Python Composition

Table of contents

Basic pattern 1
Worked example: `Car`/`Engine` 2
Hot-swapping a part 4
Composition vs. inheritance 4

Composition is a way of building a class out of other objects: an object of one class holds an object of another class as one of its attributes. This forms a **has-a** relationship (e.g. a `Car` **has-a** `Engine`) — as opposed to inheritance's **is-a** relationship (e.g. a `SportsCar` **is-a** `Car`).

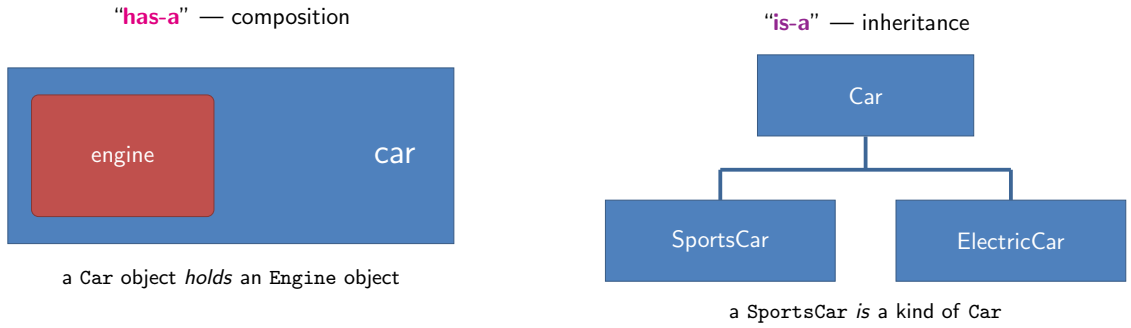

Figure 1: The two ways of reusing a class, side by side. On the left, composition is **containment**: the part sits inside the whole, and there is no connecting line. On the right, inheritance is a **hierarchy**: parent above, children below, joined by a stem-and-bus connector with no arrowhead. *has-a* is lettered in magenta and *is-a* in purple.

Basic pattern

```
class Car():
    def __init__(self, engine: Engine) -> None:
```

```
        self._engine = engine   # Car has-a engine

    def start(self):
        self._engine.start()
```

The outer object (`Car`) doesn't need to know *how* the inner object (`Engine`) works internally — it just calls the inner object's public methods. This keeps each class focused on a single responsibility.

Delegation

The method `Car.start()` above does no work of its own: it forwards the request to the part that knows how to do it. That forwarding is called **delegation**, and it is what most methods on a composed class end up doing. The outer class keeps the *policy* — when something may happen, and under what conditions — and the inner class keeps the *mechanism*.

The `DroneFlight` example in 2025-09-23-composition[1] is the clearest case: the drone decides that a photo costs 5% battery and is refused below 5%, but it is the `Camera` that takes the picture and the `Battery` that tracks the charge.

The part is passed in, not created inside

`Car.__init__` takes an `Engine` as a parameter rather than writing `self._engine = Engine()`. Both are composition, but only the first lets the caller decide *which* engine, and only the first makes the part replaceable. It is also what makes the class testable — a test can pass in a stand-in object instead of the real part.

Worked example: `Car/Engine`

This example stores the part in a **public** `engine` attribute, so that `swap_engine()` has something ordinary to reassign, where the basic pattern above uses the private `_engine`.

```
class Engine:
    def __init__(self, horsepower, fuel_type="gasoline"):
        self.horsepower = horsepower
        self.fuel_type = fuel_type
        self.running = False

    def start(self):
```

[1] courses/csse1001/2025-09-23-composition.pdf

```
        if not self.running:
            self.running = True
            print(f"{self.fuel_type.capitalize()} engine with {self.horsepower} HP started.")
        else:
            print("Engine is already running.")

    def stop(self):
        if self.running:
            self.running = False
            print("Engine stopped.")
        else:
            print("Engine is already off.")


class Car:
    def __init__(self, make, model, engine: Engine):
        self.make = make
        self.model = model
        self.engine = engine

    def start(self):
        print(f"Starting the {self.make} {self.model}...")
        self.engine.start()

    def stop(self):
        print(f"Stopping the {self.make} {self.model}...")
        self.engine.stop()

    def swap_engine(self, new_engine: Engine):
        if self.engine.running:
            print("Stopping current engine before swap...")
            self.engine.stop()
        print(f"Swapping engine in {self.make} {self.model}...")
        self.engine = new_engine
        print("New engine installed!")

>>> small_engine = Engine(150, "gasoline")
>>> car = Car("Toyota", "Corolla", small_engine)
>>> car.start()
Starting the Toyota Corolla...
Gasoline engine with 150 HP started.
```

```
>>> big_engine = Engine(300, "diesel")
>>> car.swap_engine(big_engine)   # safely stops the old engine first, if running
```

Hot-swapping a part

Because a composed attribute is just a reference, it can be replaced at runtime by any object that supports the operations actually performed on it — **duck typing**, no shared base class required. The replacement need not be a subclass, or related to the original at all; it only needs the right methods.

Two things to get right when doing this:

- **Swap the attribute the class actually uses.** If `__init__` stores the part as `self._engine` (with the leading underscore that marks it private), then assigning `car.engine = new_engine` from outside creates an unrelated *new* public attribute and leaves the real part untouched. The object will carry on using the old one. This is an easy mistake, because the swap looks like it worked and nothing raises an error.
- **Give the old part a chance to shut down.** A bare attribute assignment discards the previous object silently, mid-operation if it happens to be running. A dedicated method (`swap_engine()` above) can stop it first.

Composition vs. inheritance

- Use **composition** ("has-a") when an object is *made up of* other objects, or *uses* another object to do part of its job.
- Use **inheritance** ("is-a") when a new class is a more specific *version* of an existing class — see python-inheritance[2] and 2025-09-23-inheritance[3].

Composition tends to be more flexible: a composed attribute can be freely swapped for any object supporting the same interface, without needing any shared base class.

The two are not rivals within a single program, either. A class built by composition can perfectly well be subclassed: a `SurveyDrone` **is a** `DroneFlight`, which **has a** `Camera`.

[2] courses/csse1001/python-inheritance.pdf
[3] courses/csse1001/2025-09-23-inheritance.pdf