricky · *`def log_event(msg, ts=datetime.now()):`. What timestamp does each event get?*

> **Trap:** "The time of each call."
>
> The time the `def` statement ran, usually import time, for every call. Defaults are evaluated once and stored on the function object. Same root cause as mutable defaults (MT-O02). Use `ts=None` and compute inside the function.

TH-FN-03 · Medium · *What do `/` and `*` mean in a function signature, and why use them?*

> Parameters before `/` are positional-only (3.8+); parameters after `*` (or after `*args`) are keyword-only. Positional-only lets you rename parameters later without breaking callers and lets `**kwargs` accept keys with the same names. Keyword-only prevents "magic positional" calls like `send(msg, True, 3)`, so flags and tuning parameters must be named.

TH-FN-04 · Medium & Tricky · *Given `def f(a, b=2, *args, c, **kw)`, what does `f(1, 3, 4, c=5, d=6)` bind, and what does `f(1, 2, 3)` do?*

> **Trap:** "4 goes to `c`" or "`c` is optional".
>
> First call: `a=1, b=3, args=(4,), c=5, kw={'d': 6}`. Second call: `TypeError: f() missing 1 required keyword-only argument: 'c'`. `c` sits after `*args`, so it can only be passed by keyword and has no default.

TH-FN-05 · Hard · *How do you write a decorator that keeps the wrapped function's signature for type checkers and at runtime?*

> At runtime, `functools.wraps` copies metadata and sets `__wrapped__`, which `inspect.signature` follows. For type checkers, use `ParamSpec` + `TypeVar` (PEP 612): `def deco(f: Callable[P, R]) -> Callable[P, R]`, with `*args: P.args, **kwargs: P.kwargs` in the wrapper. For decorators that add or remove a leading parameter, use `Concatenate`.

TH-FN-06 · Hard & Tricky · *Why is a Python loop calling a small function a million times slow, and does CPython inline it?*

> **Trap:** "The interpreter inlines small functions like a compiler would."
>
> CPython never inlines at the source level. Each call creates a frame, binds arguments, and does dynamic name lookups: roughly tens of nanoseconds even after 3.11's cheaper frames and specialising adaptive interpreter (PEP 659). A million calls cost tens of milliseconds before doing any real work. Move the loop into C: builtins (`sum`, `map` with a builtin), comprehensions, `str.join`, or NumPy/pandas vectorised operations.

TH-CMP · Comprehensions

TH-CMP-01 · Easy · *List comprehension, generator expression, or `map`/`filter`: when do you use which?*

> A list comprehension builds a list eagerly and is the most readable for transform-and-filter. A generator expression produces items lazily, ideal for feeding `sum`, `any`, `max`, or a single loop over large data. `map`/`filter` are fine with an existing named function but less readable with lambdas.

TH-CMP-02 · Easy & Tricky · *After `[x * 2 for x in items]`, is `x` defined?*

> **Trap:** "Yes, it holds the last item" (true in Python 2).
>
> No. Comprehensions have their own scope in Python 3. 3.12's PEP 709 inlines comprehensions for speed but keeps this isolation; only an assignment expression (`:=`) inside a comprehension deliberately binds in the enclosing scope.

TH-CMP-03 · Medium · *When is a generator expression the wrong choice?*

> When you need the data more than once, need `len()` or indexing, or will materialise it anyway. For example, `"".join(gen)` builds a list internally, so a list comprehension is slightly faster there. Also avoid it when the source may change before consumption, because laziness means values are computed later.

TH-CMP-04 · Medium & Tricky · *What is `[x for x in range(3) for y in range(x)]`?*

> **Trap:** Reading the clauses right-to-left, or expecting nested lists.
> `[1, 2, 2]`. Multiple `for` clauses nest left to right, exactly like nested loops written in that order: x=0 yields nothing; x=1 has y=0, giving 1; x=2 has y=0,1, giving 2, 2.

TH-CMP-05 · Hard · *Why does a comprehension inside a class body raise NameError for a class attribute?*

> Class scope isn't visible to nested scopes. Only the comprehension's outermost iterable is evaluated in the class scope, so `class C: base = 10; vals = [base + i for i in range(3)]` raises `NameError: name 'base' is not defined`. Fix: compute outside the class, pass through the outermost iterable, or use a regular loop.

TH-CMP-06 · Hard & Tricky · *Where does `y` get bound in `[y := parse(x) for x in rows]`, and can you write `[x := x + 1 for x in rows]`?*

> **Trap:** "`y` stays inside the comprehension like `x`."
> An assignment expression inside a comprehension binds in the *containing* scope (PEP 572), so `y` holds the last parsed value afterwards. This is intentional, e.g. `any((hit := r) for r in rows if r.ok)` to capture the witness. Rebinding the comprehension's own iteration variable with `:=` is a SyntaxError.

TH-GEN · Iterators and generators

TH-GEN-01 · Easy · *What's the difference between an iterable and an iterator?*

> An iterable has `__iter__` returning an iterator (lists, dicts, files, ranges). An iterator has `__next__`, returns itself from `__iter__`, and is single-pass. A `for` loop calls `iter()` once, then `next()` until `StopIteration`.

TH-GEN-02 · Easy & Tricky · *You loop over a generator twice. What does the second loop do?*

> **Trap:** "It iterates the items again."
> Nothing, and silently. The generator is exhausted after the first pass (ET-O03). Recreate it, materialise with `list()`, or use `itertools.tee` if you truly need two streams.

TH-GEN-03 · Medium · *What state does a generator keep between `yield`s, and why does that save memory?*

> Its suspended frame: local variables, the current instruction, and the evaluation stack. Values are produced on demand, so a pipeline over a 50 GB file holds one item per stage instead of a list of everything: O(1) vs O(n) memory.

TH-GEN-04 · Medium & Tricky · *A generator opens a file in a `with` block. The consumer `break`s out of the loop early. When is the file closed?*

> **Trap:** "Immediately at `break`."
> When the generator is *closed*: explicitly via `.close()`, or when the generator object is garbage-collected, which throws `GeneratorExit` at the paused `yield` so `with`/`finally` run. In CPython that's usually as soon as the last reference disappears; on PyPy, or if a reference lingers, the file stays open. For deterministic cleanup, wrap with `contextlib.closing(gen)`.

TH-GEN-05 · Hard · *What does `yield from` do beyond a `for` loop that yields each item?*

> It forwards `send()`, `throw()`, and `close()` to the sub-generator, and the `yield from` expression evaluates to the sub-generator's `return` value (H-O03). That makes generators composable: a parent can collect summaries from children without mixing them into the data stream.

TH-GEN-06 · Hard & Tricky · *How do async generators clean up, and what goes wrong if you abandon one?*

> **Trap:** "Same as sync generators: they're closed on garbage collection."
> Their cleanup (`aclose()`) is itself asynchronous and must be awaited, which can't happen inside a synchronous GC callback. asyncio installs hooks (`sys.set_asyncgen_hooks`) that *schedule* `aclose()` on the loop, and `asyncio.run` calls `loop.shutdown_asyncgens()` at exit. In the meantime connections or cursors held by the generator stay open, and if the loop is already gone, cleanup never runs. Use `contextlib.aclosing(agen)` (3.10+) for deterministic cleanup.

TH-DEC · Decorators

TH-DEC-01 · Easy · *What does `@audit` above `def refund(...)` do?*

> It's syntax for `refund = audit(refund)`: the decorator receives the function object and returns what the name will be bound to, usually a wrapper that adds behaviour before or after calling the original.

TH-DEC-02 · Easy & Tricky · *When does the code inside a decorator run: at call time or at definition time?*

> **Trap:** "When the decorated function is called."
> Both parts exist: the decorator body runs once at definition time (usually module import), and only the wrapper runs per call. Registration decorators such as Flask/FastAPI routes, pytest fixtures, and Celery tasks rely on the import-time part, which is also why importing such modules has side effects.

TH-DEC-03 · Medium · *How do you write a decorator that takes arguments, and why the extra layer?*

> `@retry(attempts=3)` first *calls* `retry(attempts=3)`, which must return the real decorator, which returns the wrapper: three nested functions (factory → decorator → wrapper), or a class. To support both `@retry` and `@retry(...)`, check whether the first positional argument is callable.

TH-DEC-04 · Medium & Tricky · *You write a decorator as a class with `__call__` and apply it to a method. What happens to `self`?*

> **Trap:** "Same as a function decorator."
> The method breaks: `self` is not passed. Plain functions are descriptors: accessed through an instance, `__get__` binds `self`. A class instance with only `__call__` isn't a descriptor, so `obj.method(x)` calls `wrapper(x)` without the instance. Fix: implement `__get__` (returning a bound `functools.partial(self.__call__, obj)`) or write the decorator as a function.

TH-DEC-05 · Hard · *How do you write one decorator that works for both `def` and `async def` functions?*

> Check `inspect.iscoroutinefunction(func)` at decoration time and return an `async def` wrapper that `await`s the call, otherwise a sync wrapper. Doing it at call time (checking whether the result is awaitable) is fragile. For async functions, anything slow inside the wrapper must also be awaited (e.g. `asyncio.sleep`, never `time.sleep`).

TH-DEC-06 · Hard & Tricky · *Why does a decorated function fail to pickle when the decorator doesn't use `functools.wraps`?*

> **Trap:** "Decorators don't affect pickling."
> `pickle` stores a function as "module + qualified name" and checks that looking it up returns the same object. Without `wraps`, the wrapper's `__qualname__` is `decorator.<locals>.wrapper`, which can't be looked up, so `PicklingError`. With `wraps`, `__qualname__` is copied from the original, and the module attribute of that name *is* the wrapper, so the check passes. This matters for multiprocessing, Celery, and Spark UDFs.

TH-CTX · Context managers

TH-CTX-01 · Easy · *What does a `with` statement guarantee?*

> It calls `__enter__` on entry and `__exit__` on exit, whether the block finishes normally, returns, breaks, or raises. That's why files, locks, DB transactions, and temp directories are managed with `with`.

TH-CTX-02 · Easy & Tricky · *After `with open(path) as f:` ends, is `f` still defined?*

> **Trap:** "No, it goes out of scope."
> `f` is still bound (`with` doesn't create a scope), but the file is closed. `f.read()` raises `ValueError: I/O operation on closed file.`

TH-CTX-03 · Medium · *What are `__exit__`'s arguments, and what does its return value mean?*

> `__exit__(exc_type, exc_value, traceback)`: all `None` on a clean exit. Returning a truthy value *suppresses* the exception; returning `None`/`False` lets it propagate. Suppression should be explicit and narrow (`contextlib.suppress(FileNotFoundError)`), never a default.

TH-CTX-04 · Medium & Tricky · *In `with acquire_a() as a, acquire_b() as b:`, `acquire_b()`'s `__enter__` raises. Is `a` released?*

> **Trap:** "Nothing is released" or "both `__exit__`s run".
> The statement is equivalent to nested `with` blocks: `a`'s `__enter__` succeeded, so its `__exit__` runs with the exception; `b`'s `__exit__` is not called because it never entered. The general rule: `__exit__` runs only for managers whose `__enter__` completed.

TH-CTX-05 · Hard · *When do you need `contextlib.ExitStack`?*

> When the number of resources isn't known until runtime (open N run files in H-C03), when managers are conditional, for registering arbitrary cleanup callbacks, and for constructors that acquire several resources and must release the already-acquired ones if a later one fails (`stack.pop_all()` transfers ownership on success).

TH-CTX-06 · Hard & Tricky · *What can go wrong in an async context manager's `__aexit__` under cancellation?*

> **Trap:** "Cleanup always completes, like `finally`."
> `__aexit__` runs, but if it awaits (closing a connection, rolling back) and the task is cancelled again or a timeout fires, the cleanup itself can be interrupted halfway. Swallowing `CancelledError` inside `__aexit__` breaks the cancellation contract for the caller. Keep async cleanup short and idempotent, re-raise cancellation, and use `asyncio.shield` only for the critical step, knowing the shielded work may outlive its caller.

TH-EXC · Exceptions

TH-EXC-01 · Easy · *What are `else` and `finally` for in a `try` statement?*

> `else` runs only if the `try` body raised nothing, which keeps the `try` narrow so you don't catch errors from the success path. `finally` always runs last: after normal completion, a return, or an exception.

TH-EXC-02 · Easy & Tricky · *Does `except Exception:` catch everything?*

> **Trap:** "Yes."
>
> It skips `BaseException` subclasses that aren't `Exception`: `KeyboardInterrupt`, `SystemExit`, `GeneratorExit`, and `asyncio.CancelledError` (a `BaseException` since 3.8). That's deliberate. A bare `except:` or `except BaseException:` catches them too, which breaks Ctrl-C, `sys.exit()`, and task cancellation.

TH-EXC-03 · Medium · *How would you design exceptions for a library or service layer?*

> One base class per library (`class PaymentError(Exception)`), specific subclasses for failures callers handle differently (`CardDeclined`, `GatewayTimeout`), structured attributes instead of parsing messages, and `raise ... from err` when translating lower-level errors. Never subclass `BaseException`. Document what's retryable.

TH-EXC-04 · Medium & Tricky · *Inside an `except` block, what's the difference between `raise`, `raise err`, and `raise NewError() from err`?*

> **Trap:** "`raise err` loses the original traceback."
>
> Bare `raise` re-raises the active exception with its traceback untouched. `raise err` re-raises the same object; its existing traceback is kept and the re-raise location is added, which is noisier but not lossy. `raise NewError() from err` creates a new exception with `__cause__ = err`, printed as "direct cause". Prefer bare `raise` to propagate and `from err` to translate.

TH-EXC-05 · Hard · *What are exception groups and `except*`, and why were they added?*

> Added in 3.11 (PEP 654) to report several unrelated failures at once, mainly from concurrent tasks (`asyncio.TaskGroup`) or batch operations. An `ExceptionGroup` wraps a list of exceptions; each `except*` clause handles the subset matching its type, and unmatched exceptions re-raise as a smaller group. You can't mix `except` and `except*` in one `try`, and `return`/`break`/`continue` aren't allowed inside `except*`.

TH-EXC-06 · Hard & Tricky · *Are exceptions free in modern Python? Is there a hidden memory cost to keeping them?*

> **Trap:** "Exceptions are always expensive" or "free since 3.11".
>
> 3.11 made *entering* a `try` block zero-cost, but *raising* still allocates the exception and builds a traceback, so EAFP is cheap when failures are rare and costly in hot loops where they're common. The hidden cost: a stored exception keeps its `__traceback__`, which references every frame and therefore every local variable along the stack. Appending exceptions to a list in a long-running worker can pin large objects in memory; store formatted strings or call `traceback.clear_frames(tb)`.

TH-OOP · Object-oriented Python

TH-OOP-01 · Easy · *What is `self`, and what's the difference between a class attribute and an instance attribute?*

> `self` is the instance, passed automatically as the first argument when a method is called through an instance (`obj.m()` is `Cls.m(obj)`). Instance attributes live in the object's own `__dict__`; class attributes live on the class and are shared by every instance until an instance assigns its own (E-O04).

TH-OOP-02 · Easy & Tricky · *Does `__init__` create the object?*

> **Trap:** "Yes, it's the constructor."
>
> `__new__` creates and returns the instance; `__init__` then initialises it and must return `None`. You rarely override `__new__`, except when subclassing immutable types (`class Money(Decimal)`, where the value must be set at creation), for caching or singletons, or in metaclass work.

TH-OOP-03 · Medium · *Why use `@property` instead of `get_x()`/`set_x()` methods?*

> A property keeps plain attribute syntax while running code on access: validation in the setter, computed or derived values, read-only attributes. Idiomatic Python starts with a public attribute and converts it to a property later without changing any caller, so explicit getters and setters are unnecessary up front.

TH-OOP-04 · Medium & Tricky · *For an alternative constructor like `Order.from_json(...)`, `@classmethod` or `@staticmethod`?*

> **Trap:** "They're interchangeable."
>
> `@classmethod`. It receives `cls`, so `cls(...)` builds the *subclass* when called as `PriorityOrder.from_json(...)`. A `@staticmethod` has to hardcode `Order(...)` and silently returns base-class instances for subclasses. Use `@staticmethod` only for helpers that need neither the class nor the instance.

TH-OOP-05 · Hard · *Abstract base classes vs `typing.Protocol`: when do you use each?*

> An ABC (`abc.ABC` + `@abstractmethod`) is nominal: implementers must inherit or register, and instantiating a class with unimplemented abstract methods raises TypeError at runtime. It suits frameworks that also share behaviour through the base class. A `Protocol` is structural: any class with matching methods satisfies it for the type checker, with no inheritance needed, ideal for interfaces across libraries or for test fakes. `@runtime_checkable` enables `isinstance`, but it only checks that the attributes exist, not their signatures.

TH-OOP-06 · Hard & Tricky · *What are `__init_subclass__` and `__set_name__`, and why prefer them to a metaclass?*

> **Trap:** "Class-creation customisation needs a metaclass."
>
> `__init_subclass__(cls, **kwargs)` runs on the parent whenever a subclass is defined: ideal for plugin registries ("every `Exporter` subclass registers its format") and for validating subclasses. `__set_name__(self, owner, name)` tells a descriptor which attribute name it was assigned to, which is how field-style descriptors learn their names. Both cover most historical metaclass use cases without metaclass conflicts when combining libraries.

TH-TYP · Typing

TH-TYP-01 · Easy · *Are type hints enforced when the code runs?*

> No. They're stored in `__annotations__` and ignored by the interpreter (ET-O06). Enforcement comes from static checkers (mypy, pyright) in CI and from validation libraries (Pydantic) at runtime boundaries.

TH-TYP-02 · Easy & Tricky · *Does `orders: list[int] = []` make a list that only accepts ints?*

> **Trap:** "Yes, appending a string will fail."
>
> No. It's an ordinary list; the annotation only informs type checkers. `list[int]` itself is a `types.GenericAlias`, and `isinstance(x, list[int])` raises `TypeError` because parameterised generics can't be checked at runtime.

TH-TYP-03 · Medium · *What does `Optional[str]` mean? Is it the same as an optional parameter?*

> `Optional[str]` equals `str | None` (3.10+ syntax): the *value* may be `None`. Whether a parameter is optional depends only on having a default. `def f(x: str | None)` is still required; `def f(x: str = "a")` is optional but never `None`.

TH-TYP-04 · Medium & Tricky · *Why does a type checker reject passing `list[Dog]` where `list[Animal]` is expected, but accept `Sequence[Dog]` for `Sequence[Animal]`?*

> **Trap:** "Subtyping carries over to containers automatically."
>
> Variance. `list` is mutable, so it's *invariant*: a function that accepts `list[Animal]` could append a `Cat` into your `list[Dog]`. `Sequence` is read-only, so it's *covariant*. Rule of thumb: accept the most abstract read-only type you need (`Sequence`, `Mapping`, `Iterable`) and return concrete types.

TH-TYP-05 · Hard · *What did PEP 695 (Python 3.12) add for generics?*

> Inline type-parameter syntax: `def first[T](items: Sequence[T]) -> T`, `class Box[T]: ...`, bounds and constraints (`[T: Hashable]`, `[T: (int, str)]`), and the `type` statement for aliases (`type UserId = int`) whose value is evaluated lazily. It replaces most `TypeVar(...)` boilerplate and gives type parameters proper scoping. Older syntax still works and is needed for code that supports 3.11 or earlier.

TH-TYP-06 · Hard & Tricky · *What is `if TYPE_CHECKING:` for, and what does Python 3.14 change about annotations?*

> **Trap:** "Imports under `TYPE_CHECKING` are available at runtime."
>
> `typing.TYPE_CHECKING` is `False` at runtime and `True` for type checkers, so imports under it exist only for annotations. That breaks import cycles and avoids importing heavy modules, but the names don't exist at runtime, so annotations referencing them must not be evaluated eagerly (quote them, or use `from __future__ import annotations`). 3.14 (PEP 649/749) evaluates annotations lazily by default, so forward references work without quotes, and runtime consumers should read them via `annotationlib.get_annotations`.

TH-STR · Strings and bytes

TH-STR-01 · Easy · *What's the difference between `str` and `bytes`, and where do conversions happen?*

> `str` is a sequence of Unicode code points; `bytes` is raw 8-bit data. You `encode` text to bytes and `decode` bytes to text, at the I/O boundaries (files, sockets, HTTP bodies, Kafka messages), always with an explicit encoding, and work with `str` everywhere in between.

TH-STR-02 · Easy & Tricky · *Is `len("é")` always 1?*

> **Trap:** "Yes, it's one character."
>
> It's 1 for the precomposed `"\u00e9"` but 2 for `"e\u0301"` (e plus a combining accent), which renders identically. The byte length differs again: 2 bytes in UTF-8 for the precomposed form. Emoji are one code point in Python but two UTF-16 units in JavaScript's `.length`. Normalise (NFC/NFKC) before comparing or enforcing length limits (ET-C02).

TH-STR-03 · Medium · *What's the correct way to open a text file for reading?*

> `open(path, encoding="utf-8")`, always explicit, because the default is the locale encoding (still cp1252 on many Windows machines until UTF-8 mode becomes the default, planned by PEP 686). Use `utf-8-sig` for files from Excel (BOM), and pass `newline=""` when using the `csv` module. Decide on `errors=` deliberately; `"strict"` is usually right for pipelines.

TH-STR-04 · Medium & Tricky · *Is building a string with `s += piece` in a loop quadratic?*

> **Trap:** Either "always quadratic" or "always fine".
>
> Strings are immutable, so in principle each `+=` copies everything so far: O(n²). CPython has an optimisation that resizes in place when the string's reference count is 1, so it's often fast in practice. But it silently disappears if another reference exists (e.g. the string is also stored in a list or attribute) and on other implementations. Use `"".join(parts)` or `io.StringIO` for guaranteed linear time.

TH-STR-05 · Hard · *Why does `logging` use `logger.info("user=%s", user_id)` instead of an f-string?*

> The `%`-style arguments are formatted lazily, only if the record is actually emitted, which saves work when the level is disabled. The message template stays constant, so log aggregators (Sentry, ELK) can group occurrences. Separately, never use user input *as* a format string: `user_template.format(obj=request)` lets `{obj.__class__.__init__.__globals__}`-style fields reach into your objects.

TH-STR-06 · Hard & Tricky · *A validator checks `s.isdigit()` and then calls `int(s)`. How can this still crash or accept surprising input?*

> **Trap:** "`isdigit()` guarantees `int()` works."
>
> `"²".isdigit()` is True but `int("²")` raises ValueError: `isdigit` includes superscripts, while `int` accepts only decimal digits. Conversely, `int("١٢٣")` (Arabic-Indic digits) returns 123, and `int(" 42 ")`, `int("4_2")`, and `int("+42")` all succeed. For ASCII-only IDs, validate with `re.fullmatch(r"[0-9]+", s)` or `s.isascii() and s.isdecimal()`.

TH-NUM · Numbers

TH-NUM-01 · Easy · *What do `/`, `//` and `%` return, and how big can an int be?*

> `/` is true division and always returns a float; `//` is floor division; `%` is the remainder consistent with `//`. Ints are arbitrary precision (limited only by memory), though since 3.11 converting ints with more than 4300 digits to or from `str` raises by default (`sys.set_int_max_str_digits`) to prevent a DoS.

TH-NUM-02 · Easy & Tricky · *What is `-7 // 2`, and what is `-7 % 2`?*

> **Trap:** `-3` and `-1` (C/Java truncation semantics).
>
> `-4` and `1`. Python floors toward negative infinity and the remainder takes the divisor's sign, so `(a // b) * b + a % b == a` always holds. `int(-7 / 2)` is `-3` (truncation) and `math.fmod(-7, 2)` is `-1.0`. It matters when bucketing negative timezone offsets, timestamps before an epoch, or refunds (ET-C01).

TH-NUM-03 · Medium · *Why is `0.1 + 0.2 != 0.3`, and how should you compare floats?*

> Floats are IEEE 754 binary doubles, and 0.1 has no exact binary representation, so small errors appear. Compare with `math.isclose(a, b, rel_tol=..., abs_tol=...)`; the `abs_tol` is needed near zero. Use `Decimal` for exact decimal arithmetic (money) and `fractions.Fraction` for exact rationals.

TH-NUM-04 · Medium & Tricky · *Does using `Decimal` eliminate precision problems?*

> **Trap:** "Yes, Decimal is exact."
>
> Only for decimal inputs within its context. `Decimal(1) / 3` still rounds, to 28 significant digits by default; `Decimal(0.1)` from a float carries the binary error (ET-O01); mixing `Decimal` and `float` in arithmetic raises TypeError. For money you still choose a quantum and rounding mode (`quantize(Decimal("0.01"), ROUND_HALF_UP)`).

TH-NUM-05 · Hard · *How do NaN and infinity behave in comparisons, `max()`, and sorting?*

> Every comparison with NaN is False, and NaN != NaN. So `max([nan, 1])` returns `nan` while `max([1, nan])` returns `1`, which depends on order, and sorting a list with NaNs produces an arbitrary, unsorted-looking order without any error. Detect with `math.isnan`; in pandas, aggregations skip NaN by default (`skipna=True`); `json.dumps` emits the invalid token `NaN` unless `allow_nan=False`.

TH-NUM-06 · Hard & Tricky · *Can `sum()` over the same floats give different results on Python 3.11 and 3.12?*

> **Trap:** "No, floating-point addition is deterministic."
>
> Yes. Since 3.12, `sum()` of floats uses compensated (Neumaier) summation, so `sum([0.1] * 10)` is `1.0` on 3.12 but `0.9999999999999999` on 3.11. Golden-file tests or checksums over float sums can break on upgrade. `math.fsum` gives correctly rounded sums on every version; for reproducible reporting, round at the presentation boundary or use `Decimal`.

TH-COL · collections, itertools, functools

TH-COL-01 · Easy · *When would you reach for `Counter`, `defaultdict`, `deque`, and `namedtuple`?*

> `Counter` for frequency counts and multiset maths (M-C02). `defaultdict(list)` for grouping without key checks. `deque` for queues and sliding windows (O(1) at both ends, `maxlen` for bounded buffers). `namedtuple` or `typing.NamedTuple` for small immutable records with named fields (often a frozen dataclass today).

TH-COL-02 · Easy & Tricky · *Is reading `groups["missing"]` from a `defaultdict(list)` side-effect free?*

> **Trap:** "Yes, it just returns an empty list."
>
> It *inserts* the key with an empty list. Later `"missing" in groups` is True, `len(groups)` grows, and serialised output gains empty entries. Use `groups.get(key, [])` or `key in groups` for reads.

TH-COL-03 · Medium · *Why use `collections.deque` instead of a list for a queue?*

> `list.pop(0)` and `list.insert(0, x)` shift every element: O(n). `deque.popleft()` and `appendleft()` are O(1). `deque(maxlen=N)` discards from the other end automatically, which is handy for "last N events". Indexing into the middle of a deque is O(n), so it's not a list replacement for random access.

TH-COL-04 · Medium & Tricky · *Is `@lru_cache` safe on a function that returns a list or dict?*

> **Trap:** "Yes, it just remembers the result."
>
> It returns the *same object* to every caller, so one caller's mutation corrupts all later results (MT-C01). Also: arguments must be hashable; `maxsize=None` (or `@functools.cache`) grows without bound; on methods it holds `self` and prevents instances from being freed (HT-C03). Return immutable values or copies.

TH-COL-05 · Hard · *How does `functools.lru_cache` work, and is it thread-safe?*

> The C implementation keeps a dict from argument keys to entries in a circular doubly linked list ordered by recency; hits move to the front, and misses beyond `maxsize` evict the tail. Keys are built from positional args plus keyword items (so `f(1)` and `f(x=1)` are different entries, and `typed=True` separates `1` from `1.0`). It's thread-safe in that its structure won't be corrupted, but concurrent misses on the same key can call the function more than once: no single-flight.

TH-COL-06 · Hard & Tricky · *When is `itertools.tee` worse than calling `list()`?*

> **Trap:** "`tee` is always the memory-efficient option."
>
> `tee` buffers every item that one copy has consumed and another hasn't. If one iterator runs to the end before the other starts (e.g. `sum(a)` then `max(b)`), the buffer holds the entire stream, which is equivalent to `list()` but with more overhead. `tee` copies aren't thread-safe, and the original iterator must not be used afterwards. It only helps when the copies advance roughly in step.

TH-SRT · Sorting

TH-SRT-01 · Easy · *`sorted(x)` vs `x.sort()`?*

> `sorted()` accepts any iterable and returns a new list; `list.sort()` sorts in place and returns `None`. Both take `key=` and `reverse=`, and both are stable.

TH-SRT-02 · Easy & Tricky · *What does `sorted(["item10", "item2", "item1", "Item3"])` return?*

> **Trap:** `['item1', 'item2', 'Item3', 'item10']`.
>
> `['Item3', 'item1', 'item10', 'item2']`. Strings compare lexicographically by code point, so uppercase sorts before lowercase and `"10"` before `"2"`. Use `key=str.casefold` for case-insensitive order, and a natural-sort key (split digits with a regex and convert them to ints) for human numbering.

TH-SRT-03 · Medium · *What does "stable sort" mean, and how do you use it for multi-key ordering?*

> Items with equal keys keep their original relative order. You can sort by the least significant key first and then by more significant keys, or use one tuple key, negating numeric fields that should be descending (E-O05, ET-C03).

TH-SRT-04 · Medium & Tricky · *How do you sort records where some `due_date` values are `None`?*

> **Trap:** "`None` sorts first automatically" (as in Python 2 or some SQL engines).
>
> In Python 3, `sorted` raises `TypeError: '<' not supported between instances of 'NoneType' and ...`. Use a key that ranks the Nones explicitly: `key=lambda r: (r.due_date is None, r.due_date or date.min)` puts them last. Decide the policy deliberately, since SQL engines differ (`NULLS FIRST/LAST`).

TH-SRT-05 · Hard · *Why is Python's sort so fast on real-world data?*

> It's an adaptive merge sort (Timsort; 3.11 switched its merge policy to "powersort"): O(n log n) worst case and O(n) on data that's already sorted or reverse-sorted, because it detects existing runs and merges them, galloping through long one-sided stretches. The `key` function is called exactly once per element (decorate-sort-undecorate), so an expensive key costs n calls, not n log n. `functools.cmp_to_key` adapts old comparator functions at some cost.

TH-SRT-06 · Hard & Tricky · *What happens if your comparison isn't a consistent ordering, or the key function raises halfway through?*

> **Trap:** "Python detects it and raises an error."
>
> An inconsistent ordering (NaN values, a `__lt__` that isn't transitive) produces an arbitrary result with no error. If a key function or comparison raises, the exception propagates, and for `list.sort()` the list may be left partially modified. `functools.total_ordering` helps define a consistent set of comparison methods from `__eq__` and one of `__lt__`/`__gt__`.

TH-HSH · Hashing, dicts and sets

TH-HSH-01 · Easy · *Why are dict and set lookups O(1) on average?*

> The key's hash selects a slot in a hash table; one or a few probes plus an equality check confirm the match. It's O(1) on average and O(n) in the worst case when many keys collide.

TH-HSH-02 · Easy & Tricky · *How many keys does `{1: "a", 1.0: "b", True: "c"}` have?*

> **Trap:** "Three: they're different types."
>
> One: `{1: 'c'}[1.0]` is `'c'`. `1 == 1.0 == True` and they hash equal, so they're the same key. The dict keeps the first key object (`1`) and the last value (`"c"`). Real case: mixing boolean flags and integer IDs as keys, or JSON `true` vs `1`.

TH-HSH-03 · Medium · *What makes an object hashable, and what's the contract with `__eq__`?*

> It has a `__hash__` that returns an int which never changes while the object is stored, and objects that compare equal must have equal hashes. Defining `__eq__` without `__hash__` makes a class unhashable (ET-O04); hash only immutable fields; `@dataclass(frozen=True)` generates a consistent pair.

TH-HSH-04 · Medium & Tricky · *Why does a test that prints a `set` of strings pass locally and fail in CI?*

> **Trap:** "Set order is random" or "set order is insertion order."
>
> Neither. Set iteration order follows hash values and table layout, and `str/bytes` hashes are randomised per process (`PYTHONHASHSEED`) to prevent hash-flooding attacks. So the same code prints a different order in different runs. Ints hash to themselves, which is why small int sets often *look* sorted and hide the problem. Sort before printing or comparing; never rely on set order.

TH-HSH-05 · Hard · *How does a CPython dict preserve insertion order while staying compact?*

> Since 3.6 it stores entries in a dense array in insertion order (hash, key, value) plus a separate small sparse index table that maps hash slots to positions in the dense array. Iteration walks the dense array, which gives insertion order. Deleted entries leave dummies until a resize compacts them. Instance attribute dicts share key tables between instances of the same class, and 3.11+ stores attributes more compactly still, which is partly why `__slots__` saves less than it used to.

TH-HSH-06 · Hard & Tricky · *What is hash flooding, and why are `str` hashes randomised but `int` hashes not?*

> **Trap:** "Hash randomisation protects all dict keys."
>
> An attacker who can choose many colliding keys (form fields, JSON object keys) turns dict inserts into $O(n^2)$ work: a CPU denial of service. Python hashes `str/bytes` with SipHash and a random per-process key, so collisions can't be precomputed. `int` hashes are deterministic (`hash(n)` is `n` modulo a large prime), so user-controlled integer keys remain attackable. Mitigate by limiting payload sizes and key counts at the boundary.

TH-CON · Concurrency (GIL, threads, processes)

TH-CON-01 · Easy · *Threads, processes, or asyncio: which fits which workload?*

> Threads for blocking I/O with libraries that aren't async (DB drivers, `requests`, boto3). Processes for CPU-bound pure-Python work (separate interpreters, true parallelism). asyncio for large numbers of concurrent I/O operations with async-native libraries on one thread.

TH-CON-02 · Easy & Tricky · *"The GIL makes my Python code thread-safe." True?*

> **Trap:** "Yes, only one thread runs at a time."
>
> The GIL protects the interpreter's internals, not your invariants. Threads can switch between bytecodes, so `counter += 1` (load, add, store) and check-then-act sequences (`if key not in d: d[key] = compute()`) can interleave and lose updates. Use `threading.Lock`, thread-safe structures (`queue.Queue`), or avoid shared mutable state. On free-threaded builds there's no GIL at all.

TH-CON-03 · Medium · *When does multithreading actually speed up Python code?*

> When threads spend time outside the interpreter: waiting on sockets, disk, or `time.sleep`, which release the GIL, or running C code that releases it (NumPy operations, `hashlib`, `zlib`, many DB drivers). Pure-Python CPU work gets no speed-up on the standard build and can even slow down due to contention.

TH-CON-04 · Medium & Tricky · *A `ProcessPoolExecutor` script works on Linux but on macOS it runs the top-level code again in each worker, or fails with a pickling error. Why?*

> **Trap:** "macOS has a multiprocessing bug."
>
> Start methods differ. `fork` copies the parent; `spawn` (default on macOS and Windows) and `forkserver` (default on Linux since 3.14) start a fresh interpreter that *imports the main module*, so unguarded top-level code runs again in every worker. Hence the `if __name__ == "__main__":` guard. Everything sent to workers must pickle, so no lambdas or nested functions. Separately, `fork` in a process that already has threads can deadlock on locks copied in a held state.

TH-CON-05 · Hard · *How do you design a producer/consumer pipeline with threads and shut it down cleanly?*

> Use `queue.Queue(maxsize=N)` so producers block when consumers fall behind (backpressure). Consumers loop on `get()`, handle errors per item, and call `task_done()`. Shutdown: put one sentinel per consumer (or use `Queue.shutdown()` in 3.13+), then `join()` the threads. Avoid shared mutable state beyond the queue, acquire multiple locks in a fixed order to prevent deadlock, and propagate worker exceptions back to the main thread (e.g. via futures).

TH-CON-06 · Hard & Tricky · *What changes for library authors on free-threaded CPython (3.13t/3.14t)?*

> **Trap:** "Nothing, since the GIL never guaranteed safety" or "all existing C extensions just work."
>
> Threads truly run in parallel. Built-in containers use per-object locking, so a single `list.append` or `dict[key] = v` stays safe, but compound operations need explicit locks, and code that relied on the GIL for accidental atomicity breaks more often. C extensions must declare free-threading support; importing one that doesn't re-enables the GIL (with a warning). Expect some single-threaded overhead, and note that immortal objects (PEP 683) and biased reference counting exist largely to make this workable.

TH-ASY · asyncio

TH-ASY-01 · Easy · *What is the event loop, and what does `await` do?*

> The event loop runs on one thread and schedules tasks. `await` suspends the current coroutine until the awaited operation completes, and gives the loop the chance to run other tasks meanwhile. Concurrency comes from interleaving at `await` points, not from parallel execution.

TH-ASY-02 · Easy & Tricky · *What does calling `fetch_orders()` do if it's defined with `async def`?*

> **Trap:** "It runs the function."
>
> It only creates a coroutine object; none of the body runs until it's awaited or wrapped in a task (`asyncio.create_task`). Forgetting the `await` leads to "coroutine was never awaited" RuntimeWarnings and work that silently never happens.

TH-ASY-03 · Medium · *`gather`, `TaskGroup`, `as_completed`, `wait`: when do you use which?*

> `gather`: results in argument order; `return_exceptions=True` to collect errors as values. `TaskGroup` (3.11+): structured concurrency, where a failure cancels the siblings and raises an `ExceptionGroup` (HT-O04). `as_completed`: process results as they finish. `wait`: lower-level control with `FIRST_COMPLETED` or a timeout, returning done/pending sets.

TH-ASY-04 · Medium & Tricky · *You move a slow `requests.get` call into an `async def` helper. Does that make it non-blocking?*

> **Trap:** "Yes, it's async now."
>
> No. `async def` doesn't change what the code inside does. A synchronous call still blocks the only thread running the loop, stalling every other task (H-O02). Use an async client (`httpx.AsyncClient`, `aiohttp`), or offload with `await asyncio.to_thread(requests.get, url)`; CPU-bound work needs a process pool.

TH-ASY-05 · Hard · *How does task cancellation work in asyncio?*

> `task.cancel()` requests that `CancelledError` be raised inside the task at its next `await`. The task should clean up (`finally`, `async with`) and let it propagate. Timeouts (`asyncio.timeout()` 3.11+, `wait_for`) are built on cancellation and convert it to `TimeoutError` at the boundary. `asyncio.shield()` protects an inner operation from the outer cancellation. Tasks that swallow `CancelledError` break timeouts and graceful shutdown (HT-C01).

TH-ASY-06 · Hard & Tricky · *Why must you keep a reference to a task returned by `asyncio.create_task()`?*

> **Trap:** "The event loop keeps every task alive until it finishes."
>
> The loop keeps only *weak* references to tasks. A fire-and-forget task with no other reference can be garbage-collected mid-execution, and its exceptions surface only as a "Task exception was never retrieved" log line, if at all. Keep tasks in a set and remove them with `add_done_callback(tasks.discard)`, or run them inside a `TaskGroup`.

TH-MEM · Memory management

TH-MEM-01 · Easy · *How does CPython free memory?*

> Primarily reference counting: an object is freed as soon as nothing refers to it. A cyclic garbage collector periodically finds groups of objects that only reference each other. Small objects come from the `pymalloc` allocator, which reuses memory internally.

TH-MEM-02 · Easy & Tricky · *Does `del df` free the DataFrame's memory?*

> **Trap:** "Yes, `del` deletes the object."
>
> `del` removes a *name*. The object is freed only when its last reference goes away. Another variable, a list, a cache, a closure, or an exception traceback that still references it keeps it alive (HT-O05).

TH-MEM-03 · Medium · *How do you process a 20 GB file or table without running out of memory?*

> Stream it: iterate the file object line by line, use generators for each stage, read in chunks (`pd.read_csv(chunksize=...)`, `pyarrow` batches), push computation into the database or a columnar engine (DuckDB, Polars lazy, Spark), and write results incrementally. Keep only aggregates or bounded buffers in memory.

TH-MEM-04 · Medium & Tricky · *You delete a huge list, but the process's RSS doesn't drop. Is that a leak?*

> **Trap:** "Yes, Python leaked memory."
>
> Not necessarily. `pymalloc` keeps freed small-object memory in its arenas for reuse, and fragmentation can prevent whole arenas from returning to the OS; the C allocator may also hold on to freed blocks. The memory is available to Python again, just not to other processes. Confirm with `tracemalloc` (Python-level allocations) before calling it a leak; for batch jobs with large peaks, do the heavy work in a subprocess so its memory is released when it exits.

TH-MEM-05 · Hard · *How would you track down a memory leak in a long-running service?*

> Take `tracemalloc` snapshots at intervals and compare top allocations by traceback; count objects by type (`gc.get_objects`, `objgraph.show_growth`); use `memray` for native allocations. Usual suspects: unbounded caches (`lru_cache(maxsize=None)`, module-level dicts), closures or callbacks capturing large objects, stored exceptions with tracebacks, `lru_cache` on methods (HT-C03), growing global lists, and C-extension leaks.

TH-MEM-06 · Hard & Tricky · *What are immortal objects (PEP 683, 3.12), and why do they matter?*

> **Trap:** "`sys.getrefcount(None)` tells you how many references to `None` exist."
>
> Some objects (`None`, `True`/`False`, small ints, many interned strings) are marked immortal: their reference counts are never updated and they're never freed. That avoids writes to shared cache lines, keeps memory pages shared after `fork` (copy-on-write friendliness), and supports per-interpreter GILs and free-threading. `sys.getrefcount` on them returns a huge constant, not a real count.

TH-MOD · Modules and imports

TH-MOD-01 · Easy · *What does `if __name__ == "__main__":` do?*

> A module's `__name__` is `"__main__"` when it's run as a script and its module name when imported. The guard runs code only in the script case, so importing the module (from tests, other modules, or multiprocessing workers) has no side effects.

TH-MOD-02 · Easy & Tricky · *Does importing the same module in ten files run its top-level code ten times?*

> **Trap:** "Yes, each import executes the file."
>
> Once. The first import caches the module in `sys.modules`; later imports return the cached object. The exception: the same file imported under two *names* (run as `__main__` and also imported as `orders`, or reachable as both `src.app.db` and `app.db` due to `sys.path` setup) produces two separate module objects, with duplicated state and "singletons".

TH-MOD-03 · Medium · *Why does `python app/worker.py` fail with "attempted relative import with no known parent package", and how do you fix it?*

> Running a file by path makes it `__main__` with no package context, so relative imports (`from .db import session`) can't resolve. Run it as a module from the project root (`python -m app.worker`), install the package (`pip install -e .`), or use absolute imports with a proper package layout (the `src/` layout avoids accidental imports from the working directory).

TH-MOD-04 · Medium & Tricky · *Module A does `from config import settings`. Later, code runs `config.settings = load_new()`. Which settings does A see?*

> **Trap:** "The new ones: it's the same module."
>
> The old object. `from config import settings` copied the *binding* into A's namespace at import time; rebinding `config.settings` doesn't update A's name. Code that uses `import config` and reads `config.settings` sees the new value. Same mechanism as patching where a name is looked up (M-O06).

TH-MOD-05 · Hard · *How do you fix circular imports structurally rather than with hacks?*

> Move shared definitions (constants, base classes, interfaces) into a lower-level module that both depend on; invert dependencies so high-level modules depend on abstractions; use `import module` and attribute access at call time; import inside functions for rare paths; put type-only imports under `TYPE_CHECKING`. A cycle usually means two modules are one concept or a layer boundary is wrong (H-O06).

TH-MOD-06 · Hard & Tricky · *Walk through what `import pricing` does internally.*

> **Trap:** "It reads `pricing.py` and runs it."
>
> Check `sys.modules` (return the cached module if present). Otherwise ask each finder on `sys.meta_path` (built-in, frozen, then `PathFinder`, which scans `sys.path` entries) for a *module spec*. The spec's loader creates the module object, which is inserted into `sys.modules` **before** execution (the reason partially initialised modules exist in cycles), then `exec_module` runs the code, using cached bytecode from `__pycache__/*.pyc` if its mtime or hash is still valid. Directories without `__init__.py` become namespace packages. On failure, the module is removed from `sys.modules`.

TH-TST · Testing

TH-TST-01 · Easy · *What is a pytest fixture, and why is it preferable to `setUp` methods?*

> A function that provides a test dependency, requested simply by naming it as a test parameter. Fixtures compose (fixtures can use fixtures), have scopes, and handle teardown after a `yield`. They make dependencies explicit per test instead of hiding them in class-level setup.

TH-TST-02 · Easy & Tricky · *`mock.patch("pricing.get_rate")`: does that patch every module that uses `get_rate`?*

> **Trap:** "Yes, it patches the function everywhere."
>
> It replaces the attribute on the `pricing` module only. Modules that did `from pricing import get_rate` keep their own reference to the original. Patch where the name is looked up: `mock.patch("invoices.get_rate")` (M-O06).

TH-TST-03 · Medium · *What are fixture scopes, and what goes wrong with broad scopes?*

> `function` (default), `class`, `module`, `package`, `session`. Broader scopes cost less setup (e.g. one DB container per session), but mutable state leaks between tests, making them order-dependent and flaky. Keep broad fixtures read-only or reset them per test (transactions rolled back per test, factories instead of shared objects), and use built-ins such as `tmp_path` and `monkeypatch`, which undo themselves automatically.

TH-TST-04 · Medium & Tricky · *Your unit tests pass, but production fails with `AttributeError: 'RateClient' object has no attribute 'fetch_rate'`. How did the tests miss it?*

> **Trap:** "The mock library is broken" or "tests just need more coverage."
>
> A plain `Mock`/`MagicMock` invents any attribute you access, so tests calling `client.fetch_rate()` pass even after the real method was renamed to `get_rate`. Use `mock.create_autospec(RateClient)`, `patch(..., autospec=True)`, or `spec_set=` so the mock only allows the real API and signature. Modern Python (3.12) rejects common misspelled assertion names (`asert_called_once`, `called_once_with`), but it can't detect API drift; also watch for `m.assert_called_once` without parentheses, which asserts nothing.

TH-TST-05 · Hard · *How do you test code that depends on time, randomness, and the network?*

> Inject dependencies: pass a clock function (H-C02, BE-C01), a seeded `random.Random`/`np.random.default_rng`, and an HTTP client or repository object that tests replace with fakes. For code you can't change, use `time-machine`/`freezegun` for time and `respx`/`responses` for HTTP. Use real dependencies in containers (testcontainers) for integration tests, and never `sleep()` in tests to wait for things; await events or poll with a timeout.

TH-TST-06 · Hard & Tricky · *What does property-based testing find that example tests miss? Give invariants for `split_amount` (ET-C01) and `external_sort` (H-C03).*

> **Trap:** "It's just random testing."
>
> Tools like Hypothesis generate inputs from strategies, explore edge cases (empty, huge, Unicode, boundary values), and *shrink* failures to a minimal counterexample. Useful invariants: for `split_amount`, the parts sum exactly to the total, their count equals `parts`, and max − min ≤ 0.01. For `external_sort`, the output equals `sorted(input_lines)` for arbitrary text including no trailing newline, duplicates, and non-ASCII, and the result doesn't depend on `max_lines_in_memory`. Invariants find the bugs nobody thought to write an example for.

Role-specific topics

TH-DE · Data Engineer

TH-DE-01 · Easy · *What does it mean for a pipeline to be idempotent, and why does it matter?*

> Running it again with the same input leaves the target in the same state as running it once: no duplicated rows, no double-counted aggregates. It matters because retries, backfills, and partial failures are normal, and an idempotent job can simply be rerun instead of manually repaired.

TH-DE-02 · Easy & Tricky · *You read a CSV exported from Excel with `open(path)` and the `csv` module. What can go wrong?*

> **Trap:** "Nothing, CSV is simple."
>
> The default encoding is platform-dependent, and Excel often writes a UTF-8 BOM, so the first header becomes `"\ufeffcustomer_id"` and lookups by column name fail. Quoted fields can contain embedded newlines, which break unless the file is opened with `newline=""`. Use `open(path, encoding="utf-8-sig", newline="")`, and remember every field arrives as a `str` (DE-O02).

TH-DE-03 · Medium · *How do you implement an idempotent load into a relational table?*

> Upsert on a natural or business key (`INSERT ... ON CONFLICT (key) DO UPDATE` in Postgres and SQLite, `MERGE` in warehouses), or replace a whole partition atomically (write to staging, then swap or overwrite the partition for that date). Deduplicate the input first (latest record per key), make derived columns deterministic (no `now()` in outputs), and record the load's batch or watermark so reruns are traceable.

TH-DE-04 · Medium & Tricky · *In PySpark, `df.filter(...).withColumn(...)` returns in 50 ms. Is the pipeline fast?*

> **Trap:** "Yes, it ran in 50 ms."
>
> Nothing ran. Transformations are lazy: they only build a logical plan. Work happens at an *action* (`count`, `collect`, `write`, `show`). Each action recomputes the full lineage from the source unless you `cache()`/`persist()` (which is itself lazy until the next action). So timing transformations measures nothing, and calling `count()` "for logging" before `write()` can double the job's cost.

TH-DE-05 · Hard · *Explain shuffles, broadcast joins, and data skew in Spark, and how you'd fix a skewed join.*

> Wide transformations (`groupBy`, `join`, `distinct`, `repartition`) redistribute rows by key across the network: a shuffle, usually the dominant cost. A broadcast join ships a small table to every executor (default threshold `spark.sql.autoBroadcastJoinThreshold`, 10 MB) and avoids shuffling the big side. Skew means a few keys hold most rows, so one task runs far longer than the rest. Fixes: enable AQE skew-join handling (`spark.sql.adaptive.skewJoin.enabled`), salt hot keys (append a random suffix and replicate the matching side), handle hot keys separately, or pre-aggregate before joining.

TH-DE-06 · Hard & Tricky · *What does "exactly-once" mean for a Kafka → processing → sink pipeline?*

> **Trap:** "Kafka guarantees exactly-once delivery, so duplicates can't happen."
>
> Delivery over a network is at-least-once; "exactly-once" is an end-to-end *effect* guarantee. It needs either transactions that commit consumer offsets atomically with the outputs (Kafka transactions for Kafka-to-Kafka, or Spark Structured Streaming checkpoints paired with an idempotent or transactional sink), or at-least-once processing plus idempotent writes (upsert by event ID). Late and out-of-order data additionally need event-time windows with watermarks. In practice: at-least-once plus idempotent sinks, and dedupe keys everywhere.

TH-DS · Data Scientist

TH-DS-01 · Easy · *What is vectorisation, and why is it faster than a Python loop?*

> Expressing an operation over whole arrays (`prices * qty`) so the loop runs in compiled C over contiguous memory, often with SIMD instructions, instead of one interpreted Python operation per element. Speed-ups of 10–100× are common; `df.iterrows()` for numeric work is the classic anti-pattern.

TH-DS-02 · Easy & Tricky · *Does `df[df["income"] == np.nan]` select rows with missing income?*

> **Trap:** "Yes."
>
> It returns no rows: NaN compares unequal to everything, including itself. Use `df["income"].isna()` (or `notna()`). Similarly, `np.nan in values` checks identity first and can give inconsistent answers.

TH-DS-03 · Medium · *`groupby().agg()` vs `transform()` vs `apply()`: when do you use each?*

> `agg` reduces each group to one row (sum, mean per segment). `transform` returns a result aligned to the original rows, ideal for features like "customer's average order value" on every order row, or group-wise normalisation. `apply` is the flexible, slow fallback for logic that neither covers. Prefer named built-in aggregations; they run in optimised code.

TH-DS-04 · Medium & Tricky · *A churn model gets 0.98 AUC with a random 80/20 split. What do you check before celebrating?*

> **Trap:** "Nothing, the metric is high; ship it."
>
> Leakage. Time leakage (training on the future: use a time-based split); entity leakage (the same customer in train and test: `GroupKFold`); target leakage (a feature derived from the outcome, like `cancellation_reason`); preprocessing fitted on the full dataset (scalers, imputers, target encoders fit before the split: put them in a `Pipeline`). Also duplicates across the split. Results that look too good usually are.

TH-DS-05 · Hard · *How does a bootstrap confidence interval work, and when is it misleading?*

> Resample the observations with replacement B times (e.g. 10,000), compute the statistic on each resample, and take percentiles (2.5th and 97.5th for 95%), using a seeded `np.random.default_rng(seed)` for reproducibility. It assumes independent, representative samples. It misleads for dependent data (time series need block bootstrap; clustered data needs resampling whole clusters), for extreme statistics like the max, and for very small samples.

TH-DS-06 · Hard & Tricky · *What breaks when a pandas 2.x codebase upgrades to pandas 3.0?*

> **Trap:** "It's a major version, but nothing important changes for normal code."
>
> Copy-on-Write becomes the only mode: chained assignment (`df[mask]["col"] = v`, or `df["col"].fillna(0, inplace=True)`) never modifies `df` (it warns with `ChainedAssignmentError`), and Series extracted from a DataFrame no longer write through (DS-O02). String columns default to the new `str` dtype (PyArrow-backed when available) instead of `object`, so `dtype == object` checks, some `.apply` string code, and missing-value handling (NaN semantics) change. Test with `pd.options.mode.copy_on_write = True` on 2.2 first.

TH-MLE · Machine Learning Engineer

TH-MLE-01 · Easy · *Why batch inference requests, and what's the tradeoff?*

> Batching amortises fixed per-call costs (Python overhead, GPU kernel launches, network round trips) and uses vectorised hardware efficiently, increasing throughput. The tradeoff is latency: a request may wait for a batch to fill. Dynamic batching (wait at most N ms or until the batch is full) balances the two.

TH-MLE-02 · Easy & Tricky · *Is pickle a safe way to share a trained model with another team?*

> **Trap:** "Yes, it's the standard Python serialisation format."
>
> Unpickling executes code chosen by whoever produced the file (`__reduce__`), so loading an untrusted pickle is remote code execution. Pickles also depend on exact class paths and library versions. `joblib` files are pickles too. Prefer `safetensors` for tensors, ONNX for portable models, or `skops` for scikit-learn, and pin library versions alongside any pickled artefact.

TH-MLE-03 · Medium · *Precision, recall, F1: when do you prioritise each?*

> Precision = TP / (TP + FP): prioritise when false positives are costly (blocking legitimate payments). Recall = TP / (TP + FN): prioritise when misses are costly (fraud, disease screening). F1 is their harmonic mean for a single balanced number. All three depend on the decision threshold; under heavy class imbalance, PR-AUC is more informative than ROC-AUC, which can look good while precision is poor.

TH-MLE-04 · Medium & Tricky · *`random.seed(42)` is set at the top of the training script. Is training reproducible?*

> **Trap:** "Yes, the seed is fixed."
>
> Several independent RNGs exist: Python's `random`, NumPy's global state and any `Generator` objects, the framework's (`torch.manual_seed`, CUDA seeds), and each data-loader worker's seed. GPU kernels can be nondeterministic unless deterministic algorithms are enabled, set iteration order of strings changes with hash randomisation, and thread scheduling or library versions change results too (MLE-O02). Seed all of them, record versions, and pass explicit `Generator` objects.

TH-MLE-05 · Hard · *The model predicts in 5 ms in a notebook but the API takes 400 ms. What are the likely causes?*

> Model loading per request instead of once at startup (FastAPI `lifespan`); CPU inference inside an `async def` endpoint blocking the event loop (use a plain `def` endpoint, which runs in the threadpool, or a process pool); no batching; thread oversubscription (each of N workers running BLAS/torch with all cores; set `OMP_NUM_THREADS`); JSON serialisation of large arrays; input validation or feature fetching on the hot path; cold starts. Measure with timing spans around each stage before optimising (MLE-C01).

TH-MLE-06 · Hard & Tricky · *How does top-k cosine similarity search change from 10,000 to 100 million vectors?*

> **Trap:** "Same code, just a bigger matrix multiply."
>
> At 10k, exact search is fine: normalise vectors once, then `corpus @ query` gives cosine similarity, and `np.argpartition` finds top-k in O(n) (MLE-C02). At 100M, a float32 1536-dimension corpus is about 600 GB, and a brute-force scan per query is too slow, so you move to approximate nearest-neighbour indexes (HNSW, IVF-PQ in FAISS or a vector database), trading recall for latency and memory, possibly with quantisation (float16/int8/PQ). Metadata filters interact badly with ANN (filtering after search can return fewer than k results). Measure recall@k against exact search on a sample.

TH-BE · Python Backend Engineer

TH-BE-01 · Easy · *Which HTTP status codes would you return for: created, validation failure, missing auth, forbidden, not found, conflict, rate-limited, dependency down?*

> 201 Created, 422 (FastAPI's default for body validation; 400 for malformed requests), 401 Unauthorized (not authenticated), 403 Forbidden (authenticated but not allowed), 404 Not Found, 409 Conflict, 429 Too Many Requests (with `Retry-After`), 502/503 for a failing upstream or overload. Never 200 with an error body.

TH-BE-02 · Easy & Tricky · *Are PUT and POST interchangeable for updates?*

> **Trap:** "Yes, it's just convention."
>
> The semantics differ: PUT replaces the resource at a known URI and is idempotent (repeating it gives the same state), so clients and proxies may retry it. POST is not idempotent, so a retried POST can create duplicates unless you support idempotency keys (BE-C02). PATCH is a partial update; GET must be safe (no side effects).

TH-BE-03 · Medium · *What does FastAPI's dependency injection give you?*

> Declarative dependencies (`Depends(get_db)`) resolved per request, with sub-dependencies cached within a request; `yield` dependencies for setup and teardown (open a session, commit or roll back, close); shared auth and pagination parameters; and `app.dependency_overrides[get_db] = fake` for tests without patching. The equivalents in Flask are extensions and `g`, and in Django middleware and request attributes.

TH-BE-04 · Medium & Tricky · *An endpoint listing 50 orders with customer names runs 51 queries. Each query takes 2 ms, but the page is slow and gets slower as data grows. What's happening?*

> **Trap:** "The database is slow; add an index."
>
> The N+1 query problem: the ORM lazily loads each order's customer with its own query, so latency is dominated by round trips and grows with page size. Fix with eager loading (SQLAlchemy `selectinload` / `joinedload`, Django `select_related` / `prefetch_related`) or an explicit join, and add tests that assert the query count so it doesn't regress.

TH-BE-05 · Hard · *Explain transaction isolation levels and the "lost update" anomaly.*

> Read uncommitted, read committed (Postgres default), repeatable read, and serializable prevent progressively more anomalies: dirty reads, non-repeatable reads, phantoms, and write skew. A lost update happens when two transactions read the same row, compute new values, and both write, so one overwrites the other; it's possible at read committed. Prevent it with atomic updates (`SET stock = stock - 1`), `SELECT ... FOR UPDATE`, optimistic locking (a version column checked in the `WHERE` clause), or serializable isolation with retries. SQLite serialises writers entirely (BE-O02).

TH-BE-06 · Hard & Tricky · *You cache product data with cache-aside ("read cache, else load DB and set cache; on update, write DB then delete cache"). How can the cache still end up stale?*

> **Trap:** "Deleting on write guarantees freshness."
>
> Race: reader A misses, loads the *old* row from the DB, then writer B updates the DB and deletes the key, *then* A writes the old value into the cache, which stays stale until its TTL. Mitigations: always set a TTL; use versioned keys or compare-and-set; delay a second delete; or update the cache from a change stream. Hot-key expiry also causes stampedes (many requests recompute at once): use single-flight, early probabilistic refresh, or stale-while-revalidate.

TH-FS · Python Full Stack Engineer

TH-FS-01 · Easy · *What is CORS, and who enforces it?*

> The browser enforces the same-origin policy; CORS headers from the server (`Access-Control-Allow-Origin`, and `-Credentials`, `-Methods`) tell the browser which other origins may read responses. Non-simple requests get a preflight `OPTIONS`. CORS protects users' browsers, not your server: `curl` ignores it, so it's never an authorisation mechanism.

TH-FS-02 · Easy & Tricky · *The API returns `"created_at": "2026-09-24T10:00:00"` from a naive `datetime`. What does the frontend show?*

> **Trap:** "10:00, the same time for everyone."
>
> JavaScript's `new Date("2026-09-24T10:00:00")` interprets an ISO date-time *without an offset* as the user's local time, so users in different timezones see different instants and any UTC assumption is off by hours. Store and return aware UTC datetimes (`"2026-09-24T10:00:00Z"`) and convert to local time only in the UI. Plain `json.dumps` can't serialise datetimes at all (FS-O01).

TH-FS-03 · Medium · *Offset vs cursor pagination: tradeoffs?*

> Offset (`LIMIT 20 OFFSET 400`) is simple and supports jumping to page N, but the database still scans and discards the skipped rows, and concurrent inserts or deletes shift pages, causing duplicates or skipped items (FS-O02). Cursor (keyset) pagination (`WHERE (created_at, id) < (:t, :id) ORDER BY created_at DESC, id DESC LIMIT 20`) is stable under writes and uses an index, but it can't jump to arbitrary pages and needs a unique tie-breaker (FS-C01). Infinite scroll and feeds want cursors.

TH-FS-04 · Medium & Tricky · *Jinja2 autoescaping is on. Is the app safe from XSS?*

> **Trap:** "Yes, autoescaping escapes everything."
>
> It escapes HTML *text and quoted attribute* contexts. It doesn't make `href="{{ url }}"` safe from `javascript:` URLs, doesn't make data inside `<script>` blocks safe (use the `tojson` filter), is bypassed by `|safe/Markup()` on user data, and does nothing for client-side rendering via `innerHTML` or `dangerouslySetInnerHTML`. Add URL-scheme allow-lists, a Content-Security-Policy, and safe DOM APIs (FS-O03).

TH-FS-05 · Hard · *How do you accept file uploads safely?*

> Enforce a size limit while streaming (reject early, don't buffer the whole body); validate type by inspecting content (magic bytes), not the client's `Content-Type` or extension; never use the client's filename as a path (path traversal); generate your own storage key and store outside the web root (object storage); scan if needed; serve downloads with `Content-Disposition: attachment` and a safe content type. For large files, issue presigned URLs so the browser uploads directly to S3/GCS. FastAPI's `UploadFile` spools to disk above a threshold.

TH-FS-06 · Hard & Tricky · *What's different about securing and scaling WebSockets compared to REST endpoints?*

> **Trap:** "CORS protects WebSockets like any other request."
>
> CORS doesn't apply to WebSocket handshakes: a malicious site can open a socket to your server with the user's cookies (cross-site WebSocket hijacking), so check the `Origin` header and authenticate explicitly (a token in the first message or a short-lived ticket). Connections are long-lived and pinned to one instance, so broadcasting across instances needs pub/sub (Redis, NATS); you also need heartbeats (ping/pong), backpressure for slow clients, and reconnection with resume (last event ID) so clients don't miss messages.