Module 15 of 20

Type hints and code quality

The hint vocabulary from generics and unions to TypeVar, Generic, Literal and TypedDict; what a type checker verifies and how narrowing works; PEP 8 layout, naming and docstrings with ruff and black; logging with levels, loggers, handlers and formats; and the idioms that make code Pythonic.

A program that runs is not yet a program others can work on. This module is about the layer that makes code maintainable: type hints precise enough for a checker to catch the `None` you forgot to test, a style that every reader already knows, logging that tells an operator what happened without a debugger, and the idioms that say what the code means rather than how it grinds through it. None of it is enforced by the interpreter; all of it is enforced by the tools a serious codebase runs on every commit.

Type hints in depth gives the vocabulary — built-in generics, `Iterable` / `Sequence` / `Mapping` for parameters, unions and `Optional`, `Callable`, `TypeVar` and `Generic`, `ClassVar`, `Final`, `Literal`, `TypedDict`, `NewType`, `Any`. Static analysis covers what `mypy` checks, narrowing, `cast`, `reveal_type`, `TYPE_CHECKING`, strictness and hints at run time. Code style covers PEP 8 layout and naming, PEP 257 docstrings, comments, and the `ruff` / `black` / `pre-commit` workflow. Logging covers levels, the module-logger idiom, `basicConfig`, the hierarchy, formats, lazy arguments, handlers, `log.exception` and the library rule. Writing idiomatic Python gathers the idioms and anti-patterns and works through a refactoring.

The exercises are whole programs: a generic `Stack[T]`, signatures rendered from `inspect`, hints observed at run time with and without the `annotations` future import, a validator that reads `get_type_hints` and understands unions and `Literal`, a naming and idiom linter, a docstring checker built on `ast`, a level-filtered logger writing to stdout, a logger with two handlers and two formats, the refactored totals report, and an idiom drill. The checkpoint adds a hint-driven validator with nested generics, a linter-lite over a source file, and a logging pipeline routed by level into two buffers.

CONTENTS

1	Type hints in depth — generics, unions, Callable, TypeVar and Protocol	15 min
2	Static analysis with mypy — narrowing, strictness and the run-time view of hints	13 min

3	Code style and PEP 8 — layout, naming, imports, docstrings and the tools that enforce them	13 min
4	logging — levels, loggers, handlers and formats	14 min
5	Writing idiomatic Python — the idioms, the anti-patterns and a refactoring	14 min
✓	Checkpoint — Type hints and code quality	22 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 15 MIN

Type hints in depth — generics, unions, Callable, TypeVar and Protocol

Module 4 introduced annotations as documentation that tools can check. This lesson is the vocabulary a real codebase uses: the built-in generics and what to hint on parameters versus returns, unions and `Optional`, `Callable` signatures, `TypeVar` for functions and classes that are generic over a type, `Generic` classes, `ClassVar` and `Final`, `TypedDict` for dict-shaped data, `Literal` for a fixed set of strings, and `Any` as the deliberate escape hatch. All of it runs on 3.11; the 3.12 `type` statement and `def f[T]` syntax are described at the end as reading only.

The built-in generics

```
def total(xs: list[int]) → int: ...
def index(names: dict[str, int]) → None: ...
def pair() → tuple[int, str]: ...            # a 2-tuple with those types
def many() → tuple[int, ...]: ...            # any length, all ints
def uniq(xs: set[str]) → frozenset[str]: ...
```

Since 3.9 the built-in types take subscripts directly; `typing.List` and friends are the pre-3.9 spellings, still accepted, no longer needed. A subscript is a hint only — `list[int]` at run time is a `types.GenericAlias`, and `isinstance(x, list[int])` is a `TypeError`; `isinstance(x, list)` is the run-time check.

Parameters wide, returns narrow

```
from collections.abc import Iterable, Sequence, Mapping, Iterator, Callable

def mean(xs: Iterable[float]) → float: ...          # accepts list, tuple, se
def first(xs: Sequence[str]) → str: ...             # needs indexing: list, t
def lookup(table: Mapping[str, int], k: str) → int: ...
def evens(n: int) → Iterator[int]: ...              # a generator function's
def build() → list[str]: ...                        # the concrete type you c
```

A parameter hinted `list[int]` rejects a tuple; `Iterable[int]` accepts anything the function can loop over. Hint the *minimum* the function needs — `Iterable` if it only loops, `Sequence` if it indexes or slices, `Mapping` if it looks up, `MutableSequence` / `MutableMapping` if it modifies — and return the concrete type. The abstract names live in `collections.abc` (the `typing` aliases are deprecated).

Unions and Optional

```
def parse(text: str) → int | None: ...                  # 3.10 syntax
def load(source: str | Path) → bytes: ...
from typing import Optional, Union
def parse_old(text: str) → Optional[int]: ...           # == int | None
def load_old(source: Union[str, Path]) → bytes: ...
```

`X | None` is the honest type of anything that can be absent, and a checker then requires a `None` test before use. `Optional[X]` means exactly `X | None` — *not* "an optional parameter"; a parameter with a default is optional whatever its type.

Callable

```
from collections.abc import Callable

def apply(f: Callable[[int, str], bool], n: int, s: str) → bool:
    return f(n, s)

Handler = Callable[[str], None]                 # an alias
def on_event(handler: Handler) → None: ...
def make_adder(k: int) → Callable[[int], int]: ...
Callable[..., int]                              # any arguments, returns int
```

`Callable[[arg types], return type]` describes functions, lambdas, bound methods and callable objects alike. For keyword arguments or `*args` precision, `Protocol` with a `__call__` (Module 9) or `ParamSpec` (3.10, reading) are the tools.

TypeVar and generic functions

```python
from typing import TypeVar
from collections.abc import Sequence

T = TypeVar("T")

def first(xs: Sequence[T]) -> T:                # whatever element type goes in
    return xs[0]

def pairwise_max(a: T, b: T) -> T: ...

N = TypeVar("N", int, float)                    # constrained: only these
def double(x: N) -> N: ...

from typing import Hashable
H = TypeVar("H", bound=Hashable)                # bound: anything hashable
def dedupe(xs: Sequence[H]) -> list[H]: ...
```

A `TypeVar` links the types of parameters and return: `first(["a"])` is `str`, `first([1])` is `int`, and the checker knows. Constraints limit to a list of types; `bound=` limits to subtypes of one.

Generic classes

```python
from typing import Generic, TypeVar

T = TypeVar("T")

class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

s: Stack[int] = Stack()
s.push(1)
s.push("x")            # checker: error
```

`Generic[T]` makes the class subscriptable; `Stack[int]` is a stack whose `push` / `pop` are checked for ints. The standard containers are generic in exactly this way.

ClassVar, Final, Literal, TypedDict, NewType

```
from typing import ClassVar, Final, Literal, TypedDict, NewType, Any

class Config:
    instances: ClassVar[int] = 0          # a class attribute, not a per-inst
    name: str

MAX_RETRIES: Final = 3                    # the checker forbids reassignment

def open_mode(mode: Literal["r", "w", "a"]) -> None: ...   # only these strin

class Movie(TypedDict):                   # a dict with known string keys and
    title: str
    year: int

m: Movie = {"title": "Heat", "year": 1995}

UserId = NewType("UserId", int)           # a distinct type for the checker, 
def fetch(uid: UserId) -> None: ...
fetch(UserId(5)); fetch(5)                # the second is a checker error

def log(value: Any) -> None: ...          # Any: anything; disables checking 
```

`TypedDict` types JSON-shaped data without converting it to a class; `Literal` types the string flags every API has; `NewType` stops ids of different kinds being mixed; `Any` is the explicit opt-out, and `object` is the type meaning "anything, but you may do nothing with it until you narrow it".

3.12 and later, as reading

```
type Pair = tuple[int, int]               # 3.12: a type alias statement
def first[T](xs: Sequence[T]) -> T: ...   # 3.12: PEP 695 — no TypeVar declar
class Stack[T]: ...
```

On 3.11, aliases are ordinary assignments (`Pair = tuple[int, int]`, or `Pair: TypeAlias = …`) and generics use `TypeVar`. The meaning is identical; only the spelling changed.

Pitfalls

- `isinstance(x, list[int])` — use `list`.
- `list[int]` on a parameter that only iterates.
- Confusing `Optional[X]` with "has a default".
- Forgetting that hints are not enforced — validate at boundaries (Module 10).
- A `TypeVar` used in only one place (it links nothing; use the plain type).
- `Any` spreading through a codebase until nothing is checked.

Key takeaways

- `list[int]`, `dict[str, int]`, `tuple[int, ...]`, `X | None`, `Callable[[…], R]` are the everyday hints; abstract parameter types from `collections.abc`.
- Hint parameters as wide as the function allows and returns as narrow as it produces.
- `TypeVar` links parameter and return types; `Generic[T]` makes classes generic; constraints and bounds limit it.
- `ClassVar`, `Final`, `Literal`, `TypedDict`, `NewType` express intent the checker enforces; `Any` opts out.
- 3.12's `type` and `def f[T]` are spellings of the same ideas.

EXERCISE 1

A generic stack

Write `Stack(Generic[T])` with hinted `push(item: T) → None`, `pop() → T`, `peek() → T` and `__len__`. Drive it with commands `push v`, `pop`, `peek`, `len` (`empty` when a pop or peek finds nothing). Before the commands, print `str(Stack[int])` to show the parameterised class, and after them print the annotations of `push` as `name: hint` pairs from `__annotations__`.

Input: commands. **Output:** the class line, the command results, then the annotation lines.

```
push 1
push 2
peek
len
```

prints

```
__main__.Stack[int]
2
2
item: ~T
return: None
```

INPUT 1

```
push 1
push 2
peek
len
```

OUTPUT 1

```
__main__.Stack[int]
2
2
item: ~T
return: None
```

REFERENCE SOLUTION

```python
import sys
from typing import Generic, TypeVar

T = TypeVar("T")


class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

    def peek(self) -> T:
        return self._items[-1]

    def __len__(self) -> int:
        return len(self._items)


print(Stack[int])
stack: Stack[int] = Stack()
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "push":
        stack.push(int(args[0]))
    elif cmd in ("pop", "peek"):
        print(getattr(stack, cmd)() if len(stack) else "empty")
    elif cmd == "len":
        print(len(stack))
for name, hint in Stack.push.__annotations__.items():
    print(f"{name}: {hint}")
```

EXERCISE 2

Signatures on display

Four hinted functions are defined in the program. Read function names and print each one's signature exactly as `str(inspect.signature(f))` renders it (which includes the hints and defaults), or `unknown` for a name not defined.

Input: lines. **Output:** one line per name.

```
total
lookup
nope
```

prints

```
(xs: collections.abc.Iterable[float]) → float
(table: collections.abc.Mapping[str, int], key: str, default: int | None = None) →
unknown
```

INPUT 1

```
total
lookup
nope
```

OUTPUT 1

```
(xs: collections.abc.Iterable[float]) →
float
(table: collections.abc.Mapping[str,
int], key: str, default: int | None =
None) → int | None
unknown
```

REFERENCE SOLUTION

```
import inspect
import sys
from collections.abc import Callable, Iterable, Mapping
from typing import Literal


def total(xs: Iterable[float]) -> float:
    return sum(xs)


def lookup(table: Mapping[str, int], key: str, default: int | None = None) -> int |
    return table.get(key, default)


def apply(f: Callable[[int], int], values: list[int]) -> list[int]:
    return [f(v) for v in values]


def open_mode(mode: Literal["r", "w"] = "r") -> str:
    return mode


FUNCTIONS = {"total": total, "lookup": lookup, "apply": apply, "open_mode": open_moc
for line in sys.stdin:
    name = line.strip()
    f = FUNCTIONS.get(name)
    print(str(inspect.signature(f)) if f else "unknown")
```

QUIZ

1. Which hint should a function that only loops over its argument use?

A. `list[int]`

B. `Iterable[int]`

C. `Sequence[int]`

D. `Any`

2. What does `Optional[int]` mean?

A. The parameter may be omitted

B. `int | None`

C. An int that may be unset later

D. `int` or any other type

3. What does a `TypeVar` in a signature express?

A. Any type at all

B. That the marked positions share one type — e.g. the element type in equals the return type out

C. A class variable

D. A constant

4. What is `Literal["r", "w"]` for?

A. Regular expressions

B. Restricting a value to exactly those strings, checked statically

C. String constants

D. File modes only

5. What does `isinstance(x, list[int])` do?

A. Checks a list of ints

B. Raises `TypeError` — subscripted generics cannot be used with `isinstance`

C. Returns `True` for any list

D. Returns `False`

ANSWER KEY · TYPE HINTS IN DEPTH — GENERICS, UNIONS, CALLABLE, TYPEVAR AND PROTOCOL

1. B Parameters as wide as the function allows: `Iterable` accepts lists, tuples, sets, generators and ranges.

2. B Optionality of a parameter comes from a default value; `Optional` is only about `None`.

3. B It links the types; a `TypeVar` used once links nothing and should be a plain type.

4. B The checker rejects `open_mode("x")` at analysis time.

5. B Run-time checks use the plain class; element types are not checked by `isinstance`.

LESSON 2 · 13 MIN

Static analysis with mypy — narrowing, strictness and the run-time view of hints

A type checker reads a program without running it and reports every place where a value could have a type the code does not handle: an `int | None` used as an `int`, a function called with the wrong argument, a method that does not exist on the declared type. `mypy` is the reference checker (`pyright` is the other widely used one, built into VS Code's Pylance). This lesson covers what a checker does and does not catch, *narrowing* — the way `isinstance` and `is None` tests make a union usable — `cast`, `reveal_type`, `TYPE_CHECKING`, `# type: ignore`, the strictness levels, and the run-time side: `__annotations__`, `get_type_hints` and the string annotations of `from __future__ import annotations`.

What a checker does

```
$ pip install mypy
$ mypy app.py
app.py:12: error: Argument 1 to "area" has incompatible type "str"; expected "
app.py:20: error: Item "None" of "int | None" has no attribute "bit_length"  [
Found 2 errors in 1 file (checked 1 source file)
```

It infers the type of every expression from literals, annotations and library stubs, follows values through assignments and calls, and reports mismatches with a code (`[arg-type]`) and a line. Unannotated functions are checked as taking and returning `Any` by default — which means their bodies are barely checked — so a codebase gets value in proportion to how much of its public surface is hinted. It does not run the code, so it cannot know a list is empty or a file is missing; it checks *shape*, not *state*.

Narrowing

The most useful thing a checker does is refuse to let an `X | None` be used before it is tested, and the tests it understands are called narrowing:

```python
def size(x: int | None) -> int:
    if x is None:               # after this line, x is int
        return 0
    return x.bit_length()

def describe(v: int | str) -> str:
    if isinstance(v, int):      # v is int here
        return f"{v:d}"
    return v.upper()            # and str here

def first(xs: list[str]) -> str:
    assert xs, "empty"          # assert narrows too (and, unlike an if, does
    return xs[0]

match cmd:
    case str():                 # class patterns narrow
        ...
```

`is None` / `is not None`, `isinstance`, `callable`, truthiness on `X | None`, `in` against a `Literal` tuple, early `return` / `raise` in a branch, and `match` class patterns all narrow. A custom test function does not — unless it is declared `→ TypeGuard[T]` (3.10), which tells the checker "when this returns `True`, the argument is a `T`".

---

cast, reveal_type, ignore

```python
from typing import cast, reveal_type

value = cast(int, some_object_the_checker_thinks_is_Any)   # no run-time effe
reveal_type(value)                                          # the checker pri
result = legacy_call()  # type: ignore[no-untyped-call]     # silence one err
```

`cast` is a promise, not a check — if it is wrong, the run-time error appears somewhere else. `reveal_type` is the debugging tool for "what does the checker think this is?". A `# type: ignore` with the error code in brackets is the honest way to accept a known limitation; a bare `# type: ignore` hides everything on that line.

TYPE_CHECKING and forward references

```python
from __future__ import annotations          # annotations become strings, eva
from typing import TYPE_CHECKING

if TYPE_CHECKING:                           # imports only the checker sees —
    from .models import User

def greet(user: User) -> str: ...

class Node:
    def link(self, other: Node) -> None: ...   # a class referring to itself:
```

`from __future__ import annotations` (3.7) stores every annotation as a string instead of evaluating it, which allows forward references and cheap imports; `typing.get_type_hints(obj)` evaluates them back to objects when a library needs the real types at run time (dataclasses do this).

Strictness

`mypy --strict` turns on the checks that matter for a mature codebase: every function must be annotated (`disallow-untyped-defs`), calls into unannotated code are errors, `Any` returns are flagged, and optional values must be handled. New projects should start strict; existing ones tighten one flag at a time, per package, with a `[tool.mypy]` table in `pyproject.toml` (Module 12). The checker also needs types for third-party libraries: most ship them inline (a `py.typed` marker) or as `types-*` stub packages; `ignore_missing_imports` bridges the gap.

Hints at run time

```python
def area(w: float, h: float) -> float: ...
area.__annotations__                    # {'w': <class 'float'>, 'h': <class
import typing, inspect
typing.get_type_hints(area)             # the same, with string annotations re
inspect.signature(area).parameters["w"].annotation
```

Nothing in the interpreter checks them, but libraries do: `dataclasses` builds fields from them, `functools.singledispatch` registers by them, `argparse` -like tools (`typer`), validation libraries (`pydantic`) and serialisers read them. Module 4's checker exercise was the small

version of that idea. The rule: annotations are data about the code, available to any tool that wants them, enforced by none unless it chooses to.

The workflow

Write hints as you write functions — the signature is the cheapest place; run the checker in the editor and in CI (`mypy src/`), treat its errors as bugs to fix or limitations to annotate with a coded ignore, and use `reveal_type` when a union does not narrow as expected. A checker's errors are usually right: a `None` it will not let you dereference is a `None` your code was about to crash on.

Pitfalls

- Believing a passing check means correct behaviour; it means consistent types.
- Unannotated functions, which the checker treats as `Any` and skips.
- Testing with `== None` (narrows) versus a helper `is_missing(x)` (does not, without `TypeGuard`).
- Bare `# type: ignore` hiding new errors on the same line.
- `cast` used to silence an error that is real.
- Forgetting `from __future__ import annotations` when a class refers to itself in a hint (or quoting it: `"Node"`).

Key takeaways

- A checker verifies type consistency without running the code; hint the public surface or it checks little.
- `is None`, `isinstance`, `assert`, early returns and `match` narrow unions; `TypeGuard` lets a custom test narrow.
- `cast` asserts to the checker only; `reveal_type` shows its inference; `# type: ignore[code]` accepts a known limitation.
- `TYPE_CHECKING` and `from __future__ import annotations` handle cycles and forward references; `get_type_hints` resolves strings at run time.
- Run `mypy --strict` in the editor and CI; annotations are data that libraries read and the interpreter ignores.

EXERCISE 1

Hints at run time

With `from __future__ import annotations` in effect, define `f(x: int, y: str | None = None) → list[int]`. Print `f.__annotations__["y"]` (a string under the future import), then the resolved hint from `typing.get_type_hints(f)["y"]`, then `typing.TYPE_CHECKING`. Then read a literal, apply `typing.cast(int, value)` and print the value's actual run-time type name to show that `cast` changes nothing.

Input: one literal line (parse with `ast.literal_eval`). **Output:** `raw <text>`, `resolved <hint>`, `type_checking <bool>`, `cast <value> is <type>`.

```
"5"
```

prints

```
raw str | None
resolved str | None
type_checking False
cast 5 is str
```

INPUT 1

```
"5"
```

OUTPUT 1

```
raw str | None
resolved str | None
type_checking False
cast 5 is str
```

REFERENCE SOLUTION

```python
from __future__ import annotations

import ast
import typing


def f(x: int, y: str | None = None) -> list[int]:
    return [x]


print(f"raw {f.__annotations__['y']}")
print(f"resolved {typing.get_type_hints(f)['y']}")
print(f"type_checking {typing.TYPE_CHECKING}")
value = typing.cast(int, ast.literal_eval(input()))
print(f"cast {value} is {type(value).__name__}")
```

EXERCISE 2

Narrow by hand

Implement `describe(v: int | str | list[int] | None) → str` using the narrowing tests a checker understands — `is None`, then `isinstance` for each branch — returning `none`, `int <v>`, `str of <len>` or `list sum <s>`; any other type returns `unsupported <type name>`. Read literals one per line and print the description.

Input: literal lines. **Output:** one line per literal.

```
None
42
"hey"
[1, 2, 3]
2.5
```

prints

```
none
int 42
str of 3
list sum 6
unsupported float
```

INPUT 1

```
None
42
"hey"
[1, 2, 3]
2.5
```

OUTPUT 1

```
none
int 42
str of 3
list sum 6
unsupported float
```

REFERENCE SOLUTION

```python
import ast
import sys


def describe(v: int | str | list[int] | None) -> str:
    if v is None:
        return "none"
    if isinstance(v, bool):
        return "unsupported bool"
    if isinstance(v, int):
        return f"int {v}"
    if isinstance(v, str):
        return f"str of {len(v)}"
    if isinstance(v, list):
        return f"list sum {sum(v)}"
    return f"unsupported {type(v).__name__}"


for line in sys.stdin:
    print(describe(ast.literal_eval(line.strip())))
```

QUIZ

1. What does a type checker verify?

A. That the program produces correct output

B. That types are used consistently — without running the code

C. That tests pass

D. That the code follows PEP 8

2. After `if x is None: return 0`, what does the checker believe about `x: int | None`?

A. Still `int | None`

B. `int` — the test narrowed it

C. `None`

D. `Any`

3. What does `typing.cast(int, value)` do at run time?

A. Converts `value` to an int

B. Returns `value` unchanged; only the checker is told the type

C. Raises if `value` is not an int

D. Rounds it

4. What is `typing.TYPE_CHECKING` at run time?

A. `True`

B. `False` — its `if` block imports only the checker sees

C. `None`

D. Undefined

5. What does `from __future__ import annotations` change?

A. Enables the checker

B. Stores annotations as strings, evaluated lazily — allowing forward references

C. Enforces hints at run time

D. Disables `Any`

ANSWER KEY · STATIC ANALYSIS WITH MYPY — NARROWING, STRICTNESS AND THE RUN-TIME VIEW OF HINTS

1. B It checks shape, not state; an empty list or a missing file is invisible to it.

2. B Narrowing follows `is None`, `isinstance`, `assert`, early returns and `match` patterns.

3. B A wrong `cast` surfaces later as a run-time error elsewhere.

4. B It breaks import cycles and avoids heavy imports needed only for hints.

5. B `get_type_hints` evaluates the strings back to objects when a library needs them.

LESSON 3 · 13 MIN

Code style and PEP 8 — layout, naming, imports, docstrings and the tools that enforce them

PEP 8 is the style guide the Python community converged on in 2001 and has followed since; reading code that follows it costs nothing, and reading code that does not costs a little on every line. The guide's rules fall into layout, naming, expressions and imports, and nearly all of them are now enforced or applied automatically by two tools — `ruff` (a linter) and `black` or `ruff format` (a formatter) — so the human part is knowing *why* the rules exist and the few that a tool cannot decide for you. This lesson covers the rules that matter, the PEP 257 docstring conventions, the linter/formatter workflow, and what "readable" means beyond compliance.

Layout

Four spaces per indent level, never tabs.

Line length 79 by PEP 8; most projects choose 88 (black's default) or 100 and say so in `pyproject.toml`. The point is a limit, not the number.

Two blank lines before top-level `def` / `class`; **one** between methods.

Spaces around binary operators and after commas; none inside brackets or before a call's parentheses: `f(a, b)`, `x = y + 1`, `xs[i]`, `d = {"k": v}`. No spaces around `=` in keyword arguments and defaults: `f(x=1)`, `def g(n=0)`.

Line breaks inside brackets, with a trailing comma when items go one per line.

Imports at the top, one per line, grouped standard library / third-party / local, each group alphabetical (Module 12).

Naming

Kind	Style	Example
modules, packages	`lowercase`, short	`parser.py`, `shop/`
functions, methods, variables	`snake_case`	`total_price`, `is_valid`
classes, exceptions	`PascalCase`	`OrderItem`, `ValidationError`
constants	`UPPER_SNAKE`	`MAX_RETRIES`
internal names	`_leading_underscore`	`_cache`
avoid clashing with a keyword	trailing underscore	`class_`
single-character loop indexes	`i`, `j`, `k`; never `l`, `O`, `I`	

Names describe *what*, not *how*: `active_users` not `list_of_users_filtered`; `is_` / `has_` / `can_` prefixes for booleans; verbs for functions that act (`save`, `parse`), nouns or adjectives for those that compute (`total`, `is_valid`); no type in the name (`users_list`, `str_name`) — the hint says the type.

Expressions

```python
if not xs:                      # not: if len(xs) == 0
if x is None:                   # not: if x == None
if flag:                        # not: if flag == True
if isinstance(x, int):          # not: if type(x) == int
value = d.get(key, default)     # not: if key in d: ... else: ...
"".join(parts)                  # not: s += part in a loop
for i, x in enumerate(xs):      # not: for i in range(len(xs))
with open(...) as f:            # always
```

Comparisons to singletons use `is`; truthiness replaces explicit emptiness checks; comprehensions replace `map` / `filter` with lambdas; `return` a value rather than mutating and returning `None` unless the method's job is mutation.

Docstrings (PEP 257)

```python
def parse_duration(text: str) -> int:
    """Return the number of seconds denoted by `text`, such as "1h30m".

    Units are h, m and s, each optional, in that order. Raises ValueError
    for anything else, including a negative or empty value.
    """
```

Triple double quotes; a one-line summary in the imperative that fits on the first line and ends with a full stop; a blank line; then details a caller needs — arguments whose meaning the names do not convey, the return value's shape, exceptions raised. Public modules, classes and functions get docstrings; a private helper may have one line or none. Google style (`Args:` / `Returns:` / `Raises:` sections) and NumPy style are structured variants documentation tools render; pick one per project. A docstring is not a comment: comments explain *why* a piece of code is the way it is, docstrings explain *what* a function does for its caller.

Comments

A comment earns its place by saying something the code cannot: the reason for a non-obvious choice, a reference to a specification or an issue, an invariant a loop maintains, a warning about a trap. `# increment i` is noise; `# retry once: the first call after wake-up fails on this firmware` is worth its line. Delete commented-out code — version control remembers it. A `TODO` names an owner and a reason.

The tools

- **`ruff check .`** — a linter: unused imports and variables, undefined names, comparisons to `None` with `==`, mutable defaults, bare `except`, shadowed built-ins and hundreds more, in milliseconds; `ruff check --fix` repairs what is safe. It subsumes `flake8`, `pyflakes`, `isort` and most of `pylint`.
- **`ruff format .`** or **`black .`** — a formatter: rewrites layout (spacing, line breaks, quotes) to one canonical form, so style is never discussed in review. Both are deliberately unconfigurable beyond line length.
- **`mypy`** — types (previous lesson).
- **`pre-commit`** — runs them on every commit; CI runs them on every push.

Configuration lives in `pyproject.toml` (`[tool.ruff]`, `[tool.black]`), so a checkout carries its rules. The workflow is: write, save, the editor formats and lints, fix what is flagged, commit. Style stops being a matter of taste and becomes a property of the repository.

Beyond compliance

PEP 8-clean code can still be unreadable. The judgement calls a tool cannot make: a function that does one thing and is named for it; a module with a single purpose; early returns instead of nesting (Module 3); no clever one-liners where two clear lines would do; data structures chosen so the code that uses them is simple; consistency with the surrounding code above consistency with your preference. "Readability counts" is the Zen's line, and the test is whether a reader who did not write it can predict what it does from its shape.

Pitfalls

Tabs, or mixed indentation.

`== None`, `== True`, `type(x) == T`, `len(xs) == 0`.

Names with types in them, or single letters outside a loop index.

Commented-out code and `# obvious` comments.

Ignoring the linter, or disabling rules globally to silence it.

Reformatting files you did not otherwise change (noisy diffs).

Key takeaways

Four spaces, a line limit, two blank lines between top-level definitions, spaces around operators, imports grouped at the top.

`snake_case` functions, `PascalCase` classes, `UPPER_SNAKE` constants, `_internal` names; describe what, not how or type.

`is None`, truthiness, `isinstance`, `get`, `join`, `enumerate`, `with` — the idioms PEP 8 expects.

PEP 257 docstrings: imperative summary, blank line, details; comments say why.

`ruff` lints, `black` / `ruff format` formats, `mypy` types, `pre-commit` and CI run them; configure in `pyproject.toml`.

EXERCISE 1

A naming and idiom linter

Read Python source lines and report, per line, the first rule it breaks: `tab` (a tab in the indentation), `long` (more than 79 characters), `def-name` (a `def` whose name is not `snake_case`), `class-name` (a `class` whose name is not `PascalCase`), `none-compare` (`== None` or `!= None`), `bool-compare` (`== True` or `== False`), `type-compare` (`type(...) ==`), `len-zero` (`len(...) == 0`). Print `line <n>: <rule>` for each offending line, or `clean` if none.

Input: source lines. **Output:** the report.

```
def getUser(x):
    if x == None:
        return None
class order_item:
    pass
```

prints

```
line 1: def-name
line 2: none-compare
line 4: class-name
```

INPUT 1

```
def getUser(x):
    if x == None:
        return None
class order_item:
    pass
```

OUTPUT 1

```
line 1: def-name
line 2: none-compare
line 4: class-name
```

REFERENCE SOLUTION

```
import re
import sys

SNAKE = re.compile(r"[a-z_][a-z0-9_]*")
PASCAL = re.compile(r"[A-Z][A-Za-z0-9]*")


def check(line):
    indent = line[: len(line) - len(line.lstrip())]
    if "\t" in indent:
        return "tab"
    if len(line) > 79:
        return "long"
    m = re.match(r"\s*def\s+(\w+)", line)
    if m and not SNAKE.fullmatch(m.group(1)):
        return "def-name"
    m = re.match(r"\s*class\s+(\w+)", line)
    if m and not PASCAL.fullmatch(m.group(1)):
        return "class-name"
    if re.search(r"[=!]=\s*None\b", line):
        return "none-compare"
    if re.search(r"==\s*(True|False)\b", line):
        return "bool-compare"
    if re.search(r"\btype\([^)]*\)\s*==", line):
        return "type-compare"
    if re.search(r"\blen\([^)]*\)\s*==\s*0\b", line):
        return "len-zero"
    return None


found = False
for n, line in enumerate(sys.stdin, start=1):
    line = line.rstrip("\n")
    rule = check(line)
    if rule:
        print(f"line {n}: {rule}")
        found = True
if not found:
    print("clean")
```

EXERCISE 2

Docstring checker

Read a Python module's source from standard input, parse it with `ast.parse`, and for every top-level function and class (in source order) report on its docstring via `ast.get_docstring`: `missing`

when there is none, `no period` when the first line does not end with `.`, `lowercase` when the first line does not start with an upper-case letter, otherwise `ok`. Then print `documented <k>/<n>`.

Input: Python source. **Output:** `<name>: <verdict>` per definition, then the summary.

```
def area(w, h):
    """Return the area."""
    return w * h

def perimeter(w, h):
    return 2 * (w + h)

class Shape:
    """a shape"""
```

prints

```
area: ok
perimeter: missing
Shape: no period
documented 1/3
```

INPUT 1

```
def area(w, h):
    """Return the area."""
    return w * h

def perimeter(w, h):
    return 2 * (w + h)

class Shape:
    """a shape"""
```

OUTPUT 1

```
area: ok
perimeter: missing
Shape: no period
documented 1/3
```

REFERENCE SOLUTION

```
import ast
import sys

tree = ast.parse(sys.stdin.read())
ok = total = 0
for node in tree.body:
    if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
        total += 1
        doc = ast.get_docstring(node)
        if doc is None:
            verdict = "missing"
        else:
            first = doc.strip().splitlines()[0]
            if not first.endswith("."):
                verdict = "no period"
            elif not first[0].isupper():
                verdict = "lowercase"
            else:
                verdict = "ok"
                ok += 1
        print(f"{node.name}: {verdict}")
print(f"documented {ok}/{total}")
```

QUIZ

1. Which name follows PEP 8 for a function?

A. `getUserName`

B. `get_user_name`

C. `GetUserName`

D. `GET_USER_NAME`

2. What does a formatter such as `black` do?

A. Finds bugs

B. Rewrites layout to one canonical style so formatting is never debated

C. Checks types

D. Runs tests

3. Which is the PEP 8 way to test for an empty list?

A. `if len(xs) == 0:`

B. `if not xs:`

C. `if xs == []:`

D. `if xs.empty():`

4. What should a comment contain?

A. A restatement of the next line

B. The *why* — a reason, a reference, an invariant, a warning

C. Commented-out old code

D. The author's name

5. What is the first line of a PEP 257 docstring?

A. The function's name

B. An imperative summary ending in a full stop, on one line

C. A list of parameters

D. A blank line

ANSWER KEY · CODE STYLE AND PEP 8 — LAYOUT, NAMING, IMPORTS, DOCSTRINGS AND THE TOOLS THAT ENFORCE THEM

1. B Functions and variables are `snake_case`; classes `PascalCase`; constants `UPPER_SNAKE`.

2. B Linting (`ruff check`) finds problems; formatting (`black` / `ruff format`) normalises appearance.

3. B Truthiness of containers is the idiom; the others say the same thing longer.

4. B The code says what; the comment says why. Version control remembers old code.

5. B Then a blank line, then details a caller needs.

LESSON 4 · 14 MIN

logging — levels, loggers, handlers and formats

`print` is for a program's output; `logging` is for its diary — what it did, what went wrong, in what order — written to stderr, a file, or a service, at a verbosity the operator chooses without editing the code. The module has more parts than it first appears to need, and this lesson explains them in the order they matter: levels, the module-level logger idiom, `basicConfig` with a format, the logger hierarchy, handlers and formatters, `logging.exception` for tracebacks, lazy `%s` formatting, and the reason a library never configures logging.

Levels and the first call

```
import logging

logging.basicConfig(level=logging.INFO, format="%(levelname)s:%(name)s:%(messa
log = logging.getLogger(__name__)

log.debug("cache size %d", len(cache))        # below INFO: dropped
log.info("loaded %d rows", n)                 # INFO:__main__:loaded 42 rows
log.warning("retrying %s", url)
log.error("failed to save %s", path)
log.critical("out of disk")
```

Five levels, `DEBUG` (10) < `INFO` (20) < `WARNING` (30) < `ERROR` (40) < `CRITICAL` (50). A logger drops messages below its threshold, so a program logs `debug` details freely and an operator turns them on with one setting when needed. The default level is `WARNING`, which is why `log.info` prints nothing until `basicConfig` sets `INFO`.

Loggers and the hierarchy

`logging.getLogger(__name__)` at the top of every module gives it a logger named after its import path — `shop.pricing`, `shop.utils.text`. Names form a tree by the dots: a setting on `shop` applies to `shop.pricing`; the root logger (`logging.getLogger()`) is the top. That is how an operator says "everything at `WARNING`, but `shop.pricing` at `DEBUG`":

```
logging.getLogger("shop.pricing").setLevel(logging.DEBUG)
```

Messages propagate up the tree to the handlers attached at each level, which is why one `basicConfig` on the root is enough for a whole program. Never `getLogger("myapp")` with a hand-written name in every module — `__name__` is the convention that makes the hierarchy match the code.

Format

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s %(levelname)-8s %(name)s: %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
```

The format string uses `%`-style attributes of the log record: `asctime`, `levelname`, `name`, `message`, `filename`, `lineno`, `funcName`, `process`, `thread`. For a judged or tested program, leave `asctime` out — a timestamp is not reproducible — and use a format like `%(levelname)s:%(name)s:%(message)s`.

Lazy arguments

```
log.debug("user %s has %d items", user, len(items))    # right: formatted onl
log.debug(f"user {user} has {len(items)} items")        # wrong: the f-string
```

The message and its arguments are stored and formatted only when a handler actually writes the record. With f-strings the work is done on every call, at every level, and the `%s` form also lets log aggregators group identical messages by template.

Handlers and formatters

```
logger = logging.getLogger("shop")
logger.setLevel(logging.DEBUG)

console = logging.StreamHandler(sys.stdout)            # default is stderr
console.setLevel(logging.INFO)
console.setFormatter(logging.Formatter("%(levelname)s: %(message)s"))

file = logging.FileHandler("shop.log", encoding="utf-8")
file.setLevel(logging.DEBUG)
file.setFormatter(logging.Formatter("%(asctime)s %(name)s %(levelname)s %(mess

logger.addHandler(console)
logger.addHandler(file)
```

A logger can have several handlers, each with its own level and formatter: brief messages to the console, everything to a file. `RotatingFileHandler` and `TimedRotatingFileHandler` cap file sizes; `NullHandler` silences a library by default; `QueueHandler` moves logging off a hot thread. `logging.config.dictConfig` sets all of this from a dict (loaded from TOML or JSON) so that deployment, not code, decides the destinations.

Exceptions

```
try:
    process(record)
except Exception:
    log.exception("failed on record %s", record.id)     # ERROR level, with th
    raise
```

`log.exception` must be called from inside a handler; it is `log.error(..., exc_info=True)`. This is what a boundary handler does instead of `print(e)`: the type, message and full traceback go to the log, and the exception can still be re-raised or recorded.

Libraries versus applications

A library calls `getLogger(__name__)` and logs; it **never** calls `basicConfig`, adds handlers or sets levels — those decisions belong to the application that imports it. The application configures logging once, at start-up in `main`, from its settings. A library that configures

logging hijacks the application's output; the `NullHandler` on its top logger is the polite default so that no "no handlers found" warning appears if the application configures nothing.

When to log what

`DEBUG`: values and steps useful only when diagnosing. `INFO`: milestones an operator wants to see once — started, loaded 42 rows, finished in 3 s. `WARNING`: something unexpected that the program handled. `ERROR`: an operation failed. `CRITICAL`: the program cannot continue. Do not log secrets (tokens, passwords, personal data); do not log in tight loops at `INFO`; do log the identifiers (`record.id`, `order_id`) that let someone find the problem later.

---

print versus logging

`print` writes the program's *result* to stdout. `logging` writes its *narrative* to wherever it is configured, with levels, timestamps, names and a switch. A script that prints "loading..." mixes narrative into results and cannot be piped; converting those prints to `log.info` fixes it in one line each.

Pitfalls

- `basicConfig` in a library.
- f-strings in log calls.
- `print(e)` at a boundary instead of `log.exception`.
- A hand-written logger name instead of `__name__`.
- Timestamps in a format string for output that must be reproducible.
- Logging at `INFO` inside a loop that runs a million times.

Key takeaways

- `logging.getLogger(__name__)` per module; levels `DEBUG` < `INFO` < `WARNING` < `ERROR` < `CRITICAL`; the default threshold is `WARNING`.
- `basicConfig(level=, format=)` once in the application's `main`; never in a library.

Loggers form a tree by name; settings and handlers on a parent apply to children; messages propagate up.

`%s` arguments are formatted lazily; `log.exception` records the traceback; handlers and formatters route output.

`print` is for results, `logging` for the narrative.

EXERCISE 1

Levels and thresholds

Configure logging once with a `StreamHandler(sys.stdout)` and the format `%(levelname)s:%(name)s:%(message)s` (no timestamps). Commands: `level NAME` sets the root logger's level; `log LEVEL logger.name message words` emits a record on the named logger (obtained with `getLogger`) at that level using a `%s` argument for the message. Records below the threshold are dropped by the machinery — you do not filter by hand.

Input: commands (the initial level is `WARNING`). **Output:** the emitted records.

```
log info app.db connected
log warning app.db slow query
level DEBUG
log debug app.web request 1
```

prints

```
WARNING:app.db:slow query
DEBUG:app.web:request 1
```

INPUT 1

```
log info app.db connected
log warning app.db slow query
level DEBUG
log debug app.web request 1
```

OUTPUT 1

```
WARNING:app.db:slow query
DEBUG:app.web:request 1
```

REFERENCE SOLUTION

```
import logging
import sys

handler = logging.StreamHandler(sys.stdout)
handler.setFormatter(logging.Formatter("%(levelname)s:%(name)s:%(message)s"))
root = logging.getLogger()
root.addHandler(handler)
root.setLevel(logging.WARNING)

for line in sys.stdin:
    cmd, *args = line.rstrip("\n").split()
    if cmd == "level":
        root.setLevel(getattr(logging, args[0]))
    elif cmd == "log":
        level, name, *words = args
        logging.getLogger(name).log(getattr(logging, level.upper()), "%s", " ".join(
```

EXERCISE 2

Two handlers, two formats

Attach two handlers to the logger `app` (level `DEBUG`): a console handler on `sys.stdout` at `INFO` with the format `%(levelname)s %(message)s`, and a "file" handler on an `io.StringIO` at `DEBUG` with `%(name)s|%(levelname)s|%(message)s`. Emit records from `log LEVEL message` lines through `app` or its child `app.child` (`log LEVEL child message` — the message is the remaining words). Afterwards print `--- file ---` and the buffer's contents.

Input: lines. **Output:** the console records, then the separator and the buffer.

```
log debug starting
log info loaded
log warning child slow
```

prints

```
INFO loaded
WARNING slow
--- file ---
app|DEBUG|starting
app|INFO|loaded
app.child|WARNING|slow
```

INPUT 1

```
log debug starting
log info loaded
log warning child slow
```

OUTPUT 1

```
INFO loaded
WARNING slow
--- file ---
app|DEBUG|starting
app|INFO|loaded
app.child|WARNING|slow
```

REFERENCE SOLUTION

```
import io
import logging
import sys

app = logging.getLogger("app")
app.setLevel(logging.DEBUG)
buffer = io.StringIO()
console = logging.StreamHandler(sys.stdout)
console.setLevel(logging.INFO)
console.setFormatter(logging.Formatter("%(levelname)s %(message)s"))
filelike = logging.StreamHandler(buffer)
filelike.setLevel(logging.DEBUG)
filelike.setFormatter(logging.Formatter("%(name)s|%(levelname)s|%(message)s"))
app.addHandler(console)
app.addHandler(filelike)

for line in sys.stdin:
    _, level, *words = line.split()
    logger = app
    if words and words[0] == "child":
        logger = logging.getLogger("app.child")
        words = words[1:]
    logger.log(getattr(logging, level.upper()), "%s", " ".join(words))
print("--- file ---")
print(buffer.getvalue(), end="")
```

QUIZ

1. What is the default logging level when nothing is configured?

A. `DEBUG`

B. `INFO`

C. `WARNING`

D. `ERROR`

2. Why is `logging.getLogger(__name__)` the convention?

A. It is faster

B. The logger names then mirror the package hierarchy, so levels and handlers can be set per subtree

C. It is required

D. It avoids imports

3. Why write `log.debug("n=%d", n)` rather than `log.debug(f"n={n}")`?

A. f-strings are not allowed

B. The `%s` form is formatted only if the record is emitted, and aggregators can group by template

C. It is shorter

D. There is no difference

4. Which call logs an exception with its traceback?

A. `log.error(e)`

B. `log.exception("failed")` inside the handler

C. `print(e)`

D. `log.traceback()`

5. What must a library never do with logging?

A. Create a logger

B. Call `basicConfig`, add handlers or set levels — that is the application's job

C. Log at `DEBUG`

D. Use `__name__`

ANSWER KEY · LOGGING — LEVELS, LOGGERS, HANDLERS AND FORMATS

1. C That is why `log.info` prints nothing until `basicConfig(level=logging.INFO)`.

2. B `shop.pricing` is a child of `shop`; settings propagate down and records propagate up.

3. B An f-string does its work even when `DEBUG` is off.

4. B It is `log.error(..., exc_info=True)` and must run while the exception is being handled.

5. B Configuration by a library hijacks the application's output; a `NullHandler` is the polite default.

LESSON 5 · 14 MIN

Writing idiomatic Python — the idioms, the anti-patterns and a refactoring

"Pythonic" is not a vague compliment; it names a specific set of idioms that the language's design makes shorter, clearer and often faster than their alternatives, and a matching set of anti-patterns that read as a translation from another language. This lesson lists both sides — most have appeared earlier in the track, and this is where they are gathered — and then works through one refactoring from a literal translation of Java-style code to the Python it should have been, so that the idioms are seen doing their job together.

The idioms

Instead of	Write
`for i in range(len(xs)): x = xs[i]`	`for x in xs:` / `for i, x in enumerate(xs):`
`i = 0; while i < n: …; i += 1`	`for i in range(n):`
`for i in range(len(a)): a[i], b[i]`	`for x, y in zip(a, b):`
`if len(xs) == 0:` / `if xs == []:`	`if not xs:`
`if x == None:`	`if x is None:`
`if flag == True:`	`if flag:`
`if type(x) == int:`	`if isinstance(x, int):`
`tmp = a; a = b; b = tmp`	`a, b = b, a`
`result = []; for x in xs: result.append(f(x))`	`result = [f(x) for x in xs]`
`if k in d: v = d[k] else: v = default`	`v = d.get(k, default)`
`if k not in d: d[k] = []; d[k].append(x)`	`d.setdefault(k, []).append(x)` or `defaultdict(list)`
`s = ""; for p in parts: s += p`	`s = "".join(parts)`
`"%s is %d" % (a, b)` / `a + " is " + str(b)`	`f"{a} is {b}"`
`f = open(p); …; f.close()`	`with open(p, encoding="utf-8") as f:`
`try: … except: pass`	`except SpecificError:` or `contextlib.suppress`
`def f(xs=[]):`	`def f(xs=None): xs = [] if xs is None else xs`
a class with only `__init__` and getters	a `@dataclass` (or a `NamedTuple`)
`class Stack(list)`	composition: a class holding a list
`lambda x: x[1]` as a key	`operator.itemgetter(1)`

Instead of	Write
a flag variable to signal "found"	`return` from the loop, or `for … else`
`os.path.join(a, b)` string juggling	`Path(a) / b`
checking then acting (`if os.path.exists(p): open(p)`)	`try: open(p) except FileNotFoundError:` (EAFP)
global mutable state	pass values in, return values out; a class if state must persist

Each right-hand side is shorter *and* expresses the intent — "for each element", "if empty", "the value or a default" — where the left-hand side expresses the mechanism.

The anti-patterns that hide bugs

Mutable default arguments (Module 4): shared across calls.

Modifying a list while iterating it (Module 3): skipped elements.

`==` **on floats** (Module 2): use `math.isclose` or integer arithmetic.

Bare `except:` (Module 10): swallows `KeyboardInterrupt` and every bug.

`is` **on values** (`x is 5`): identity is not equality.

`import *`: names of unknown origin, silent shadowing.

String-built SQL or shell commands: injection; use parameters and argument lists.

Printing sets or relying on their order (Module 7): nondeterministic.

`eval` **on input**: code execution; `ast.literal_eval` for literals.

`time.time()` **for timing**: `perf_counter`.

A refactoring

The literal translation:

```
class ReportGenerator:
    def __init__(self):
        self.lines = []

    def generate(self, records):
        totals = {}
        for i in range(len(records)):
            r = records[i]
            if r[1] in totals.keys():
                totals[r[1]] = totals[r[1]] + r[2]
            else:
                totals[r[1]] = r[2]
        keys = list(totals.keys())
        keys.sort()
        for k in keys:
            line = k + ": " + str(totals[k])
            self.lines.append(line)
        output = ""
        for i in range(len(self.lines)):
            output = output + self.lines[i]
            if i != len(self.lines) - 1:
                output = output + "\n"
        return output
```

The same program, idiomatically:

```
from collections import defaultdict

def report(records: list[tuple[str, str, int]]) -> str:
    totals: dict[str, int] = defaultdict(int)
    for _, category, amount in records:
        totals[category] += amount
    return "\n".join(f"{category}: {totals[category]}" for category in sorted(
```

What changed and why: a class with no invariant became a function; index loops became iteration with unpacking, which also names the fields; the `if k in d` dance became `defaultdict`; `keys.sort()` on a copy became `sorted(totals)`; string concatenation became `join` over a generator with an f-string; the `lines` attribute — state that existed only to be concatenated — disappeared; type hints document the shape. Twenty lines became five, and each of the five says one thing. The behaviour is identical; the reader's work is not.

How to get there

Read your own code for the mechanisms — index arithmetic, temporaries, flags, manual accumulation, `if` chains that map keys to values — and ask what the intent is; there is usually a built-in or an idiom that states it. Run `ruff`; many of the table's left-hand sides are rules it flags. Read good code: the standard library's pure-Python modules (`dataclasses.py`, `functools.py`, `textwrap.py`) are the idioms in practice. And when two spellings are equally clear, pick the one the surrounding code already uses.

Pitfalls

- Chasing brevity into unreadability — a nested comprehension with two conditions is not idiomatic, it is compressed.
- Rewriting working code purely for style in the same change as a behaviour change (separate them).
- Idioms applied without their conditions (`x or default` when `0` is a valid `x`).
- A dataclass where the class had real invariants to protect.
- Treating "Pythonic" as a licence to skip hints, docstrings or tests.

Key takeaways

- Iterate, do not index; test truthiness, not length; `is None`; `isinstance`; comprehensions; `get` / `setdefault` / `defaultdict`; `join`; f-strings; `with`; specific `except`; `None` defaults.
- The anti-patterns that hide bugs are mutable defaults, mutating while iterating, float `==`, bare `except`, `is` on values, `import *`, string-built commands, set order, `eval`.
- A class without an invariant is a function or a dataclass; state that exists only to be returned is a local.
- Refactor by naming the intent behind each mechanism and choosing the idiom that states it; let `ruff` find the rest.
- Clarity is the goal; brevity is a side effect.

EXERCISE 1

The refactored report

Implement the lesson's idiomatic `report(records)`: records are `name category amount` lines; total the amounts per category with a `defaultdict`, and return one string of `<category>: <total>` lines in category order joined with newlines (an empty string when there are no records). `main` reads, calls and prints — the function itself does no I/O.

Input: lines. **Output:** the report, or `(no records)`.

```
ada food 500
bob travel 1200
cy food 300
```

prints

```
food: 800
travel: 1200
```

INPUT 1

```
ada food 500
bob travel 1200
cy food 300
```

OUTPUT 1

```
food: 800
travel: 1200
```

REFERENCE SOLUTION

```python
import sys
from collections import defaultdict


def report(records: list[tuple[str, str, int]]) -> str:
    totals: dict[str, int] = defaultdict(int)
    for _, category, amount in records:
        totals[category] += amount
    return "\n".join(f"{category}: {totals[category]}" for category in sorted(totals


def main() -> None:
    records = []
    for line in sys.stdin:
        name, category, amount = line.split()
        records.append((name, category, int(amount)))
    text = report(records)
    print(text if text else "(no records)")


if __name__ == "__main__":
    main()
```

EXERCISE 2

Idiom drill

Read a line of words. Print, each from one idiomatic expression: the distinct words in first-seen order (`dict.fromkeys`); the words grouped by length as `<len>: <words>` lines in increasing length (`defaultdict(list)` , preserving order within a group); the longest word (`max` with `key=len` , first on ties); whether any word is capitalised (`any` over a generator with `str.istitle`); and the words with their positions swapped pairwise using tuple unpacking in a loop over `zip(words[::2], words[1::2])` , with an odd last word kept.

Input: one line. **Output:** `distinct …` , the group lines, `longest <w>` , `capitalised <bool>` , `swapped …` .

```
to be or not to Be
```

prints

```
distinct to be or not Be
2: to be or to Be
3: not
longest not
capitalised True
swapped be to not or Be to
```

INPUT 1

```
to be or not to Be
```

OUTPUT 1

```
distinct to be or not Be
2: to be or to Be
3: not
longest not
capitalised True
swapped be to not or Be to
```

REFERENCE SOLUTION

```python
from collections import defaultdict

words = input().split()
print("distinct", " ".join(dict.fromkeys(words)))
by_len: dict[int, list[str]] = defaultdict(list)
for w in words:
    by_len[len(w)].append(w)
for n in sorted(by_len):
    print(f"{n}: {' '.join(by_len[n])}")
print("longest", max(words, key=len))
print("capitalised", any(w.istitle() for w in words))
swapped = []
for a, b in zip(words[::2], words[1::2]):
    swapped += [b, a]
if len(words) % 2:
    swapped.append(words[-1])
print("swapped", " ".join(swapped))
```

QUIZ

1. Which rewrite is idiomatic?

A. `for i in range(len(xs)): print(i, xs[i])`

B. `for i, x in enumerate(xs): print(i, x)`

C. `i = 0; for x in xs: print(i, x); i += 1`

D. `for i in xs.keys(): …`

2. Which of these hides a bug rather than merely looking foreign?

A. `if len(xs) == 0:`

B. `def f(xs=[]):`

C. `s = s + t`

D. `for i in range(len(xs)):`

3. When should a class become a plain function?

A. When it has more than three methods

B. When it holds no invariant and its only state exists to be returned

C. Never

D. When it is slow

4. What does `x or default` get wrong?

A. Nothing

B. It replaces a legitimate falsy `x` — `0`, `""`, `[]` — with the default

C. It is slower than `if`

D. It raises on `None`

5. What is the goal of idiomatic code?

A. Fewest lines

B. Clarity — brevity is a side effect

C. Fastest execution

D. Most features used

ANSWER KEY · WRITING IDIOMATIC PYTHON — THE IDIOMS, THE ANTI-PATTERNS AND A REFACTORING

1. B `enumerate` states the intent — element with index — without index arithmetic.

2. B A mutable default is shared across calls; the others are just verbose.

3. B A class without an invariant is a namespace for a function, or a dataclass for a record.

4. B Use `x if x is not None else default` when falsy values are meaningful.

5. B A compressed comprehension nobody can read is not idiomatic.

CHECKPOINT · 22 MIN

Checkpoint — Type hints and code quality

This checkpoint covers the whole module: the hint vocabulary — built-in generics, unions, `Callable`, `TypeVar`, `Generic`, `ClassVar`, `Final`, `Literal`, `TypedDict`, `NewType` — with parameters hinted wide and returns narrow; what a type checker verifies and how narrowing, `cast`, `TYPE_CHECKING` and `get_type_hints` work; PEP 8 layout, naming, docstrings and the linter/formatter workflow; `logging` with levels, the logger hierarchy, formats and handlers; and the idioms and anti-patterns that make code Pythonic.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Which hint accepts a list, a tuple and a generator of ints, and which requires indexing?
- What does `Optional[int]` mean, and what does it not mean?
- How does an `is None` test change what the checker allows afterwards?
- Which naming style applies to functions, classes and constants?
- Why must a library never call `logging.basicConfig`?
- Why write `log.debug("%s", x)` rather than an f-string?
- Which five anti-patterns hide bugs rather than just looking foreign?

The three programs are a run-time validator driven by `get_type_hints` that handles unions and `Literal`s, a linter-lite that flags PEP 8 naming and idiom violations line by line, and a logging pipeline whose records are routed by level to two in-memory handlers with different formats.

EXERCISE 1

A validator that reads the hints

Given hinted functions `greet(name: str, times: int = 1)`, `scale(x: float, by: int | None = None)`, `open_mode(mode: Literal["r", "w"])` and `total(xs: list[int])`, write `check(f, *args)` that reads `typing.get_type_hints(f)` and validates each positional argument: a plain class with `isinstance` (an `int` satisfies `float`; a `bool` never satisfies `int` or `float`); a union (`get_origin` is `types.UnionType`) if any member accepts; `Literal` if the value is one of `get_args`;

`list[T]` if it is a list whose every element satisfies `T`. Missing arguments with defaults are fine. Each input line is a name followed by comma-separated argument literals; print `ok <result>` or `type error: <param>`.

Input: lines. **Output:** one line per call.

```
greet "ada", 2
scale 3, None
open_mode "x"
total [1, 2, "3"]
```

prints

```
ok Hello, ada! Hello, ada!
ok 3
type error: mode
type error: xs
```

INPUT 1

```
greet "ada", 2
scale 3, None
open_mode "x"
total [1, 2, "3"]
```

OUTPUT 1

```
ok Hello, ada! Hello, ada!
ok 3
type error: mode
type error: xs
```

REFERENCE SOLUTION

```
import ast
import sys
import types
import typing
from typing import Literal


def greet(name: str, times: int = 1) -> str:
    return " ".join([f"Hello, {name}!"] * times)


def scale(x: float, by: int | None = None) -> float:
    return x * (by if by is not None else 1)


def open_mode(mode: Literal["r", "w"]) -> str:
    return f"mode {mode}"


def total(xs: list[int]) -> int:
    return sum(xs)


FUNCTIONS = {"greet": greet, "scale": scale, "open_mode": open_mode, "total": total}


def accepts(value, hint) -> bool:
    origin = typing.get_origin(hint)
    if origin is types.UnionType or origin is typing.Union:
        return any(accepts(value, member) for member in typing.get_args(hint))
    if origin is Literal:
        return value in typing.get_args(hint)
    if origin is list:
        (inner,) = typing.get_args(hint)
        return isinstance(value, list) and all(accepts(v, inner) for v in value)
    if hint is type(None):
        return value is None
    if isinstance(value, bool):
        return hint is bool
    if hint is float:
        return isinstance(value, (int, float))
    return isinstance(value, hint)


def check(f, *args):
    hints = typing.get_type_hints(f)
    params = [p for p in hints if p != "return"]
    for param, value in zip(params, args):
        if not accepts(value, hints[param]):
```

```
            return f"type error: {param}"
    return f"ok {f(*args)}"


for line in sys.stdin:
    name, *rest = line.strip().split(maxsplit=1)
    args = list(ast.literal_eval(f"({rest[0]},)")) if rest else []
    print(check(FUNCTIONS[name], *args))
```

EXERCISE 2

Linter-lite

Read a Python source and report, per line, every rule it breaks (comma-separated, in rule order): `tab`, `long` (over 79 characters), `trailing-space`, `def-name` (not `snake_case`), `class-name` (not `PascalCase`), `none-compare` (`== None` / `≠ None`), `bare-except` (`except:`), `mutable-default` (`=[]`, `={}` or `=set()` inside a `def` line's parameters), `star-import` (`import *`). Print `line <n>: <rules>` for each offending line, then `problems <total rules>`.

Input: source lines. **Output:** the report and the count.

```
from os import *
def Load(xs=[]):
    try:
        pass
    except:
        pass
```

prints

```
line 1: star-import
line 2: def-name, mutable-default
line 5: bare-except
problems 4
```

INPUT 1

```
from os import *
def Load(xs=[]):
    try:
        pass
    except:
        pass
```

OUTPUT 1

```
line 1: star-import
line 2: def-name, mutable-default
line 5: bare-except
problems 4
```

REFERENCE SOLUTION

```
import re
import sys

RULES = [
    ("tab", lambda s: "\t" in s[: len(s) - len(s.lstrip())]),
    ("long", lambda s: len(s) > 79),
    ("trailing-space", lambda s: s ≠ s.rstrip(" ")),
    ("def-name", lambda s: bool((m := re.match(r"\s*def\s+(\w+)", s)) and not re.ful
    ("class-name", lambda s: bool((m := re.match(r"\s*class\s+(\w+)", s)) and not re
    ("none-compare", lambda s: re.search(r"[=!]=\s*None\b", s) is not None),
    ("bare-except", lambda s: re.match(r"\s*except\s*:", s) is not None),
    ("mutable-default", lambda s: re.match(r"\s*def\s", s) is not None and re.search
    ("star-import", lambda s: re.search(r"\bimport\s+\*", s) is not None),
]

problems = 0
for n, line in enumerate(sys.stdin, start=1):
    line = line.rstrip("\n")
    broken = [name for name, test in RULES if test(line)]
    if broken:
        problems += len(broken)
        print(f"line {n}: {', '.join(broken)}")
print(f"problems {problems}")
```

EXERCISE 3

Routed logging

Build a logger `svc` at `DEBUG` with two `StringIO` handlers: `errors` at `ERROR` with format `%(levelname)s %(name)s: %(message)s`, and `all` at `DEBUG` with `%(levelname)s: %(message)s`. Process lines `LEVEL message…` through `svc`, with the special line `fail message…` raising and catching a `RuntimeError` and logging it with `log.exception` (which appends the traceback; count its records but print only their first line). Afterwards print `--- errors ---`, the first line of each record in the errors buffer, `--- all ---`, the first line of each record in the all buffer, then `records <n>` for the all buffer.

Input: lines. **Output:** as described.

```
info started
debug detail
fail disk full
error plain error
```

prints

```
--- errors ---
ERROR svc: disk full
ERROR svc: plain error
--- all ---
INFO: started
DEBUG: detail
ERROR: disk full
ERROR: plain error
records 4
```

INPUT 1

```
info started
debug detail
fail disk full
error plain error
```

OUTPUT 1

```
--- errors ---
ERROR svc: disk full
ERROR svc: plain error
--- all ---
INFO: started
DEBUG: detail
ERROR: disk full
ERROR: plain error
records 4
```

REFERENCE SOLUTION

```
import io
import logging
import sys

log = logging.getLogger("svc")
log.setLevel(logging.DEBUG)
errors, everything = io.StringIO(), io.StringIO()
h_err = logging.StreamHandler(errors)
h_err.setLevel(logging.ERROR)
h_err.setFormatter(logging.Formatter("%(levelname)s %(name)s: %(message)s"))
h_all = logging.StreamHandler(everything)
h_all.setLevel(logging.DEBUG)
h_all.setFormatter(logging.Formatter("%(levelname)s: %(message)s"))
log.addHandler(h_err)
log.addHandler(h_all)

for line in sys.stdin:
    level, *words = line.split()
    message = " ".join(words)
    if level == "fail":
        try:
            raise RuntimeError(message)
        except RuntimeError:
            log.exception("%s", message)
    else:
        log.log(getattr(logging, level.upper()), "%s", message)


def first_lines(text):
    # a record with a traceback spans several lines; keep the first line of each rec
    lines = text.splitlines()
    return [l for l in lines if l and not l.startswith(("Traceback", "  ", "RuntimeE


print("--- errors ---")
print("\n".join(first_lines(errors.getvalue())))
print("--- all ---")
kept = first_lines(everything.getvalue())
print("\n".join(kept))
print(f"records {len(kept)}")
```

CHECKPOINT QUESTIONS

1. Which hint is right for a function returning a generator of strings?

A. `list[str]`

B. `Iterator[str]`

C. `Generator`

D. `str`

2. What does `TypedDict` describe?

A. A dict with typed keys and values of one type

B. A dict with a fixed set of string keys, each with its own value type

C. A dataclass

D. An ordered dict

3. What does `NewType("UserId", int)` give you?

A. A subclass with methods

B. A distinct type for the checker that is a plain int at run time

C. A validated integer

D. An enum

4. Which test does *not* narrow a union for the checker?

A. `isinstance(x, int)`

B. `x is None`

C. A custom `is_int(x)` returning `bool`

D. `assert isinstance(x, str)`

5. What does `# type: ignore[arg-type]` do?

A. Disables the checker for the file

B. Suppresses that one error code on that line

C. Fixes the type

D. Marks a TODO

6. Which tool rewrites layout to a canonical style?

A. `mypy`

B. `black` / `ruff format`

C. `pytest`

D. `pip`

7. Which is PEP 8-compliant?

A. `def f( a,b ):`

B. `def f(a, b):`

C. `def f (a, b) :`

D. `def f(a , b):`

8. What does a `.setLevel(INFO)` on a *handler* do that the logger's level does not?

A. Nothing different

B. Filters what reaches that one destination; the logger's level filters before any handler

C. Changes the format

D. Applies to child loggers

9. Where does a child logger's record go if the child has no handlers?

A. Nowhere

B. Up to its ancestors' handlers, keeping the child's name

C. To stdout

D. It raises

10. Which line is the idiomatic swap?

A. `tmp = a; a = b; b = tmp`

B. `a, b = b, a`

C. `a = b; b = a`

D. `swap(a, b)`

11. Which spelling of 'value or a default when the key is missing' is idiomatic?

A. `d[k] if k in d else default`

B. `d.get(k, default)`

C. `try: d[k] except: default`

D. `d.setdefault(k)`

12. What is the difference between a docstring and a comment?

A. None

B. A docstring is the first statement, stored in `__doc__`, describing the contract; a comment is discarded and explains why

C. Comments are stored; docstrings are not

D. Docstrings use `#`

ANSWER KEY · CHECKPOINT — TYPE HINTS AND CODE QUALITY

1. B A generator function's return type is `Iterator[T]` (or `Generator[T, None, None]` when send/return types matter).

2. B It types JSON-shaped data without converting it to a class.

3. B It prevents ids of different kinds being mixed, at zero run-time cost.

4. C A custom predicate narrows only if declared with a `TypeGuard` return type.

5. B Always include the code; a bare ignore hides every future error on the line.

6. B `ruff check` lints; `mypy` types; formatting is the formatter's job.

7. B No space inside brackets or before the call's parentheses; a space after each comma.

8. B Logger level first, then each handler's level — two independent gates.

9. B Propagation is what makes one root configuration serve a whole application.

10. B Tuple packing evaluates the right side before binding.

11. B `get` does one lookup and states the intent; bare `except` is a bug in the third.

12. B `help()` shows docstrings; nothing shows comments except the source.
