Composition

Table of contents

Outline . . . 1
Recap . . . 1
Reusing code . . . 2
Composition . . . 2
Is-a or has-a . . . 4
Worked example: `DroneFlight` . . . 4
Hot-swapping . . . 8
Summary . . . 10

Lecture 9A. See csse1001[1] for course logistics and python-composition[2] for the full reference on has-a relationships.

Outline

- The DRY principle, and two ways to reuse classes: composition and inheritance
- Composition ("has-a" relationships)
- Worked example: `DroneFlight` composed of `Battery`, `Camera`, and `Motor`
- Hot-swapping composed objects

Recap

Dunder methods from last lecture (see 2025-09-16-dunder-methods[3], python-dunder-methods[4]):

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-composition.pdf
[3] courses/csse1001/2025-09-16-dunder-methods.pdf
[4] courses/csse1001/python-dunder-methods.pdf

Category	Key methods	What they enable
Lifecycle	`__init__`	Control object initialisation
String rep	`__repr__`, `__str__`	Unambiguous & user-friendly text
Numeric ops	`__add__`, `__sub__`, `__mul__`, …	Custom arithmetic (`1/2 + 1/3`, `v1 * v2`)
Comparisons	`__eq__`, `__lt__`, `__gt__`, …	Sorting, `==`, `<`

And representation invariants (see 2025-09-16-representation-invariants[5], python-representation-invariants[6]):

What?	Why?	How?
Conditions that must hold for an object's private state at all times (e.g. $0 \leq$ altitude ≤ 120, ID is 6 digits).	Prevents invalid states; simplifies reasoning & testing (assume invariants true); catches bugs early with clear, localised checks.	Set valid values in `__init__`; centralise assertions in a private helper like `_check_invariants()`.

Reusing code

DRY (don't repeat yourself) principle: in general, duplication and copy/paste are bad:

- Increased risk of bugs.
- Maintenance overhead.
- Poor readability.
- Code bloat.
- Signals that a common abstraction/refactoring is needed.

There are two ways to reuse *classes* in a new class: **composition** and **inheritance**.

Composition

We can use objects built from other classes (implemented by us or others) as attributes of our new class/object. This isn't new — we've already used integers, strings, lists, etc. as attributes for our classes; we can do this with other (more complex) classes as well. For example, if we are

[5] courses/csse1001/2025-09-16-representation-invariants.pdf
[6] courses/csse1001/python-representation-invariants.pdf

implementing a `Robot` class, we can use an object from a `Battery` class (already implemented by us or others) as one of its attributes.

Modularity:

- A `Student` **has a** `String` (e.g. to store the student's name).
- A `Robot` **has a** `Sensor`, `Motor`, etc.

Has-a relationship

When an object (say `Car`) includes another object inside itself as an attribute (say `Engine`), this forms a **has-a** relationship — we call this **class composition**:

```
>>> class Car():
...     def __init__(self, engine : Engine) -> None:
...             self._engine = engine        # Car has-a engine.
```

The relationship is drawn as one box **inside** another, not as a line between two boxes. The whole encloses the part:

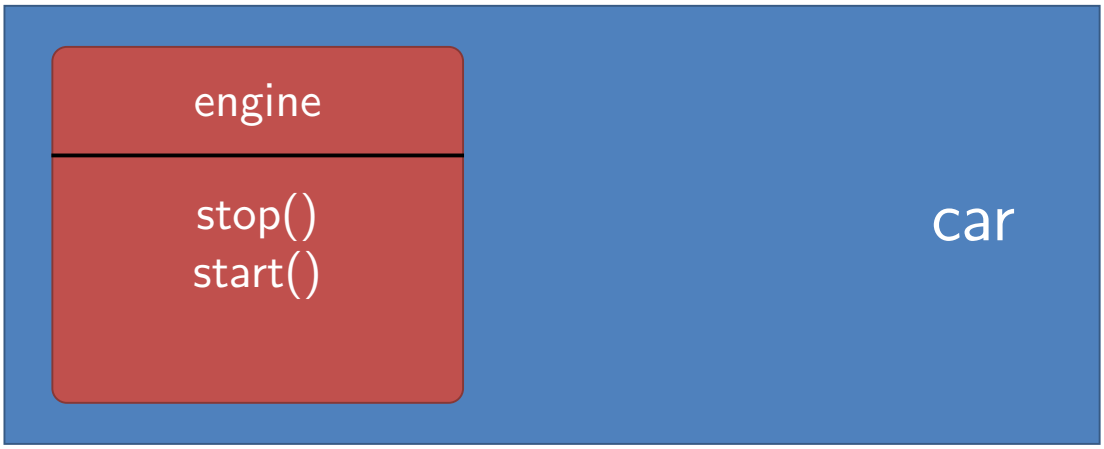

Figure 1: Composition as containment. The steel-blue rectangle is the whole (the `car` object); the brick-red rounded rectangle nested inside it is the part (the `engine` object), with its own methods `stop()` and `start()` listed under a rule. Nesting is the notation: a has-a relationship is drawn by putting the part inside the whole. A connecting line comes with UML in week 10.

Using it:

```
engine = Engine()
car = Car(engine)
car.engine.start()
car.engine.stop()
print(car.engine.power)
```

> **i** `car.engine` does not match `self._engine`
>
> The snippet reaches into the part as `car.engine`, but the `__init__` above stores the part as `self._engine`, a private attribute. Written literally, `car.engine` would raise `AttributeError`.
> The snippet is there for the shape of the relationship, not as a runnable transcript. The same mismatch recurs in the hot-swapping example below, where it changes what the code does.

Is-a or has-a

Recall the `DroneFlight` representation-invariants exercise from last lecture. The brief:

- **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.

The recap table above states the ID rule more loosely as "ID is 6 digits". "Six alphanumeric characters" is the version to work from, since `"A1B2C3"` is not six digits.

"A Drone has a Camera" reads clearly; *"A Drone is a Camera"* does not. A drone **has a** camera, **has a** motor, and **has a** battery, so composition is the natural fit here, not inheritance.

Worked example: `DroneFlight`

```
class Battery:
    def __init__(self, capacity: int = 100) -> None:
        if capacity < 0:
            raise ValueError("Capacity must be non-negative.")
        self._level = capacity   # percentage

    def use_power(self, amount: int) -> None:
        if amount < 0:
            raise ValueError(f"Power usage must be non-negative.")
```

```
        self._level -= amount
        if self._level < 0:  self._level = 0
        print(f"Power used: {amount}%. Remaining: {self._level}%.")

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


class Camera:
    def __init__(self, resolution: str = "12MP") -> None:
        self._resolution = resolution

    def take_photo(self) -> None:
        print(f"Photo taken at {self._resolution} resolution.")


class Motor:
    def __init__(self, power_rating: float = 100.0) -> None:
        self._power_rating = power_rating   # Watts
        self._is_running = False

    def start(self) -> None:
        self._is_running = True
        print("Motor started.")

    def stop(self) -> None:
        self._is_running = False
        print("Motor stopped.")
```

Each of the three is a complete, self-contained class that knows nothing about drones. `DroneFlight` then holds one of each:

```
class DroneFlight:
    _ID_LEN = 6
    _MAX_ALT = 120.0   # metres (CASA limit)

    def __init__(self, flight_id: str, battery: Battery,
                 camera: Camera, motor: Motor) -> None:
        self._id = flight_id
        self._altitude = 0.0
        self._battery = battery   # Drone has-a Battery
```

```
        self._camera = camera      # Drone has-a Camera
        self._motor = motor        # Drone has-a Motor
        self._check_invariants()
```

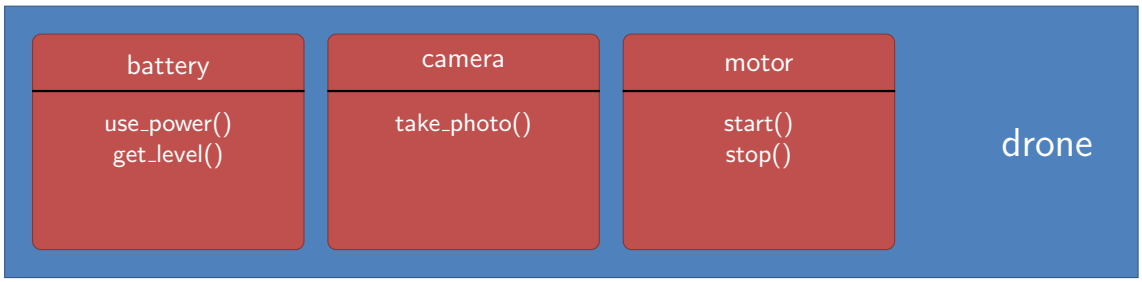

Figure 2: The `DroneFlight` object: one whole holding three parts, each part a complete class in its own right. The `battery`, `camera` and `motor` boxes are nested inside the `drone` box, the same containment notation as `car` and `engine`, and each part lists its own methods under a rule.

The rest of `DroneFlight` is written entirely in terms of those three parts. It never touches their internals, only their public 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.get_level()

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

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

```
        self._check_invariants()

    def land(self) -> None:
        """Land the drone and consume 5% battery."""
        self.set_altitude(0.0)
        self._battery.use_power(5)
        if self._motor.is_running():
            self._motor.stop()
        self._check_invariants()

    def take_photo(self) -> None:
        if self._battery.get_level() < 5:
            print("Not enough battery to take photo.")
            return
        self._camera.take_photo()
        self._battery.use_power(5)
```

`get_battery()` delegates straight to `self._battery.get_level()`. Don't call `use_power(0)` here. Just show the level. Even a zero-amount power use would print a spurious `Power used: 0%. Remaining: ...` line and would be conceptually the wrong operation: a getter reports state, it does not perform an action on the part.

Every method above has the same shape: `DroneFlight` decides *when* something happens and *what the rules are* (the 5% battery threshold, the 120 m ceiling), and the part decides *how* it happens. That division is the whole point of composition. `take_photo()` is the clearest case: the drone checks the battery, the camera takes the picture, the battery bills for it, and no class knows more than its own job.

Gaps in the listing

The listing above is not self-contained. It has four gaps:

- **`set_altitude()` is called but not defined above.** Both `ascend()` and `land()` go through it rather than assigning `self._altitude` directly, which is the right instinct, because a setter is where an invariant check belongs. Lecture 8B's `DroneFlight` defines one.
- **`_check_invariants()` is called but not defined above either.** It is the helper from the previous lecture.
- **`Motor` has no `is_running()` method.** `land()` calls `self._motor.is_running()`, but the `Motor` class above defines only the attribute `self._is_running` and the methods `start()` and `stop()`. As written, `land()` would raise `AttributeError`; the class needs a getter (or `land()` needs to consult `self._motor._is_running`, which would reach into a private attribute and defeat the encapsulation the example is demonstrating).

- `_ID_LEN = 6` is declared as a class constant but not used in any method above. It belongs to the invariant check, which is not in the listing.

Transcript

```
>>> b = Battery(); c = Camera (); m = Motor()
>>> d = DroneFlight("SAD564", b, c, m)
>>> d.ascend(80); d.take_photo()
Photo taken at 12 MP resolution.
Power used: 5%. Remaining: 95%
>>> d.land()
Power used: 5%. Remaining: 90%
>>> print(d)
DroneFlight(SAD564, alt=0.0 m, bat=90 %)
```

This transcript is not exact output. The default resolution is the string `"12MP"`, so the real line reads `12MP resolution`. `use_power` ends its message with a full stop, and `__str__` has no space before the `%`. Read it for the sequence of events, not character by character.

The sequence itself is right: `ascend(80)` is silent, `take_photo()` costs 5%, `land()` costs another 5%, and the final state is 0.0 m and 90%. That holds only once the pieces missing from the listing are supplied. With `set_altitude()` and `_check_invariants()` from lecture 8B, which defines both, one gap still bites: `land()` calls `self._battery.use_power(5)` before `self._motor.is_running()`, so it prints its battery line and *then* raises `AttributeError`, because `Motor` has no `is_running()`. The closing `print(d)` is not reached in a real session.

Hot-swapping

Because a composed attribute is just a reference to an object, it can in principle be replaced by any other object that supports the operations actually used on it. No shared base class is required. Take a camera that isn't a `Camera`:

```
class ThermalCamera:
    def __init__(self, resolution: str = "12MP"):
        self.resolution = resolution

    def take_photo(self):
        print(f"Thermal image saved.")
```

```
>>> b = Battery(); c = Camera (); m = Motor()
>>> d = DroneFlight("SAD564", b, c, m)
>>> d.take_photo()
Photo taken at 12 MP resolution.
Power used: 5%. Remaining: 95%
>>> d.camera = ThermalCamera()
>>> d.take_photo()
```

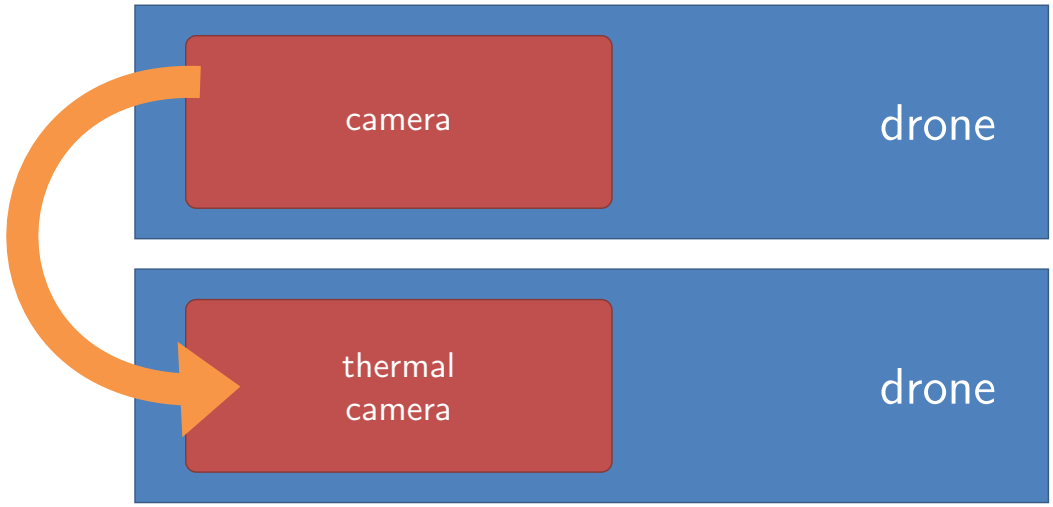

Figure 3: Hot-swapping. The same whole, drawn twice: the part inside it is replaced, and everything around it is untouched. The thick curved orange arrow from the old configuration to the new marks the substitution.

`ThermalCamera` does not inherit from `Camera` at all. It just happens to provide a `take_photo()` method with a compatible signature, so it works as a drop-in replacement. This is **duck typing**, a standard Python term: Python never checks that the new object is "officially" a `Camera`, only that it supports the operations actually performed on it. Contrast this with inheritance, where the substitute must be a subclass; see 2025-09-23-inheritance[7].

> ⚠ `d.camera = ...` does not actually swap anything
>
> `DroneFlight.__init__` stores the camera as `self._camera`, and `take_photo()` calls `self._camera.take_photo()`. Assigning `d.camera = ThermalCamera()` creates a *brand new public attribute* called `camera` and leaves `_camera` — the one the class actually uses — pointing at the original `Camera`. The final `d.take_photo()` above would print `Photo taken at 12MP resolution.` again, not `Thermal image saved.`

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

> The idea is sound; the line as written misses the underscore. A working swap needs `d._camera = ThermalCamera()` (reaching into private state from outside, which the course discourages) or, better, a small setter on `DroneFlight` that does it deliberately.

python-composition[8] spells out that better version for the `Car`/`Engine` pair: a dedicated `swap_engine()` method that stops the currently-running engine *before* replacing it. A method gets the chance to clean up the object it is discarding; a bare attribute assignment does not.

Summary

Build by pieces: snap self-contained objects together to create a richer whole.

- **Modular**: each component handles one clear task.
- **Reusable**: same parts work in new contexts; no code rewrite.
- **Readable**: data flow is explicit and easy to trace.
- **Few side effects**: isolated state keeps surprises and bugs low.

Next: 2025-09-23-inheritance[9] (Lecture 9B), the other way of reusing a class.

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