Python Representation Invariants

Table of contents

Documenting invariants 1
Establishing vs. checking 2
Enforcing invariants 2
Encapsulation: don't expose mutable internals 3
Best practices 4

A **representation invariant** is a condition or property that must remain true about an object's internal state throughout its lifetime. They're primarily a *development-time* tool: they help catch bugs early, simplify method implementations (since you can assume a valid starting state), and document the assumptions a class relies on.

See 2025-09-16-representation-invariants[1] for the lecture note.

Documenting invariants

State invariants in the class docstring, underneath its attributes:

```
class Fraction():
    """Represents a mathematical fraction.

    Representation Invariants:
    - denominator != 0
    - if fraction is zero, it is represented as 0/1
    - if fraction is negative, numerator is negative
    """
```

A good invariant is a *checkable statement about the stored attributes* — something a method could evaluate and get `True` or `False` from. "The denominator is not zero" is an invariant; "this class models fractions" is not.

[1] courses/csse1001/2025-09-16-representation-invariants.pdf

Establishing vs. checking

There are two ways to make an invariant hold, and they are easy to confuse:

- **Establish** it — normalise the input so the property is true by construction, e.g. `if denom < 0: numer, denom = -numer, -denom`.
- **Check** it — assert the property and fail if the caller got it wrong, e.g. `assert denom > 0`.

`Fraction` uses both, on different invariants. Centralising all the checks in `_check_invariants()` pushes you toward the second style: the method can only report, not repair, so anything you want normalised has to be normalised before it is called.

Enforcing invariants

Assertions

```
assert <condition that should be true>, "message if it's not"
```

- Raises `AssertionError` if the condition is false.
- Disabled when Python is run with optimization (`python -O`) — so asserts should catch *bugs*, not perform real input validation that must always run (use `raise ValueError(...)` etc. for that).

`assert` for bugs, `raise` for callers

The distinction matters enough that `DroneFlight` uses both in one class:

```
def _check_invariants(self) -> None:
    assert 0.0 <= self._altitude <= self._MAX_ALT, "Altitude out of range."

def set_altitude(self, value: float) -> None:
    if not isinstance(value, (int, float)):
        raise TypeError("Altitude must be a number.")
    if not 0.0 <= value <= self._MAX_ALT:
        raise ValueError(f"Altitude must be 0-{self._MAX_ALT} m.")
    self._altitude = float(value)
    self._check_invariants()
```

`_check_invariants()` asks "is *my own code* still correct?" — a false answer is a bug, and it is fine for that check to disappear under `-O`. `set_altitude()` asks "did *the caller* hand me something usable?" — a false answer is expected behaviour, and it must keep raising in production. See python-exceptions[2] for the exception types.

The `_check_invariants()` helper pattern

Centralise all invariant checks in one private method, and call it:

- At the end of `__init__`.
- At the end of any method that mutates state.
- At the start of any method that relies on the invariants already holding.

```
def _check_invariants(self) -> None:
    assert self._denom != 0, "Denominator cannot be zero."
    ...
```

Two habits that keep the number of call sites small:

- **Route every mutation through one setter.** If `ascend()` and `land()` both call `set_altitude()` instead of assigning `self._altitude` directly, the range check lives in exactly one place.
- **Put the limits in class constants.** `_ID_LEN = 6` and `_MAX_ALT = 120.0` on the class body give the rule one home, so the assertion and the error message change together when the limit does. Only the code that actually reads the constant is protected, though: `DroneFlight` still hard-codes `0 _altitude 120` in its docstring, and interpolating the constant into the message turns it into `0-120.0 m`, so the wording drifts from the docstring anyway. See 2025-09-16-representation-invariants[3] for that mismatch.

Encapsulation: don't expose mutable internals

Returning a direct reference to a private mutable attribute (like a list or dict) lets external code silently violate the class's invariants:

```
def get_members(self):        # BAD -- exposes the real list
    return self._members

def get_members(self):        # GOOD -- caller gets an independent copy
    return list(self._members)
```

[2] courses/csse1001/python-exceptions.pdf
[3] courses/csse1001/2025-09-16-representation-invariants.pdf

This is the aliasing behaviour of python-lists[4] showing up inside a class: the getter hands out a second name for the *same* list object, so the caller's `append` mutates the object's state without any of its methods running — and therefore without any assertion getting a chance to fire.

Best practices

- Document invariants in docstrings.
- Centralise invariant checking in a single method.
- Don't expose methods/attributes that could let external code violate invariants.
- Write test cases that target edge cases (see python-testing[5]).
- Type hints can express some invariants, but not all (e.g. numeric ranges still need explicit checks).
- Balance strictness with practicality — not every property needs an assertion.

[4] courses/csse1001/python-lists.pdf
[5] courses/csse1001/python-testing.pdf