10

Writing Pythonic Code

Idioms, testing, logging & a review checklist

Chapter 10 · telling good Python from bad at a glance

10 The journey — five habits of good Python

Reading Python you can already do. This chapter trains the eye that tells idiomatic, safe, tested code from the un-Pythonic kind — the eye a reviewer needs. Five stops, one at a time.

You are not asking 'does it run?' — you are asking 'is it idiomatic, safe, tested and maintainable?' The chapter ends with a checklist you reuse on every pull request.

10

Just try it — the core idiom

EAFP, not check-first (LBYL)

Python ships a manifesto (run 'import this' for the Zen of Python). Its biggest habit: just TRY something and catch the failure — 'Easier to Ask Forgiveness than Permission' (EAFP) — instead of checking first ('Look Before You Leap', LBYL).

```
# LBYL — the JS habit: look before you leap
if 'temperature' in config and config['temperature'] is not None:
    temp = config['temperature']
else:
    temp = 0.7

# EAFP — the Pythonic way: just try it, or ask for a default
temp = config.get('temperature', 0.7)   # => 0.7 if absent
```

JS to Python: don't read try/except as exceptional-and-scary the way you might in JS. Here it's the ordinary control-flow tool — faster (no double lookup) and it avoids race conditions.

10

Truthiness & looping over values

Two idioms on every page of Python

```
# BEFORE — length check
if len(messages) > 0:
    ...

# AFTER — empty list is falsy
if messages:
    ...
if not messages:      # nothing to do
    return []
```

```
# index AND value together
for i, prompt in enumerate(prompts):
    print(i, prompt)

# walk two lists in lockstep
for p, m in zip(prompts, models):
    print(f'{p} -> {m}')
```

Pythonic loops hand you the VALUES, not index numbers. Reach for enumerate when you also need the position, and zip to walk two sequences together. Seeing 'range(len(...))' is a review flag.

10 Comprehensions & unpacking

Build a new list in one readable line

```
tokens = [{'text': 'hi', 'logprob': -0.2}, {'text': '!', 'logprob': -3.1}]

# BEFORE — manual accumulation
confident = []
for t in tokens:
    if t['logprob'] > -1.0:
        confident.append(t['text'])

# AFTER — a comprehension (one line)
confident = [t['text'] for t in tokens if t['logprob'] > -1.0]
```

```
role, content = ('user', 'Explain RAG')   # tuple unpack
first, *rest = [1, 2, 3, 4]   # first=1, rest=[2,3,4]
merged = {**defaults, **overrides}   # dict merge
```

A comprehension that grows past one readable line, or has side effects, is itself a review flag — use a plain loop there. Unpacking pulls several values out in one line; you'll read it everywhere.

10

Four more idioms to recognise

with · f-strings · pathlib · is None

```
# 'with' guarantees cleanup, even on error
with open('prompt.txt', encoding='utf-8') as
f:
    data = f.read()

# f-strings, not concatenation
label = f'{name} x{n}'

# pathlib, not string paths
cfg = Path.home() / '.config' / 'app.json'

# identity check, not ==
if result is None:
    ...
```

Read these as the 'good' form:

with — auto-closes the file / lock / connection (a try/finally you never write)

f-strings — not name + ' x' + str(n)

pathlib — the '/' joins path parts safely across operating systems

is None — never == None

Context over comments: a well-named function beats a clever comment. A comment that restates the code ('# increment i') is noise; one that explains WHY (an API quirk, a workaround) is gold.

10

Style & tooling — the debates are over

PEP 8 is the baseline; ruff does the work

```
snake_case     ->  variables & functions      max_tokens,  build_prompt()
UPPER_CASE     ->  constants                  DEFAULT_MODEL = 'sonnet'
PascalCase     ->  classes                    class ChatSession:
_leading       ->  'private' names            _client,  _retry()
4 spaces       ->  indent (never tabs)        line length 88 (ruff default)
```

```
ruff format .          # auto-formats (Black-compatible)
ruff check . --fix     # lints + fixes imports, unused vars
mypy src/              # type-check separately (or pyright)
```

JS to Python: ESLint + Prettier collapse into ONE tool, ruff (~150x faster). In review, never leave a style nit. If it's style, the answer is 'run ruff.'

10

Immutability — two footguns to spot

The most famous trap in Python

```
# BUG: the default list is made ONCE,
# at definition, and shared across calls
def add_message(msg, history=[]):
    history.append(msg)
    return history

add_message('a')   # => ['a']
add_message('b')   # => ['a', 'b']  leaked!
```

```
@dataclass(frozen=True)
class LLMConfig:
    model: str
    temperature: float = 0.7

cfg = LLMConfig('sonnet')
# cfg.temperature = 0.9   -> raises error
new = replace(cfg, temperature=0.9)
```

The fix (sentinel pattern): default history=None, then history = [] if history is None else history, and return a NEW list [*history, msg]. Review lens: does this function mutate its arguments, or global / shared state? If two tasks touch the same list without a lock, flag it.

10

Testing with pytest

No boilerplate — files test_*.py, plain assert

```
# test_prompt.py — discovered automatically
def test_build_prompt_includes_system():
    result = build_prompt('Hello')
    assert 'You are a helpful assistant' in result   # plain assert

# assert that something raises
def test_empty_text_rejected():
    with pytest.raises(ValueError, match='empty'):
        summarize('')
```

```
@pytest.fixture
def sample_history():
    return [{'role': 'user', 'content': 'hi'}]

def test_append(sample_history):    # name =
fixture
    out = add_message('bye', sample_history)
    assert len(out) == 2
```

```
Jest to pytest:

describe/it          ->  bare  test_*  functions
expect(x).toBe(y)    ->  assert x == y
beforeEach           ->  @pytest.fixture
test.each            ->
@pytest.mark.parametrize
```

10

Mocking LLM calls — the key GenAI skill

Never hit the real API in a unit test

```
from unittest.mock import MagicMock

def test_extract_json_answer_parses_tool_call():
    fake = MagicMock()           # a stand-in client, no network
    json_reply = '{ action: search }'   # a canned response string
    fake.messages.create.return_value = MagicMock(
        content=[MagicMock(text=json_reply)]
    )
    result = extract_json_answer(fake, 'find docs')
    assert result == {'action': 'search'}
    fake.messages.create.assert_called_once()   # it WAS called
```

Why mock? The real API is slow, costs money, and is non-deterministic. Mock the CLIENT and test YOUR logic — prompt building, response parsing, validation, tool dispatch — not the model's answer quality.

10

Logging done right

print() in service code is a smell

```
import logging
logger = logging.getLogger(__name__)   # one per module

def call_model(prompt):
    logger.info('calling model', extra={'prompt_len': len(prompt)})
    try:
        return _client.complete(prompt)
    except TimeoutError:
        logger.warning('timed out, retrying')
        raise
    except Exception:
        logger.exception('call failed')   # logs full traceback
```

Levels, low to high:
DEBUG < INFO < WARNING < ERROR < CRITICAL

Configure once at the app's entry point with logging.basicConfig(...). print() has no severity, no timestamp, no module origin, and can't be filtered.

Never log secrets or PII — no API keys, no auth tokens, no full prompts containing user data. Logging a whole request body with a customer's message is a compliance incident. A hard review flag.

10

Config & secrets

API keys come from the environment, never source

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    anthropic_api_key: str          # required: startup FAILS if unset
    model: str = 'sonnet'
    max_tokens: int = 1024          # coerced & validated from env

    model_config = SettingsConfigDict(env_file='.env')

settings = Settings()    # reads ANTHROPIC_API_KEY, MODEL, ...
```

'.env' is git-ignored; you commit a '.env.example' with blank values. pydantic-settings adds VALIDATION and type coercion on top of loading — bad config fails at boot with a clear error, not at 3am in production.

Any `api_key = 'sk-...'` literal in a diff is an automatic BLOCK-THE-PR. Secrets are read from env / pydantic-settings, never committed, and kept out of logs and error messages.

10

The GenAI code-review checklist

What a senior engineer actually checks

Correctness
- Logic matches the PR's intent, not just 'runs'
- No mutable default args (=[], ={})
- enumerate / zip, not range(len(...))
- Edge cases: empty, None, '', 0

Types & validation
- Public signatures type-annotated
- LLM / JSON output validated before use
- External input checked at the boundary
- mypy / pyright passes

Errors & resources
- No bare 'except:' swallowing errors
- Specific exceptions (except KeyError)
- 'with' for every file / client / lock
- Errors logged with context, then re-raised

Security / secrets
- No hardcoded key / token -> block
- Secrets from env, never committed
- Secrets & PII kept out of logs
- No eval / exec / pickle on untrusted input

Concurrency & cost
- No un-awaited coroutine
- No blocking I/O inside async
- API calls: timeout + retry-with-backoff
- Watch O(n^2): 'in' on a list in a loop

Testing
- Tests exist and cover failure paths
- The LLM / API client is mocked
- Tests check YOUR parsing, not model quality
- No real network, no real secrets, no flakiness

10 Top 10 red flags — the 10-second smell test

1. Hardcoded API key / secret in the diff -> block immediately
2. Bare 'except:' or 'except: pass' swallowing errors
3. Mutable default argument (=[], ={})
4. LLM / JSON output used with no validation or try
5. print() for logging in service or library code
6. No timeout / no retry-with-backoff on API calls
7. Un-awaited coroutine, or blocking I/O inside async
8. Shared mutable state across tasks / threads, no lock
9. Real API hit in a unit test (slow, costly) instead of a mock
10. Secrets or PII written to logs

Read a PR against these and you are doing exactly what a senior Python engineer does — judging not whether the code works TODAY, but whether it stays correct, safe and testable TOMORROW.

The teaching — 'Does it run?' is the junior question. 'Is it idiomatic, safe, tested and maintainable?' is yours. That trained eye is what turns you into a reviewer of the code the whole AI field is written in.