BRP DEVELOPERS / FREE LEARNING LIBRARY

Python Code Review & Testing Workbook

Practice edition ? BRP developers free companion guide

Python practice | All levels

Original educational companion guide. No affiliation with any interview platform or certification provider.

Interview preparation is not only about producing an algorithm. It is also about reading unfamiliar code, finding behavioral risks, and proposing a testable improvement. This workbook uses small original examples to practise that review process.

Review behavior before style

Read the function contract and trace values through representative inputs. Look first for correctness, security, data loss, resource leaks, and unbounded work. Then consider readability and performance. A useful review comment identifies the scenario, the consequence, and a small remedy.

Weak comment: ?This is bad style.?

Useful comment: ?When `items` is omitted, this default list is shared across calls, so the second request can include values from the first. Could we use `None` and create the list inside the function??

Find the hidden shared state

```
def tags_for(name, tags=[]):
    tags.append(name.lower())
    return tags
```

The default list is created once. Calls without `tags` share it. A focused regression test proves the behavior:

```
def test_default_state_is_per_call():
    assert tags_for("Ada") == ["ada"]
    assert tags_for("Lin") == ["lin"]
```

In a real project, reset or isolate state so tests do not depend on execution order. The repair is a `None` sentinel or an immutable default, depending on the function contract.

Test the boundary, not the implementation

Suppose a parser accepts a string containing comma-separated integers. Useful tests include an empty string, whitespace, a negative number, a malformed token, and a very long input if limits matter. Avoid asserting that the parser calls a particular private helper unless that interaction is itself part of the contract.

A table of examples makes boundary behavior explicit:

```
import pytest

@pytest.mark.parametrize(
    ("raw", "expected"),
    [("1,2", [1, 2]), (" -3, 4 ", [-3, 4]), ("", [])],
)
def test_parse_integers(raw, expected):
    assert parse_integers(raw) == expected
```

Also test invalid input and specify whether the public API raises `ValueError`, returns a result type, or reports an error another way. Consistency matters more than choosing one universal convention.

Exceptions and cleanup

Review whether exceptions are caught at the right boundary. A broad catch that logs and continues can turn a clear failure into corrupted state. A `finally` block or context manager should release resources. If an exception is translated, preserve useful causality:

```
try:
```

```
    amount = int(raw_amount)
except ValueError as error:
    raise ValueError("amount must be an integer") from error
```

Do not expose stack traces or secret values to end users. Logs should help operators distinguish invalid input, dependency failure, and unexpected defects without recording credentials.

Security review checklist

For an HTTP handler, check authentication and authorization separately: a signed-in user is not automatically allowed to access every record. Validate and bound request bodies. Use parameterized SQL rather than string interpolation. Avoid logging passwords, session tokens, payment secrets, or unnecessary personal data. Rate-limit sensitive operations and use safe cookie attributes over HTTPS.

For uploaded files, validate size and type on the server, generate storage names, and avoid serving user-controlled HTML as trusted origin content. For outbound requests, consider whether user input can turn the server into a request proxy to internal services.

Performance review with evidence

Look for repeated work inside loops, accidental quadratic operations, and loading an unbounded dataset into memory. Then measure with representative data. An O(n log n) algorithm may be preferable to a complicated O(n) approach when inputs are modest and clarity reduces risk. A microbenchmark should vary data size and confirm identical output.

A useful review includes an estimate before measurement and a measured result after a change. Do not optimize based only on how short an expression looks.

Refactor in observable steps

When changing a legacy function, add a test for existing behavior first. Separate a pure calculation from I/O where that makes the rule easier to test. Preserve public input and output behavior unless changing it is intentional and documented. Small commits make regressions easier to locate.

For a suspicious branch, create the smallest failing case. For a concurrency bug, coordinate tasks with events or barriers instead of adding arbitrary sleeps. For a time-dependent function, inject a clock or freeze time in a test.

Review exercises and answer notes

Exercise 1: A function returns `dict.fromkeys(names, [])`. What is the issue? Every key receives the same list object. Use a comprehension to create distinct lists: `{name: [] for name in names}`.

Exercise 2: A test passes alone but fails in the suite. What do you inspect first? Shared mutable state, global patches not restored, environment variables, order-dependent database rows, and fixed external resources.

Exercise 3: An endpoint checks that the user is logged in but accepts any account ID in the URL. What is missing? Authorization: verify that the authenticated user may access the requested account's resource.

Exercise 4: A ?faster? change reduces runtime on one tiny input but increases memory substantially. What evidence is needed? Representative workload sizes, time and peak memory measurements, and a statement of the trade-off against expected limits.

A high-quality review is precise, respectful, and reproducible. State what input triggers the issue, why it matters, and how a test can prevent its return.

More free guides: brpdevs.online/learn/