Representation Invariants

Table of contents

Outline 1
What are representation invariants? 1
Writing representation invariants 2
Enforcing representation invariants 3
Best practices 6
Exercise: `DroneFlight` 6
Summary 11

Lecture 8B. Course logistics are in csse1001[1]. The full reference on writing and enforcing invariants is python-representation-invariants[2]. Diagrams below follow csse1001-diagram-notation[3].

Outline

- What representation invariants are, and why they matter
- Documenting invariants in docstrings
- Enforcing invariants with `assert` and a `_check_invariants()` helper
- Encapsulation and avoiding exposed mutable state
- Worked example: `DroneFlight`

What are representation invariants?

Representation invariants are conditions or properties that must remain true about the internal state of an object throughout its lifetime (usually enforced during development).

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-representation-invariants.pdf
[3] courses/csse1001/csse1001-diagram-notation.pdf

Why do they matter?

- Discover and prevent bugs and inconsistent behaviour.
- Simplify the implementation of methods (you can assume a valid starting state).
- Make code more maintainable and robust.
- Help detect errors early.
- Serve as documentation for developers.

Examples

- A stack's size must never be negative.
- A list's length must equal the number of elements it contains.
- All elements in a set must be unique.
- A fraction's denominator cannot be zero.
- A date must follow calendar rules (e.g. no 30 Feb).
- For free/basic X (Twitter) accounts, the standard limit is 280 characters per tweet.

Writing representation invariants

Clearly state invariants in **docstrings**, underneath a class's 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
    """
```

Each of these three is a *checkable* statement about the stored attributes, not a description of what the class is for. The second and third are **normalisation** rules — they pick one representative out of several possible encodings of the same value (so `0/5` and `1/-2` are not valid internal states even though they name valid numbers).

Enforcing representation invariants

Assertions

Directly check conditions in the initialiser and setter methods, using an `assert` statement to raise an `AssertionError` if an assumption isn't met:

```
assert <statement that should be true>, "Error message if not True"
```

- Assertions can be disabled in production when running Python with optimization (`python -O`).
- Assertions are meant to find bugs (e.g. invalid states) *during development*.

```
class Fraction():
    def __init__(self, numer: int, denom: int) -> None:
        assert denom != 0, "Denominator cannot be zero."
        assert isinstance(numer, int), "Numerator must be an integer."
        assert isinstance(denom, int), "Denominator must be an integer."

        # Ensure denominator is positive
        if denom < 0:
            numer, denom = -numer, -denom
        self._numer = numer
        self._denom = denom

        # Special case for zero
        if self._numer == 0:
            assert self._denom == 1, "Zero must be 0/1"
```

Two things about this listing:

- The sign rule is **established**, not checked: `if denom < 0: numer, denom = -numer, -denom` rewrites the arguments so the invariant "if the fraction is negative, the numerator is negative" holds by construction.
- The zero rule is **checked**, not established: `Fraction(0, 5)` passes every earlier assertion, then trips `assert self._denom == 1`. So a perfectly reasonable-looking call is rejected rather than normalised to `0/1`. The two invariants in the same initialiser are enforced by opposite strategies.

Private attributes and encapsulation

- Use private (`_variable`) names to discourage direct access.
- Provide public methods that maintain the invariants.
- **Never expose mutable objects directly**:

```
class Team():
    def __init__(self):
        self._members = []   # internal mutable state

    # classic getter (dangerous!)
    # BAD!
    def get_members(self):
        return self._members

    # instead, return a copy
    # GOOD!
    # def get_members(self):
    #     return list(self._members)  # caller can't modify internal state directly

t = Team()
members = t.get_members()   # gets the actual list inside the class
members.append("Alice")      # modifies it directly!

print(t.get_members())        # ['Alice'] <-- internal state was changed externally
```

The reason this defeats every assertion you could write is that no method of `Team` is running when the change happens: `get_members` has already returned, and the caller then mutates the object `Team` is still holding, from outside (Figure 1). This is the aliasing problem from 2025-08-18-non-primitive-data[4] arriving inside a class.

Helper methods

Define a private method (e.g. `_check_invariants()`) that validates the object's state, and call it after any state changes:

state change → `_check_invariants` → valid

When to call `_check_invariants()`:

- At the end of `__init__`.

[4] courses/csse1001/2025-08-18-non-primitive-data.pdf

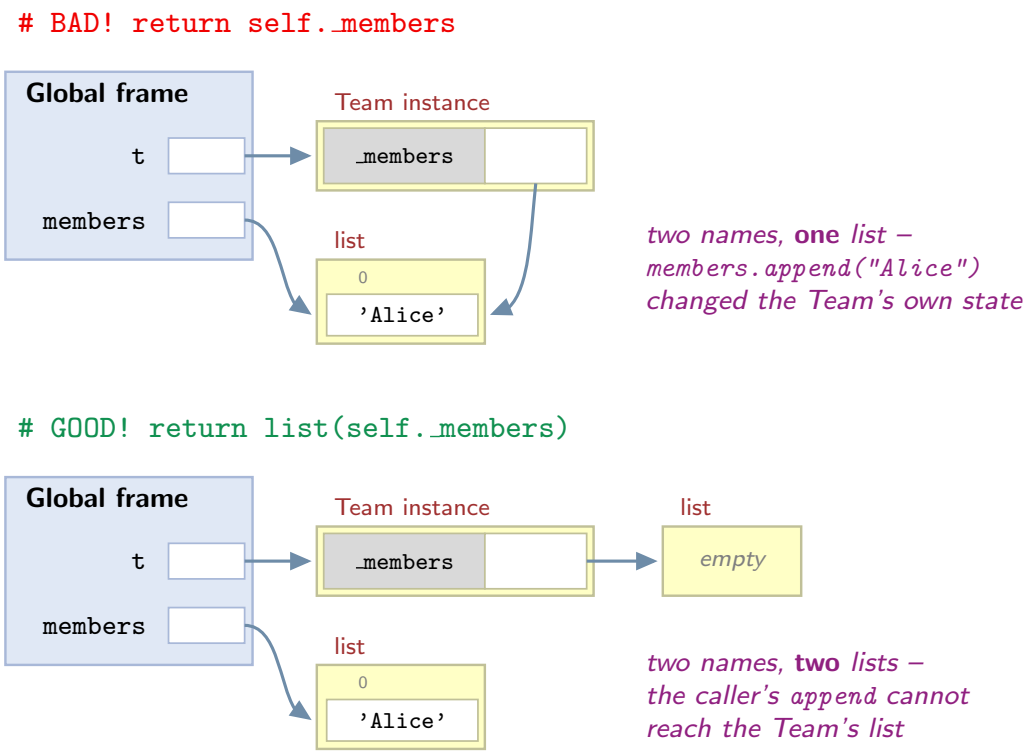

Figure 1: Why 'never expose mutable objects directly'. Top: `get_members` returns the real list, so `t._members` and the caller's `members` name the **same** object and the caller's `append` mutates the Team without any `Team` method running. Bottom: `list(self._members)` hands back a copy, so the append lands where the Team cannot see it. Aliasing is two arrows converging on one object; a copy is two arrows to two objects.

- At the end of methods that modify object state.
- At the beginning of methods that rely on invariants being true.

```
class Fraction():
    def __init__(self, numer: int, denom: int) -> None:
        self._numer = numer
        self._denom = denom
        self._check_invariants()

    def _check_invariants(self) -> None:
        """Verify that representation invariants hold."""
        assert isinstance(self._numer, int), "Numerator must be an integer."
        assert isinstance(self._denom, int), "Denominator must be an integer."
        assert self._denom != 0, "Denominator cannot be zero."
        assert self._denom > 0, "Denominator must be positive."
        if self._numer == 0:
            assert self._denom == 1, "Zero must be 0/1"
```

Compared with the assertion version above, the centralised version drops the `if denom < 0` normalisation and asserts `self._denom > 0` instead. Everything is now *checked* and nothing is *fixed up*, so `Fraction(1, -2)` raises here where the earlier version quietly rewrote it to `-1/2`. That is the trade the pattern makes — one method that says what a valid object looks like, and no hidden repair.

Best practices

- Document invariants in docstrings.
- Centralise invariant checking in a single method.
- Do not expose methods that could violate invariants.
- Create comprehensive test cases focusing on edge cases (see 2025-09-01-testing[5]).
- Python's type hints can express some invariants.
- Balance strictness with practicality.

Exercise: `DroneFlight`

You have joined a start-up that records quick test flights for hobby drones. Write a minimal `DroneFlight` class that always keeps its state valid by enforcing these representation invariants:

[5] courses/csse1001/2025-09-01-testing.pdf

- **Drone ID** — exactly six alphanumeric characters (e.g. `"A1B2C3"`).
- **Altitude** — must stay within 0 m–120 m, the CASA (Civil Aviation Safety Authority) legal ceiling for recreational drones.
- **Battery** — an integer percentage in the range 0–100.

Solution

```
class DroneFlight:
    """
    Model a quick test-flight for a small hobby drone.

    Attributes:
    _id (str) : 6-character alphanumeric flight identifier.
    _altitude (float): Current altitude in metres above launch point.
    _battery (int): Remaining battery charge as a percentage (0-100).

    Representation invariants:
    - _id is a 6-character alphanumeric string
    - 0   _altitude   120
    - 0   _battery   100
    """

    _ID_LEN = 6         # one place to change if the rule changes
    _MAX_ALT = 120.0    # metres (CASA limit)

    def __init__(self, flight_id: str) -> None:
        self._id = flight_id
        self._altitude = 0.0
        self._battery = 100
        self._check_invariants()

    def _check_invariants(self) -> None:
        assert (
            isinstance(self._id, str)
            and self._id.isalnum()
            and len(self._id) == self._ID_LEN
        ), "ID must be a 6-character alphanumeric string."
        assert 0.0 <= self._altitude <= self._MAX_ALT, "Altitude out of range."
        assert 0 <= self._battery <= 100, "Battery out of range."

    # Getter and setter methods
    def get_flight_id(self) -> str:
```

```
        """Return the immutable flight identifier."""
        return self._id

    def get_altitude(self) -> float:
        """Return current altitude in metres."""
        return self._altitude

    def get_battery(self) -> int:
        """Return remaining battery charge (percentage)."""
        return self._battery

    def set_altitude(self, value: float) -> None:
        """Set altitude, clamping to legal ceiling."""
        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()

    def __repr__(self) -> str:
        return (f"DroneFlight({self._id}, "
                f"alt={self._altitude:.1f} m, bat={self._battery} %)")

    def ascend(self, metres: float) -> None:
        """Climb *metres* metres (cannot exceed the legal ceiling)."""
        if metres < 0:
            raise ValueError("ascend() expects a non-negative distance.")
        self.set_altitude(min(self._altitude + metres, self._MAX_ALT))

    def land(self) -> None:
        """Land the drone and consume 5 % battery."""
        self.set_altitude(0.0)
        self._battery = max(self._battery - 5, 0)
        self._check_invariants()
```

Three techniques in this solution are the transferable part of the exercise:

- **Class constants for the limits.** `_ID_LEN` and `_MAX_ALT` are class variables (see 2025-09-16-dunder-methods[6]), so the code enforcing those two limits reads them from a single place: *one place to change if the rule changes*, as the comment on `_ID_LEN` says. The class

[6] courses/csse1001/2025-09-16-dunder-methods.pdf

only half keeps the promise, though: the docstring above still spells out `0 _altitude 120` in words, and the battery bounds are written literally as `100` in `__init__` and `0`/`100` in `_check_invariants()` with no constant behind them.

- **`assert` for bugs, `raise` for callers.** `_check_invariants()` uses `assert` — it is looking for a broken *implementation*, and it is fine for it to vanish under `python -O`. `set_altitude()` uses `raise TypeError` / `raise ValueError` — it is validating an *argument the caller supplied*, and that check must run in production. Both appear in the same class, deliberately.
- **Mutation routed through one method.** `ascend()` and `land()` do not touch `self._altitude`; they call `set_altitude()`, so there is exactly one place that can change the altitude and exactly one place the range check has to live. `ascend()` also clamps with `min(..., self._MAX_ALT)` before calling, so a legal climb past the ceiling stops at the ceiling instead of raising.

```
>>> d = DroneFlight("ABC123")
>>> d.ascend(50)
>>> print(d)
DroneFlight(ABC123, alt=50.0 m, bat=100 %)

>>> d.altitude = 200
Caught: Altitude must be 0-120 m.

>>> DroneFlight("BAD!")
Caught: ID must be a 6-character alphanumeric string.

>>> d.land()
DroneFlight(ABC123, alt=0.0 m, bat=95 %)
```

The `d.altitude = 200` line in the transcript appears to raise a "Caught" error, but as written this class only defines `set_altitude()` as an ordinary method — it has no `@property`/`@altitude.setter`. Plain attribute assignment like `d.altitude = 200` doesn't call `set_altitude()` at all; it silently creates a *brand-new*, unrelated `altitude` attribute (alongside the real `_altitude`) and raises nothing. To actually trigger the validation, the call has to be `d.set_altitude(200)` wrapped in a `try`/`except (TypeError, ValueError) as e: print("Caught:", e)`. The `Caught:` prefix on both error lines comes from a wrapper of that kind. Constructing `DroneFlight("BAD!")` is legitimate, though — `__init__` really does call `_check_invariants()`, so an invalid ID genuinely raises an `AssertionError` there (assuming it's likewise wrapped in a `try`/`except AssertionError`). `_check_invariants()` would *not* catch the stray `altitude` attribute either: it only ever inspects `_id`, `_altitude` and `_battery`.

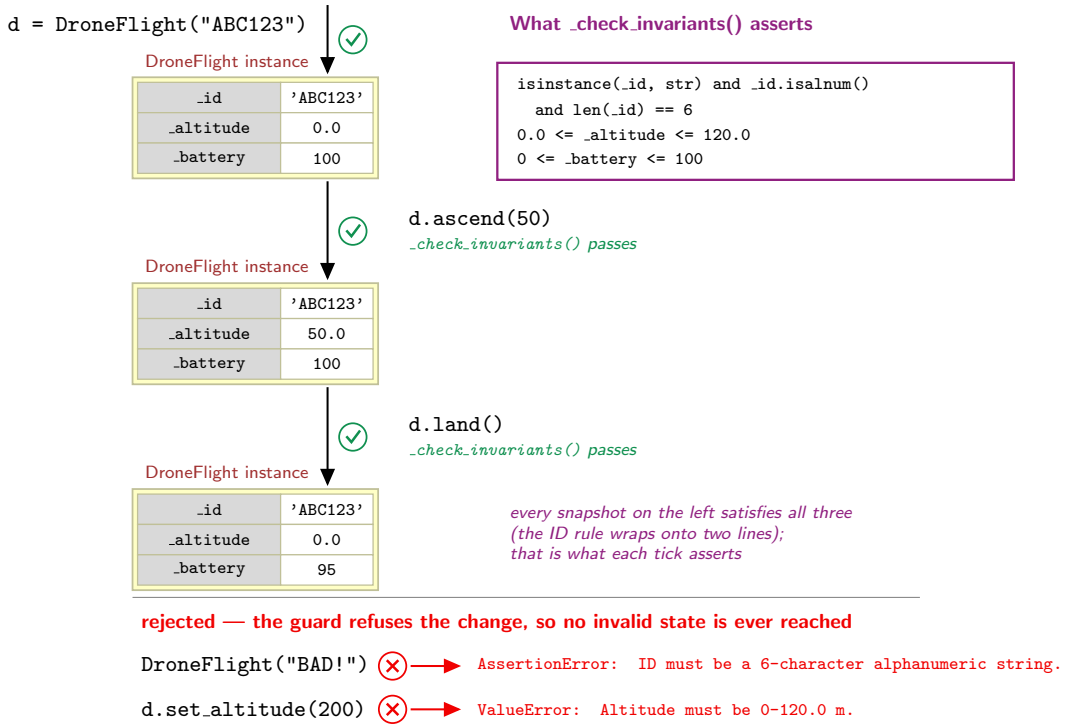

Figure 2: The `DroneFlight` demo as a state trace: each accepted call lands the object in a snapshot that satisfies all three documented invariants, and `_check_invariants()` is what asserts it. The panel on the right gives that method's three `assert` conditions, with the class constants replaced by their values (`_ID_LEN` → 6, `_MAX_ALT` → 120.0) so the bounds can be read off directly; they are the same three invariants the docstring states in words. The rejected calls below never produce a snapshot at all — the guard fires first, so no invalid state is ever reachable. A green tick is an accepted call and a red cross is a rejected one. The rejected altitude change is drawn as `d.set_altitude(200)`, the call that actually raises, because `d.altitude = 200` does not call the setter at all; its message reads `0–120.0 m`, which is what the class's own f-string over `_MAX_ALT = 120.0` produces.

One further mismatch: `_MAX_ALT` is `120.0`, so the f-string in `set_altitude` produces `"Altitude must be 0-120.0 m."`, not the `0-120 m` in the transcript. Figure 2 shows the message the listed code would actually produce, with `d.altitude = 200` replaced by the `set_altitude` call that reaches that f-string.

Summary

- Representation invariants define the valid states of an object.
- They help catch bugs early and document assumptions (mostly during development time).
- We can use `_check_invariants()` to verify invariants, and call it after state changes.
- Good invariants are specific, testable conditions.

Next: composition and inheritance (Week 9) — 2025-09-23-composition[7] and 2025-09-23-inheritance[8].

[7] courses/csse1001/2025-09-23-composition.pdf
[8] courses/csse1001/2025-09-23-inheritance.pdf