CSSE1001 — Week 9 Notes

Table of contents

Composition

See csse1001[1] for course logistics — this note covers Lecture 9A's technical content. See python-composition[2] for the full reference on has-a relationships.

Today's 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

A quick recap table of last lecture's dunder methods:

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:

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()`.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-composition.pdf

How to reuse 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.

Today: how to reuse *classes* in a new class? Via **composition** or **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 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

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

Which is clearer: "is-a" or "has-a"?

Recall the `DroneFlight` representation-invariants exercise from last lecture (Drone ID, altitude, battery). Which one sounds clearer: *"A Drone is a Camera"* or *"A Drone has a Camera"*? A drone "has a" camera, "has a" motor, and "has a" battery — 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("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.")


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()

    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()   # don't call use_power(0) here! just show the leve

    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("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)
```

```
>>> 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 %)
```

`get_battery()` delegates straight to `self._battery.get_level()` rather than `self._battery.use_power(0)` — the comment on the slide ("Don't call use_power(0) here! Just show the level.") is a reminder that even a "zero-amount" power use would still print a spurious "Power used: 0%…" message and is conceptually the wrong operation for a getter to perform. A getter should just report state, not have side effects.

Hot-swapping

Because a composed attribute is just an object reference, it can be swapped out for another compatible object at any time:

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

```
    def take_photo(self):
        print("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()
```

`ThermalCamera` doesn't 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 an example of *duck typing*: Python doesn't check that the new object is "officially" a `Camera`, only that it supports the operations actually used on it.

> The companion `composition.py` file demonstrates a more defensive hot-swap via a dedicated `swap_engine()` method on its `Car`/`Engine` example, which stops the currently-running engine *before* swapping it out — a small improvement over directly reassigning an attribute mid-flight, since a direct reassignment (like `d.camera = ThermalCamera()` above) doesn't get a chance to clean up the object it's replacing. See python-composition[3] for the full listing.

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[4] (Lecture 9B).

Inheritance

See csse1001[5] for course logistics — this note covers Lecture 9B's technical content. See python-inheritance[6] for the full reference on subclassing, `super()`, and polymorphism.

[3] courses/csse1001/python-composition.pdf
[4] courses/csse1001/2025-09-23-inheritance.pdf
[5] courses/csse1001/csse1001.pdf
[6] courses/csse1001/python-inheritance.pdf

Today's outline

- Inheritance and "is-a" relationships
- Terminology: subclass/superclass, child/parent, extends
- Inheriting, introducing, and overriding methods
- `super()` and extending methods
- Polymorphism

Inheritance

Inheritance is another way of reusing code in classes. By inheriting from a class, we can create specialized (sub)classes while reusing attributes/methods from the original class. Key idea: building a hierarchy of classes.

```
        Robot
   /      |       \
DeliveryRobot  CleanerRobot  HumanoidRobot
```

This is an **"is-a"** relationship: a `DeliveryRobot` *is a* `Robot`. `DeliveryRobot`, `CleanerRobot`, and `HumanoidRobot` all specialize the base class (`Robot`):

- They inherit the attributes and methods of the base class.
- They can **override** the attributes/methods of the base class (e.g. a `move()` method might have different implementations).
- They can have new (unique) attributes/methods or implementations.

More examples of "is-a" hierarchies:

- `Employee` → `Programmer`, `Manager`, `Admin`
- `Animal` → `Cat`, `Dog`, `Fox`; and `Dog` → `ServiceDog`
- `Person` → `Professor`, `Student`; and `Student` → `UndergradStudent`, `GradStudent`

Terminology

Given `class Student(Person):` — all of the following mean the same thing, and can be used interchangeably:

- The `Student` class **inherits** from the `Person` class.
- The `Student` class is a **subclass** of the `Person` class.
- The `Student` class is a **child class** of the `Person` class.
- The `Person` class is a **superclass** of the `Student` class.
- The `Person` class is a **parent class** of the `Student` class.
- (less often) The `Student` class **extends** the `Person` class.

Reusing code via inheritance

Without inheritance, two robot classes end up duplicating `__init__` and `charge`:

```
# BAD -- duplicating code, violating the DRY principle
class CleaningRobot:
    def __init__(self, name: str, batt_level: int):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")

    def clean(self):
        print(f"{self.name} is cleaning the floor.")

class DeliveryRobot:
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        self.name = name
        self.batt_level = batt_level
        self.load_capacity = load_capacity

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")

    def deliver(self):
        print(f"{self.name} is delivering a package up to {self.load_capacity}kg.")
```

Instead, extract the common behaviour into a **base class**, and inherit specialized robots from it:

```
class Robot:
    def __init__(self, name: str, batt_level: int):
        self.name = name
        self.batt_level = batt_level

    def charge(self):
        self.batt_level = 100
        print(f"{self.name} is fully charged!")
```

```
class CleaningRobot(Robot):
    def clean(self):
        print(f"{self.name} is cleaning the floor.")
```

The syntax `class Child(Parent):` means the child class inherits from the parent class.

```
>>> cleaner = CleaningRobot("Roomba", 80)   # calls Robot's (base class') __init__
>>> cleaner.charge()   # CleaningRobot inherited charge() from the base class
Roomba is fully charged!
>>> print(cleaner.name)   # CleaningRobot inherited attributes from the base class
Roomba
```

Introducing new methods

A child class can define methods that don't exist on the base class at all — `clean()` only exists on `CleaningRobot` objects, not on plain `Robot` objects.

Overriding methods

We can **override** a method by simply declaring one of the same name in the child class:

```
class FastChargingRobot(Robot):
    def charge(self):   # overriding the charge method from the base class
        self.batt_level = 100
        print(f"{self.name} is fully charged in record time!")
```

Calling `charge()` on a `FastChargingRobot` object calls *this* overridden method, not `Robot`'s.

Extending methods with `super()`

`super()` specifies that we want to use the method from the *superclass*, rather than the child class — useful for extending (rather than fully replacing) inherited behaviour:

```
class DeliveryRobot(Robot):
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        # super().__init__(...) calls the __init__ method from the parent (base) class
        super().__init__(name, batt_level)
        self.load_capacity = load_capacity
```

```
    def deliver(self):
        print(f"{self.name} is delivering a package up to {self.load_capacity}kg.")

>>> deliverer = DeliveryRobot("DHLBot", 80, 10)
>>> deliverer.charge()
DHLBot is fully charged!
>>> deliverer.deliver()
DHLBot is delivering a package up to 10kg.
```

Polymorphism

The word **polymorphism** means "to take on many forms". In computer science, an interface that works over different underlying data-types is called **polymorphic**.

`len()` is polymorphic — it works across many different types:

```
>>> len({'a': 1, 'b': 2, 'c': 3})
3
>>> len("Drop Bear")
9
>>> len([1, 2, 3, 4])
4
```

Classes that share a common interface (e.g. all inheriting a `charge()` method) are polymorphic too — the same call produces different behaviour depending on the actual object's class:

```
robots = [
    CleaningRobot("Roomba", 40),
    DeliveryRobot("DHL-Bot", 20, 15),
    FastChargingRobot("FlashBot", 5)
]

for r in robots:
    r.charge()   # different behaviors depending on r's charge() method
```

Exercise

Implement the `SurveyDrone` and `DeliveryDrone` classes, inheriting from `DroneFlight`:

```
>>> class DroneFlight:
...     pass   # implemented before (see [[2025-09-23-composition]])
>>> class SurveyDrone(DroneFlight):
...     pass   # implement SurveyDrone
>>> class DeliveryDrone(DroneFlight):
...     pass   # implement DeliveryDrone
```

Left unsolved in the source — just stubs.

Summary

We can create classes that **inherit** from other classes. By doing so, the subclass automatically receives all attributes and methods implemented in the superclass. The subclass can have additional methods and attributes, while overriding inherited methods and calling methods from its superclass.

Use inheritance only when there's a clear **"is-a"** relationship.

Next: Unified Modelling Language (Week 10).

Reference material

Python Composition

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`).

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.

Worked example: `Car/Engine`

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

    def start(self):
        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
```

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.

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

Python Inheritance

Inheritance lets a class (the **subclass/child class**) reuse the attributes and methods of another class (the **superclass/parent class**), while adding new behaviour or overriding existing behaviour. It models an **"is-a"** relationship (a `DeliveryRobot` *is a* `Robot`) — contrast with python-composition[7]'s **"has-a"** relationship.

Syntax

```
class Parent():
    # parent class' implementation

class Child(Parent):
    # child class' implementation -- inherits everything from Parent
```

[7] courses/csse1001/python-composition.pdf

A subclass automatically gets all of its parent's attributes and methods. It can:

- **Add** new methods/attributes that only it has.
- **Override** an inherited method by redefining it with the same name.
- **Extend** an inherited method using `super()`, rather than fully replacing it.

`super()`

`super().method(...)` calls the *parent* class' version of a method — most commonly used inside an overridden `__init__` to reuse the parent's initialization logic before adding subclass-specific setup:

```
class DeliveryRobot(Robot):
    def __init__(self, name: str, batt_level: int, load_capacity: int):
        super().__init__(name, batt_level)   # reuse Robot's __init__
        self.load_capacity = load_capacity
```

Terminology cheat-sheet

Given `class Student(Person):`

Term	Refers to
subclass / child class	`Student`
superclass / parent class	`Person`
"`Student` inherits from / extends `Person`"	the relationship between them

Polymorphism

Polymorphism ("many forms") describes an interface that works across different underlying types. Built-ins like `len()` are polymorphic (works on strings, lists, dicts, …). Classes that share a method name (whether via a common superclass or just by convention) are also polymorphic: the same call, e.g. `r.charge()`, produces different behaviour depending on which actual class `r` is an instance of.

When to use inheritance

Only when there's a genuine **"is-a"** relationship between the two classes. If the relationship is really "has-a" (one object *contains* or *uses* another), prefer python-composition[8] instead.

[8] courses/csse1001/python-composition.pdf

Abstract base classes

An **abstract base class** doesn't provide concrete implementations for some/all of its methods — it just defines the *interface* that every concrete subclass must implement. You're not meant to instantiate the abstract class directly.

The simplest (unenforced) version just raises an error from each method that subclasses must override:

```
class Shape:
    def area(self) -> float:
        raise NotImplementedError("Subclasses must implement area()")
```

This only fails when the unimplemented method is actually *called* — `Shape()` itself still succeeds. The standard-library `abc` module enforces this properly, raising `TypeError` at the point of instantiation:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float:
        ...

    # concrete helpers are still allowed alongside abstract methods
    def describe(self) -> str:
        return f"{self.__class__.__name__}"
```

With `ABC`, `Shape()` raises immediately: `TypeError: Can't instantiate abstract class Shape with abstract methods area`.

Method Resolution Order (MRO)

When a class inherits from multiple parents (or a chain of parents) that define the same method/attribute name, Python needs a deterministic rule for which one wins. That rule is the **Method Resolution Order (MRO)** — the order Python searches through classes to resolve a name on an object. It's stored on the class as `cls.__mro__` (or `cls.mro()`).

Python's inheritance models

- **Single inheritance**: `class B(A):` — one parent.
- **Multilevel inheritance**: `class C(B):` (where `B` itself inherits from `A`) — a linear chain; the MRO just follows the chain upward.
- **Hierarchical inheritance**: multiple children (`B`, `D`, …) inherit from the same parent `A`.
- **Multiple inheritance**: `class E(B, D):` — a class inherits from more than one parent directly.

The diamond problem and C3 linearisation

If `E(B, D)` and both `B` and `D` inherit from `A`, which version of `A`'s methods should `E` use? Python resolves this with **C3 linearisation**:

- Child classes are checked before parents.
- Parents are checked in the order they're listed in the class definition.
- If a class would appear more than once, only its *last* occurrence is kept.

```
>>> class E(B, D): pass
>>> [cls.__name__ for cls in E.__mro__]
['E', 'B', 'D', 'A', 'object']
```

Each attribute/method name is resolved independently by walking this list and taking the first class that defines it — so two different method calls on the same object can effectively "come from" two different classes in the hierarchy.

`super()` follows the MRO

`super()` doesn't call the immediate parent named in the `class` statement — it calls the *next class after the current one in the instance's actual MRO*. This is what makes cooperative multiple inheritance work: as long as every class in the chain calls `super()`, a single call can ripple through every class in the MRO exactly once, in order.

```
class A:
    def ping(self): print("A")
class B(A):
    def ping(self): print("B"); super().ping()
class C(A):
    def ping(self): print("C"); super().ping()
class D(B, C):
    def ping(self): print("D"); super().ping()
```

```
>>> D().ping()
D
B
C
A
```

If any class in the chain omits its `super()` call, the chain simply stops there for that call — the MRO itself doesn't change (it's purely a function of the class hierarchy), but fewer methods actually get executed.