CSSE1001 — Week 9 Notes

Table of contents

Composition **2**
- Outline . . . 2
- Recap . . . 2
- Reusing code . . . 3
- Composition . . . 3
- Is-a or has-a . . . 5
- Worked example: `DroneFlight` . . . 5
- Hot-swapping . . . 9
- Summary . . . 10

Inheritance **11**
- Outline . . . 11
- Inheritance . . . 11
- Terminology . . . 12
- Reusing code via inheritance . . . 14
- Polymorphism . . . 16
- Exercise . . . 17
- Summary . . . 18

Reference material **18**
- CSSE1001 Diagram Notation . . . 18
- Python Composition . . . 35
- Python Dunder (Magic) Methods . . . 39
- Python Inheritance . . . 43
- Python Representation Invariants . . . 48

Composition

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]):

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]):

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

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 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:

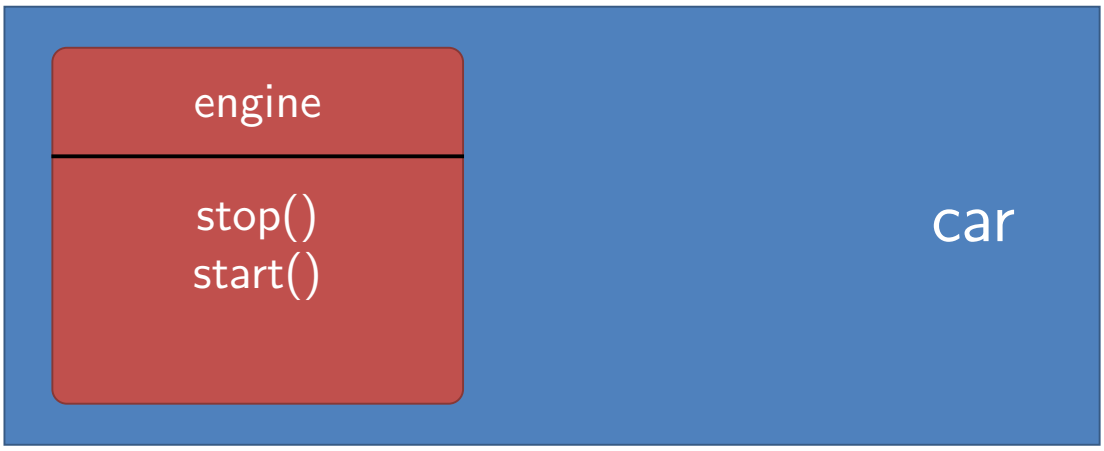

Figure 1: Composition as containment. The steel-blue rectangle is the whole (the `car` object); the brick-red rounded rectangle nested inside it is the part (the `engine` object), with its own methods `stop()` and `start()` listed under a rule. Nesting is the notation: a has-a relationship is drawn by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

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:
```

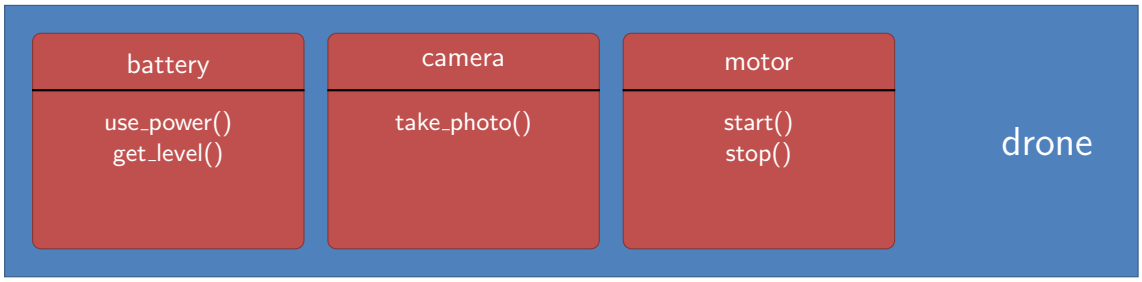

Figure 2: The `DroneFlight` object: one whole holding three parts, each part a complete class in its own right. The `battery`, `camera` and `motor` boxes are nested inside the `drone` box, the same containment notation as `car` and `engine`, and each part lists its own methods under a rule.

```
        """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()
```

`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].

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

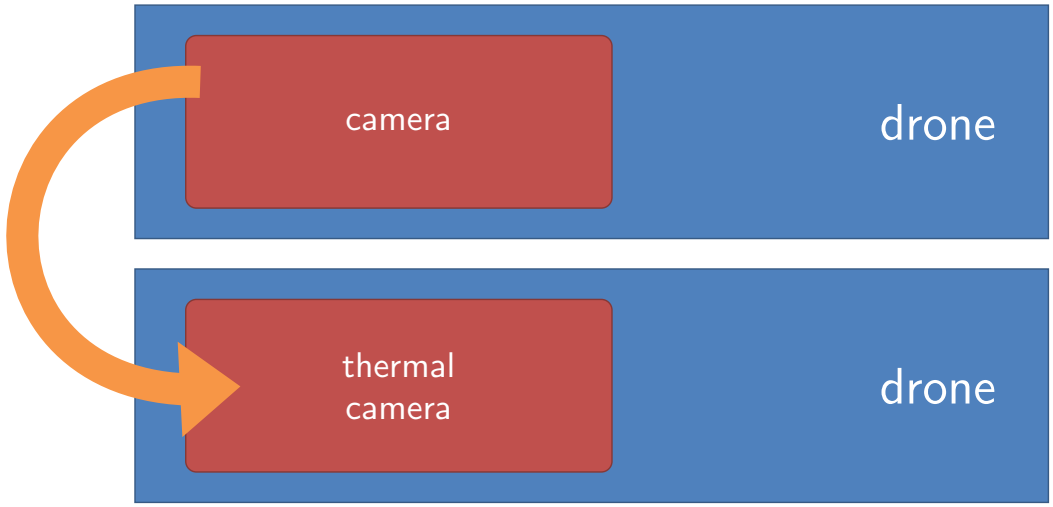

Figure 3: Hot-swapping. The same whole, drawn twice: the part inside it is replaced, and everything around it is untouched. The thick curved orange arrow from the old configuration to the new marks the substitution.

> ⚠ `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.`
> 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.

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

- **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.

Inheritance

See csse1001[10] for course logistics. This is Lecture 9B. See python-inheritance[11] for the full reference on subclassing, `super()`, and polymorphism.

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**.

This is an **"is-a"** relationship: a `DeliveryRobot` *is a* `Robot`. `DeliveryRobot`, `CleaningRobot`, 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 — for example, `move()` might have a different implementation in each.
- They can have **new (unique)** attributes/methods, or new implementations.

[9] courses/csse1001/2025-09-23-inheritance.pdf
[10] courses/csse1001/csse1001.pdf
[11] courses/csse1001/python-inheritance.pdf

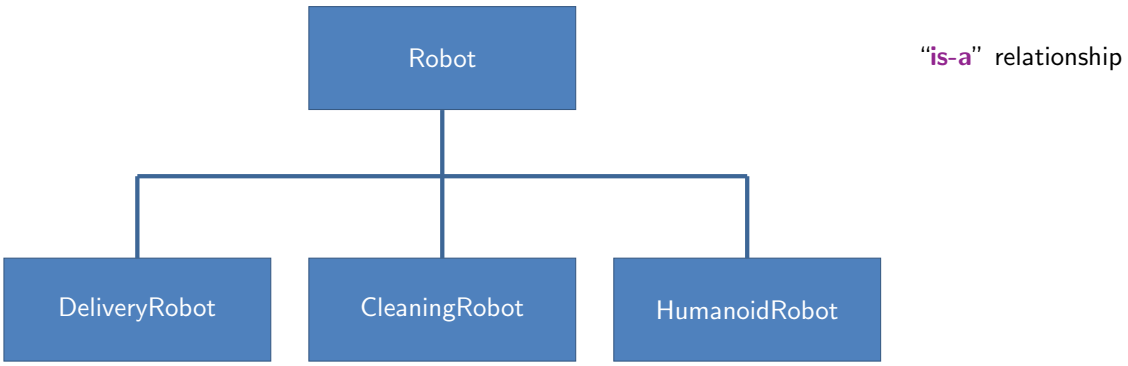

Figure 4: The `Robot` hierarchy. Parent above, children in a row beneath, joined by a single stem down to a horizontal bus with a drop to each child. The connector carries no arrowhead; the next figure draws inheritance with arrows instead.

More is-a hierarchies

Three more is-a hierarchies:

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

See csse1001-diagram-notation[12] for the two ways of drawing the inheritance connector, and 2025-10-07-uml[13] for the formal notation that supersedes both in week 10.

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.

[12] courses/csse1001/csse1001-diagram-notation.pdf
[13] courses/csse1001/2025-10-07-uml.pdf

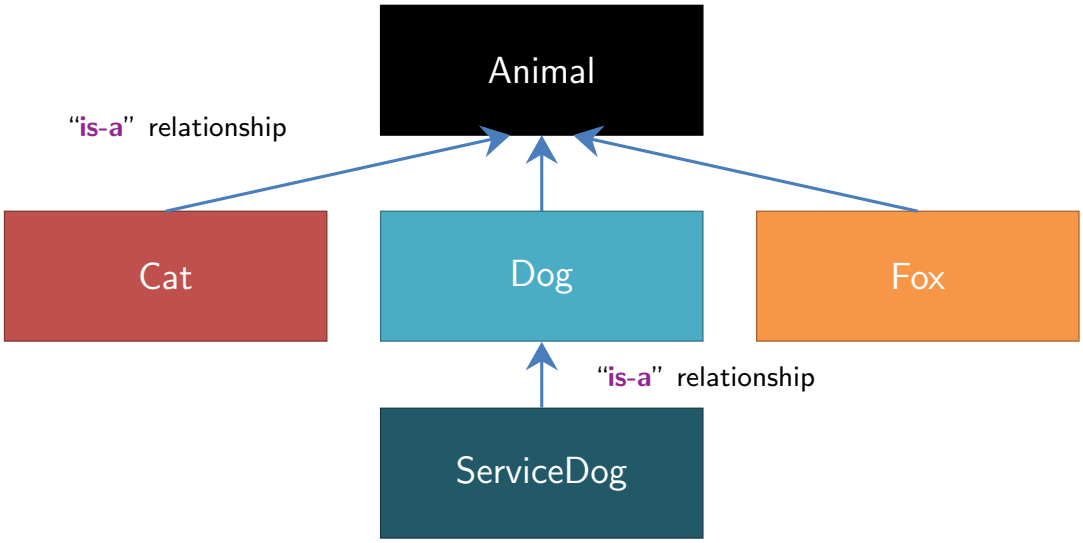

Figure 5: The `Animal` hierarchy, drawn in a different style from Figure 4. Here the root is black, each child gets its own fill, and the connectors are thin straight lines — two diagonal, one vertical — each ending in its own arrowhead on the parent's bottom edge. `ServiceDog` shows that a child can itself be a parent: it gets its own arrow into `Dog`. The child fill colours carry no meaning — `Dog` stays teal even though it is itself a parent.

Reusing code via inheritance

Start with a base `Robot`, and a `CleaningRobot` written without inheritance:

```
class Robot:
    def __init__(self, name: str, batt_level=100):
        self.name = name
        self.batt_level = batt_level

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


# CleaningRobot without inheritance -- repeated code
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.")
```

The second listing repeats two members of `Robot`: `charge` and `__init__`. Only `clean()` is genuinely new. Instead, **inherit from `Robot`**:

```
# Inherit specialized robots from the base class
class CleaningRobot(Robot):
    def clean(self):
        print(f"{self.name} is cleaning the floor.")
```

That is the entire class. The syntax `class Child(Parent):` is all it takes — see python-inheritance[14] for the annotated form of that line.

`CleaningRobot` now inherits **all attributes**, `__init__`, and `charge` from `Robot`; constructing one runs `Robot.__init__`, and `cleaner.name` and `cleaner.charge()` both work without `CleaningRobot` mentioning them.

[14] courses/csse1001/python-inheritance.pdf

The four things a subclass can do

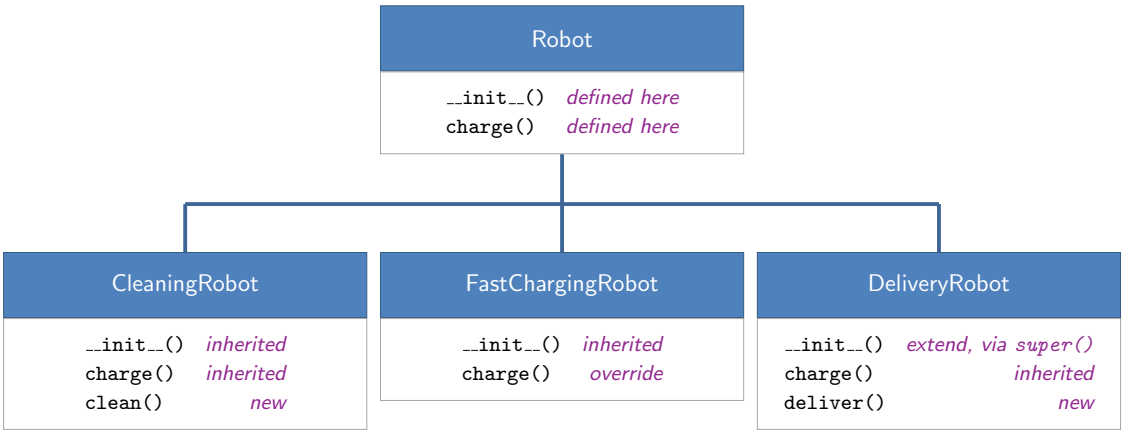

Figure 6: What each subclass does with each member of `Robot`: inherit it, override it, extend it, or add a new one. Each box lists the class's members. The purple italic annotation beside each subclass member says which of the four applies; `Robot`'s own members are marked as defined there. The three-compartment UML class box that does this properly comes in week 10.

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 — calling it on a plain `Robot` is an error.

Overriding methods

We **override** a method by declaring one of the same name in the child class, exactly as we would normally declare a method:

```
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* method, not `Robot`'s. Nothing else changes: `FastChargingRobot` still inherits `__init__` and `name`.

Extending methods with `super()`

`super()` specifies that we want to use the method from the **superclass**, rather than the child class. That is what makes it possible to *extend* inherited behaviour instead of replacing it — most often in `__init__`, where the parent already knows how to set up the attributes it defined:

```
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.")
```

`DeliveryRobot.__init__` handles only what is new (`load_capacity`) and hands `name` and `batt_level` back to `Robot.__init__`. This signature drops the `batt_level=100` default that `Robot.__init__` has — a `DeliveryRobot` must be given a battery level.

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**.

The `len` function is polymorphic:

```
>>> len({'a': 1, 'b': 2, 'c': 3})
3

>>> len("Drop Bear")
9

>>> len([1, 2, 3, 4])
4
```

Classes in one hierarchy are polymorphic in exactly the same way: the same call means "do the right thing for whatever this object actually is".

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

for r in robots:
    r.charge()   # behavior depends on class of object
```

The list holds three different classes, and the loop body does not test which is which. What each iteration runs is decided by the object, not by the loop. Following each call through the class definitions:

- `CleaningRobot` → no `charge` of its own → `Robot.charge` → *"Roomba is fully charged!"*
- `DeliveryRobot` → no `charge` of its own → `Robot.charge` → *"DHL-Bot is fully charged!"*
- `FastChargingRobot` → has its own `charge` → stops there → *"FlashBot is fully charged in record time!"*

Adding a fourth robot class later does not require touching this loop — that is what the shared interface buys.

Exercise

Implement the `SurveyDrone` and `DeliveryDrone` classes, inheriting from the `DroneFlight` class of 2025-09-23-composition[15]:

```
>>> class DroneFlight:
...     pass # Implemented before
>>> class SurveyDrone(DroneFlight):
...     pass # Implement SurveyDrone
>>> class DeliveryDrone(DroneFlight):
...     pass # Implement DroneFlight
```

Both subclasses are left as stubs to implement. The last comment should read `# Implement DeliveryDrone`.

The exercise uses both techniques: `DroneFlight` was built by *composition* in the previous lecture, and these two subclasses extend it by *inheritance*. The two techniques are not alternatives at the level of a program — a `SurveyDrone` **is a** `DroneFlight`, which **has a** `Camera`.

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

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: 2025-10-07-uml[16] (Week 10), which gives this hierarchy a formal notation, and 2025-10-07-advanced-inheritance[17], which takes `super()` further.

Reference material

CSSE1001 Diagram Notation

This note records the drawing conventions **CSSE1001 actually uses**, inferred from its own lecture slides. It is the reference that later diagrams in this course should be checked against.

> **!** This course ships no notation guide
>
> There is **no written style or notation standard** anywhere in the CSSE1001 materials. The course profile lists no prescribed textbook and no drawing guide; the "Learning resources" section points only to the UQ Library, Blackboard and Ed Lessons. Every convention below is **inferred from the drawings themselves**, not quoted from a rule. The three exceptions — places where the slides *state* a rule in words rather than merely demonstrate one — are the naming conventions (see Naming and capitalisation) and the one figure-source note in 12b. Everything else is observation.

How to read the citations

Decks are cited by their lecture code and **PDF page number**, e.g. `10a p7` = the UML deck, seventh page of the PDF.

Do not cite the printed footer numbers. They are unreliable: 09a runs `1/14 … 4/39, 5/39, 6/14 … 15/14, 16/15` and then ends on a page footed `14/14` that is actually PDF page 17. Several decks have footer totals that disagree with their own length (11a's last page reads `14/13`). PDF page index is the only stable address.

[16] courses/csse1001/2025-10-07-uml.pdf
[17] courses/csse1001/2025-10-07-advanced-inheritance.pdf

Colours were sampled from the rendered pages, quantised to 16 levels per channel, so hex values are accurate to roughly ±8 per channel.

What I sampled

248 pages read as images, out of 726 across 25 decks — 17 of the 25 decks. Full text was extracted from all 25 decks plus the course profile, and every deck was scanned for embedded raster images, so the unread decks were checked for figures even where I did not read them.

Deck	Pages read
01a history-of-computing	22–33
02a python-memory-model	4–28
03a functions	23–28
03b sequence-selection-and-iteration	2–19
04b non-primitive-data	2–17
07a exceptions	14–21
07b scope	5–19
07c object-oriented-programming	2–19
08a dunder-methods	2–5
08b representation-invariants	5–14
09a composition	1–17 (all)
09b inheritance	1–19 (all)
10a uml	1–14 (all)
10b advanced-inheritance	1–20
11a model-view-controller	1–14 (all)
12a recursion	2–15
12b complexity-and-sorting	2–19

Not read as images: 00-welcome, 01b python-as-a-calculator, 02b primitive-data, 04a while-loops, 05a for-loops, 05b comprehensions, 06a string-methods, 06b file-io. An image-extraction scan across all decks found **zero embedded raster images** in seven of those eight (00-welcome has six photo pages), and they are typeset in the same LaTeX Beamer template as the decks I did sample. So the risk that they hide an unseen diagram idiom is low but **not zero** — the eight decks above are the blind spot in this note.

The two slide families

Almost every visual inconsistency in this course traces back to one fact: the decks come from **two different production pipelines**, and many decks mix both on adjacent pages.

LaTeX Beamer family — serif body text (Computer Modern), ruled code listings with line numbers, coloured named block headings, no raster images. Covers 01b, 02a, 02b, 03a, 03b, 04a, 04b, 05a, 05b, 06a, 06b, and most of 07a/07b/12a/12b.

PowerPoint family — sans-serif labels, filled colour boxes, drop shadows, stock photography, dark-theme code screenshots. Every deck carrying `Based on slides from Dr. Azadeh Ghari-Neiat` on its title page (07a, 07b, 07c, 08a, 08b, 09a, 09b, 10a, 10b, 11a) contains PowerPoint-family pages.

Consistency: this split holds across the whole semester and is the single most useful thing to know before drawing anything for this course.

Colour palette

These five colours carry meaning and recur across many decks. Everything else is decorative.

Colour	Sampled	Means	Evidence
Dark blue	`#3030B0`	Slide title	10a p4, p7, p10; 09a p12; 10b p10, p19
Magenta	`#E00080`	Defined term, emphasis, the answer	10a p4; 02a p6; 12a p2
Purple	`#92278F`	Named block heading; margin annotation	03b p4; 12b p19; 09a p12; 10b p13
Green	`#109050` (LaTeX) / `#00B050` (PPT)	A prompt to the student	02a p5; 03b p11; 10a p12; 11a p8
Red	`#F00000`	Warning / failure path	12a p10; 07b p13; 04b p14

Purple `#7030A0` is a different purple. It is used only for the UML legend boxes (10a p7, p8, p10, p11) and for the `Warning` heading in 07b p6. Do not conflate it with the block-heading purple.

Consistency: the blue/magenta/purple/green assignments hold from week 2 (02a) to week 12 (12b). Red is the least consistent — see Inconsistencies.

Named block headings

LaTeX-family slides open a block with a coloured heading on its own line, no box and no rule.

- **Green** — a prompt the student is meant to act on: `Question` (02a p5, 12b p3, p9, p12), `Exercise` (03b p11, p14; 12b p5, p8, p11, p15; 09b p17; 10a p12; 11a p8), `Example` (02a p22, p24, p25), `Task` (07b p17).
- **Magenta italic** — the resolution: `Answer` (02a p6, p7, p8, p9, p23; 12b p18).
- **Purple** — a formal or named block: `Definition (…)` (03b p4, p6, p7, p12; 04b p3; 07c p5; 12b p19), `Proposition` (12a p13), `Base Case` / `Induction Hypothesis` (12a p14), `Observe` (03b p10), `Summary` / `Next` (02a p28, 07a p21, 07b p19), `Constants` (07b p5), and named algorithms `Insertion Sort` / `Bubble Sort` / `Merge Sort` / `Quick Sort` (12b p3, p6, p9, p12, p16).

`Definition` blocks are parenthesised with the term being defined: `Definition (Boolean domain)`, `Definition (Predicate)`, `Definition (List)`, `Definition (Complexity)`.

Consistency: green/magenta/purple hold across at least five decks spanning weeks 2–12. The `Warning` heading is the exception.

Section divider slides

A section break is a slide bearing **one large magenta italic serif word**, left-of-centre and high on the page, with a short parenthetical gloss beneath it in small black roman type.

- `Variables` — *(Named memory locations.)* — 02a p14
- `Sequencing` — *(Order of instructions matters.)* — 02a p21
- `Selection` — *(Controlling program flow.)* — 03b p2
- `Lists` — *(Mutable Ordered Collections.)* — 04b p2
- `Memes` — *(Recursions through meme examples.)* — 12a p3

When a deck runs a numbered series of dividers, they take **Roman numerals with a colon**, and the series continues **across decks**:

- `I: Structure` — *(Holding named attributes.)* — 07c p8
- `II: Methods` — *(Object scoped functions.)* — 07c p15
- `III: Magic` — *(Overloading built-in functions.)* — 08a p5

Consistency: strong — six plain dividers across four decks, plus a three-part numbered series spanning two decks. One divider (08b p11, `Exercise`) carries no gloss.

Code listings

Code is by far the most frequently drawn thing in this course, and its presentation is a notation in its own right.

Ruled block. A listing is bounded by a **horizontal rule above and below**, full text width, with no vertical rules and no background fill. Line numbers run in tiny grey type in the left margin. Code is monospace with visible letter-spacing.

REPL vs. file. The two are distinguished by prompts, not by styling:

- A **REPL transcript** uses a magenta `>>>` prompt for the first line of an input and a magenta `...` for continuation lines; output follows on its own line with **no prompt** (02a p13, p15–p20; 03b p5, p8, p13, p15–p18; 04b p4–p13; 07b p5–p16; 10b p9, p11, p15, p18).
- A **file or definition** is shown with line numbers but **no prompts at all** (04b p15, p16, p17; 08b p6, p9, p13, p14; 09a p10–p14).

Colour inside code. Black is the default. Magenta marks the token or line under discussion — class names being introduced (`Battery`, `Camera`, `Motor`, 09a p10, p11), the lines inherited or overridden (10b p10, p13, p17), the `assert` keyword (08b p6, p9), the constant being referenced (08b p13, p14). Boolean operators are coloured by identity: `and` in cyan, `or` in green, held consistently through 03b p9, p10, p11, p14.

Vertical ellipsis. Elided code is a centred on its own line (03a p23) or three stacked full stops (07a p15).

Placeholder metasyntax. Things the student must substitute are written in angle brackets in monospace: `<code>` and `ExceptionName_k` (07a p15, p16), `<object_variable>.<method>(<parameters>)` (07c p16), `<statement that should be true>` (08b p5). *Consistency: three decks.*

Annotating code

This is the course's real "diagram" vocabulary — it annotates listings far more often than it draws figures.

Right-margin note. A short **purple italic serif** phrase, right-aligned, on the same line as the code it comments on. Sampled `#92278F` in both 09a p12 and 10b p13 — identical colour, three decks apart. (Measured by histogram at 300 dpi. The neighbouring aside on 09a p13 is `#92268F`, one unit off in green and blue — the same ink to the eye, but the two pages are not bit-identical.)

Examples: *inherit from A*, *override*, *extend from B from A*, *Super is B* (10b p10, p13, p14, p17, p18); *Drone has-a Battery* (09a p12); *this is the memory location where 42 is stored* (02a p13); *Note addition happened first* (03b p5); *ys is an alias of xs*, *Aliasing relationship broken!* (04b p13); *a 'private' variable*, *this is called a 'getter'* (07c p19); *unreachable if line 5 errors* (07a p14).

Consistency: the strongest single convention in the course — present in 02a, 03b, 04b, 07a, 07b, 07c, 09a, 10b. Weeks 2 through 10.

Boxed token. A thin rectangle drawn around an identifier *inside* the listing, marking the thing being named. The box colour is **not fixed**: purple around `Point()` and `self` (07c p9, p16), magenta around `raise ZeroDivisionError` (07a p18), blue around `Child` and purple around `Parent` (09b p9). See Inconsistencies.

Brace grouping. A magenta right-brace `}` spanning consecutive lines, with a magenta italic label beside it — `Attributes` beside the two `self.x = 0` / `self.y = 0` lines (07c p9). *Single instance.*

Pointer arrow. A straight or gently curved arrow from an annotation to the code element it refers to. Magenta in 07c p12 and p16; orange in 09b p12 and p13; blue and purple in 09b p9.

Colour keying. Where an annotation has a coloured box and a coloured arrow and a coloured word in the caption, all three share one colour: on 09b p9 the blue box round `Child`, the blue arrow, and the word "Child" in the caption are all blue, and `Parent` is purple throughout. *Two instances (09b p9, p12); treat as a good practice rather than a settled rule.*

Focus markers. Small **green right-pointing triangles** in the far-left margin flag the lines the reader should look at (07b p7, p12). *Two instances, one deck.*

Scope boxes

The clearest original notation in the course, in 07b.

A **rounded rectangle drawn over a span of code lines** delimits a scope, and its colour names the scope kind:

- **Purple outline, no fill** — global scope; encloses the whole snippet (07b p7, p12).
- **Blue outline, pale blue fill** — local/function scope; encloses the `def` body only (07b p7, p9, p12).

Nesting is literal: on 07b p7 and p12 the blue local box sits **inside** the purple global box.

Each box is paired with a short horizontal arrow leading out to a **colour-matched label** — purple arrow to purple `global variable`, blue arrow to blue `local variable` (07b p7, p8, p9, p12).

Consistency: four pages of one deck, entirely self-consistent. Never reused elsewhere in the semester.

UML class diagrams

The formal UML vocabulary is taught in **10a** and reused once in **10b p19**.

Class box. A black-outlined rectangle in **three compartments**, separated by full-width horizontal rules: name (centred, serif) / attributes / methods (both left-aligned). 10a p6 states the rule in words and shows it twice.

Visibility prefixes. Stated explicitly on 10a p6: `-` private, `+` public, `#` protected. Applied consistently on 10a p7–p11: `#name: str`, `-breed: str`, `+speak()`.

Attribute typing. `name: type` with a Python type — `#age: int`, `-nationality: str`, `-pets: []` (10a p6, p9, p11). Methods show a return type only sometimes: `+info(): str` (10a p6) but bare `+info()` (10a p7–p11).

Empty compartments are kept. `Cat` on 10a p8–p11 keeps a visibly empty attributes compartment rather than collapsing to two compartments.

Single-compartment boxes are acceptable when only structure is being shown — 10a p4 and 10b p19 draw name-only boxes.

Relationship symbols

Relationship	Symbol	Evidence
Inheritance	Solid line, **solid filled** triangular arrowhead, pointing **up into the parent**	10a p4, p7–p11; 10b p19
Association	Plain solid line, no head, optional verb label at one end	10a p8 (`drinks`), p9 (`Associated`)
Composition	Solid line, **filled diamond at the "whole" end**	10a p10
Aggregation	Solid line, **hollow (white-filled) diamond at the "whole" end**	10a p11

> ⚠ The inheritance arrowhead is filled, not hollow
>
> Standard UML draws generalisation as a **hollow** triangle. This course draws it **solid filled**, verified at 600 dpi on 10a p7 and again on 10b p19. Every inheritance arrow in the course is filled. If you conform to standard UML you will not match the slides; if you match the slides you will not match standard UML. **Follow the course.**

Line routing is orthogonal. All relationship lines use right-angle elbows, never diagonals (10a p8–p11). A parent with several children gets a single vertical stem down from the parent to a horizontal bus, with vertical drops to each child (10a p4, p7–p11; 10b p19).

Layout and flow. Parent above, children in a row beneath. A grandchild hangs directly below its own parent with its own vertical arrow (10b p19: `C` below `B`). Associated non-hierarchy classes sit to the **side** at the same height as the class they attach to (10a p9 `Vet`, p10 `Heart`, p11 `Owner`).

Multiplicity. Placed **above the line, adjacent to the end it qualifies**, in magenta — `1..*` at both ends (10a p9), `1` and `1` (10a p10), `0..*` (10a p11). 10a p9 gives the vocabulary in a legend box: `0..1` zero to one, `n` specific number, `0..*` zero to many, `1..*` one to many, `m..n` specific range.

Magenta marks the relationship being taught. On each of 10a p7–p11 the relationship introduced on that slide is drawn in magenta while previously-established structure reverts to black. On p7 the inheritance tree is magenta; by p8 it is black and the new association line is magenta. *Consistency: holds on all five relationship slides — a deliberate, reusable device.*

Legend box. A rounded rectangle with a **purple `#7030A0`** outline, containing the relationship name in purple and the bare symbol drawn beneath it, parked in the top-left corner (10a p7 Inheritance, p8 Association, p10 Composition, p11 Aggregation). The multiplicity legend on 10a p9 uses the same box for a text list instead of a symbol.

Non-UML class and object diagrams

Before UML is taught in week 10, the course draws classes at least **four other ways**. These are not variants of one style — they are four unrelated idioms.

(a) Uniform blue hierarchy — flat steel-blue `#4F81BD` filled rectangles, white sans-serif labels, parent above and children in a row, joined by a **`#3D6696`** orthogonal bracket with **no arrowhead at all** — a darker shade of the box fill, not black, though it reads as black at slide size. Photos of instances sit below each child. (09b p2, p3)

(b) Multi-colour hierarchy — parent box **black** with white text; each child a different Office-palette fill — and these are the stock Office 2007 accents exactly, which is why they recur: `#C0504D` brick red, `#4BACC6` teal, `#F79646` orange, `#215968` dark teal, olive. Connectors are thin **`#4A7EBB` straight lines** — diagonal from the outer children, vertical from the one directly beneath — and each ends in **its own arrowhead**. On 09b p5 three arrowheads overlap on `Animal`'s bottom edge; at slide size they read as one, but there are three. (09b p4, p5, p6)

Black for the root holds in all three of these diagrams (`Employee`, `Animal`, `Person`). The child colours carry no meaning I can find — `Dog` stays teal on 09b p5 even though it is itself a parent.

(c) Attributes-over-methods box — the class name in **purple bold sans-serif above** two stacked **magenta-outlined** boxes: attributes on top, methods beneath, with a photo of an instance alongside. (07c p6, three times: `Dog`, `Student`, `Smartphone`)

(d) Attributes-left / methods-right — the class in a black-outlined box in the centre with its name in black and `(class)` beneath in magenta; a **purple-outlined** attributes box to the **left** and a **magenta-outlined** methods box to the **right**, each with a plain black label above it. Instances are photographs **below**, reached by straight black **downward** arrows with solid heads. (08a p2)

> **i** Instantiation arrows point down; inheritance arrows point up
>
> 08a p2 draws class → instance as a **downward** arrow. Every inheritance arrow in the course points **up** to the parent. The two are never shown on the same page, so the contrast is safe — but keep it.

(e) Containment ("has-a") boxes — composition is drawn as **nesting, not as a line**. A large steel-blue `#4080B0` filled rectangle is the whole, labelled in large white text at its right edge (`car`, `drone`); a smaller brick-red `#C05040` rounded rectangle sits **inside** it and is the part, labelled in white at its top with its methods listed beneath (`engine` / `stop()` / `start()`). (09a p6, p16)

Substitution is shown by stacking two such figures and joining them with a **thick curved orange `#F79646` arrow** from the old configuration to the new (09a p16, "Hot-Swapping").

Consistency: (e) holds twice in one deck and is the course's only purpose-built composition picture. (a)–(d) are one or two pages each.

The large worked example

10a p13 — the only whole-system class diagram in the course — is drawn in a completely different style from the rest of 10a:

- Single-compartment boxes, name only, **pastel fills with matching darker strokes**.
- Sampled fills `#D0E0F0` blue / `#D0E0D0` green / `#F0E0C0` orange, with strokes `#6080B0` / `#80B060` / `#D09000`. These are the **draw.io / diagrams.net default shape colours**, matched exactly — this figure was built in draw.io with stock styling, not drawn to match the deck.
- **Dashed lines with small open arrowheads** for association/dependency (`WTController` → `WTView`, `WTModel` → `Tank`), **solid lines with the usual filled triangle** for inheritance.
- Tiny `1` / `N` multiplicities at the line ends.

- Green fill appears on `Tank`, `Tile` and `Enemy` — the three classes that have subclasses. **This is my inference from one figure**, and `Battlefield` (blue, no subclasses) is consistent with it while `WTView` (orange, alone) is not explained. Do not rely on it.

Reference and memory diagrams

The course does not draw its own reference diagrams. It uses Python Tutor.

`pythontutor.com` output is embedded as a screenshot in 07b p7, p12 and 07c p13, p14, and named as a live-demo cue five times in 02a (p22, p24, p25, p26, p27, captioned `Python Tutor 1.` through `Python Tutor 5.`).

The Python Tutor rendering, as it appears in those screenshots:

- Two columns under bold headers **`Frames`** (left) and **`Objects`** (right).
- A frame is a **pale blue box** with a bold title (`Global frame`) and one row per name; each row is `name` right-aligned against a value cell.
- An object is a **pale yellow box** with a small red-brown type caption floating above it — `Point class`, `Point instance`, `function`.
- An attribute inside an instance is a two-cell row: grey name cell, white value cell.
- A reference is a **curved blue-grey arrow** from a frame slot to the object.
- **Aliasing is shown by two arrows converging on one object** (07c p14), versus two arrows to two separate objects for equal-but-distinct values (07c p13).

Consistency: two decks, four screenshots, plus five named references in a third deck spanning week 2 to week 7. Treat Python Tutor as this course's canonical memory picture.

Bit and memory notation

Established in 02a and legended explicitly on p4.

- **A bit is a square cell. Filled black = on/1. White with a black outline = off/0.** 02a p4 draws the two side by side and labels them `On` and `Off`.
- Held on 02a p6, p7, p8, p9, p10, p11, p12 — seven further pages.
- **A bit string is a horizontal row** of contiguous squares, read left to right, with `. . .` after the last cell for continuation (02a p10, p11).
- **Memory is rows of words stacked vertically**, each row prefixed in the left margin by its **address as a small decimal number** (02a p12: 381–388).
- Enumerating all n-bit patterns stacks one row per pattern (02a p7, p8, p9).

Consistency: very strong within 02a. Never used again — no later deck draws a bit.

Data-structure and algorithm-trace diagrams

> **i** The one explicit figure-source statement in the course
>
> 12b p2: *"Note: The diagrams from today's lecture are taken from* ***runestone.academy.***" This is the closest thing CSSE1001 has to a textbook lineage, and it governs **every sorting and merge-tree figure in 12b** (p4, p7, p10, p13, p14). Those figures are Runestone Academy's, not the lecturer's, which is why their visual language matches nothing else in the course.

List cells. A list is a horizontal row of adjacent square cells with a thin grey border and a soft drop shadow, one integer per cell in small sans-serif, centred. (12b p4, p7, p10, p13, p14)

Shading marks state. White = untouched; grey fills mark the cells the current step is acting on or the region already sorted. The **exact meaning of each grey level is never legended** — see Not established.

A trace runs top to bottom. Each row is one step; time advances downward. (12b p4, p7, p10)

Step captions sit to the right of each row, small black sans-serif, in the present tense of what just happened: `93 is largest`, `inserted 26`, `Exchange` / `No Exchange`, `91 in place after first pass`. (12b p4, p7, p10)

Movement arrows are thin black orthogonal arrows drawn *above* the row, running from source cell to destination cell (12b p4). A whole pass gets a long labelled arrow spanning the figure above everything else — `First pass` (12b p10).

Recursion trees (merge sort, 12b p13–p14) are drawn as binary trees of list-rows:

- `Merge Sort - Winding` (p13): the full list at the top, splitting downward, arrows pointing **down** from a list to its two halves.
- `Merge Sort - Unwinding` (p14): the singletons at the top, arrows pointing **down** into each merged result, ending in the sorted list at the bottom.

Both use downward arrows. The recursion direction is carried by the slide title and by which end of the tree holds the whole list — *not* by arrow direction. This is easy to get wrong.

Mathematical and logical notation

Truth tables (03b p6, p7). A 3×3 grid with visible black rules; the **operator name sits in the top-left cell in cyan monospace** (`and`, `or`); the header row and header column carry the operand values in black monospace; a **double rule** separates the headers from the body. The one semantically interesting result is coloured magenta — the single `True` for `and` (p6), the single `False` for `or` (p7).

Function signatures are given above the table, centred, mixing monospace names with maths: `and : × →` . Also `> : × →` (03b p5) and `P : → {True, False}` (12a p11).

Blackboard bold for domains — booleans (03b p4, p5, p6, p7), integers (03b p5), naturals (12a p11, p13). `= {True, False}` is defined with the Python literals inside the set, in monospace (03b p4).

Python literals stay in monospace inside maths. `True` and `False` are never italicised into maths variables.

Implication and quantifiers use and : `(P(0) and (P(n) P(succ(n)))) m , P(m)` (12a p11). Note `and` stays as the Python keyword in monospace even inside a mathematical statement.

Contradiction is (02a p20).

Displayed maths is centred on its own line, in magenta when it is the point being made (02a p20, 12a p11).

Arrow chains stand in for flow diagrams. Simple pipelines are written inline as text with → rather than drawn: `state change → _check_invariants → valid` (08b p8), `parent → child → grandchild` (10b p12), `Task + TodoList → Model (domain)` (11a p8). *Consistency: three decks. This is why the course has so few real flow diagrams.*

Geometric and proof figures

The chessboard-tiling figures for the induction proof (12a p12, p14, p15):

- The construction is drawn in **cyan solid lines**.
- The underlying reference grid is **grey dashed**.
- The distinguished (uncovered) square is a **filled magenta disc**.
- Dimensions are annotated with cyan extension lines outside the figure, labelled in black serif maths (`2 , 2 ¹`), placed below and to the right.

Consistency: three pages, one deck, entirely self-consistent.

The one flow diagram

07a p17 is the **only control-flow diagram in the 726 pages**. It shows try/except/else/finally:

- Nodes are **rounded rectangles with a hand-drawn, wobbly grey stroke** (a sketch/xkcd style).
- The node label is the bare Python keyword in monospace, **coloured per keyword**: `try` teal, `except` magenta, `else` blue, `finally` purple.

- **`finally` is drawn with a dashed sketchy border**; the other three are solid.
- Flow runs **top to bottom**, with curved grey arrows looping out to the left.
- Edge captions are bold sans-serif, placed **below** the arrow they describe: *Run the code under try*, *Execute the code under except when there is an exception*, *No exceptions? run the code under else after try*, *Always run this code.*
- **Status glyphs sit outside the flow**: a purple circled beside the try→except path, a green circled beside the try→else path, a blue circled **i** beside `finally`.

> ⚠ Warning
>
> This is a **single instance**. It does not establish a general flowchart convention for the course — see Not established.

The LEGB diagram

07b p13 draws name resolution as a vertical cascade:

- Four levels listed top to bottom, each with its **initial letter bold** — **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.
- Between consecutive levels, a large **red hook arrow** with the small red caption `If not found` to its left.
- The cascade terminates in bold red `Error!`.

Red marks the failure path. *Single instance.*

Naming and capitalisation

The three rules the course states in words, not just in examples:

Kind	Rule	Evidence
Variables, functions	lowercase, words separated by underscores — `descriptive_variable_name`	03a p25
Classes	`CamelCaps` — "That is `ClassNamesAreLikeThis`"	07c p7
Constants	`SNAKE_CASE_CAPS`	07b p5

03a p24 states the course follows **Google's PEP**, "a variant of PEP8", with both linked as lavender rounded chips. 03a p26: lines must be **strictly less than 80 characters**.

These rules govern labels in diagrams too. UML boxes use `CamelCaps` class names (`DeliveryRobot`, `WTController`, `DroneFlight`) and snake_case members (`_check_invariants()`, `has_solar_panels`, `battery_level`) throughout 10a, 10b, 07c and 08a.

Do/don't pairs. Compared alternatives go in a **two-column table headed `Yes` and `No` in magenta italic**, with a single rule under the headers and no other lines — good on the left, bad on the right (03a p25, p28). The variant form labels two separate ruled listings *Not helpful* / *Helpful* in magenta italic (03a p27). Inside dark-theme code screenshots the same judgement is made with an all-caps comment — `# BAD!` / `# GOOD!` (08b p7).

Figure sourcing and attribution

No textbook is cited as a figure source anywhere. A full-text search of all 25 decks and the course profile for textbook, figure-source and adaptation language returns only the four items below. The course profile prescribes no textbook at all.

Attribution	Where	Scope
`Based on slides from Dr. Azadeh Ghari-Neiat`	Title page of 07a, 07b, 07c, 08a, 08b, 09a, 09b, 10a, 10b, 11a	Slide lineage, not figure lineage
`Based on slides from Dr. Azadeh Ghari-Neiat and Dr. Paul Vrbik`	07b title page	as above
`Note: The diagrams from today's lecture are taken from runestone.academy.`	12b p2	**All sorting figures in 12b**
`Image source: www.freepik.com`	09a p9	That slide's stock art
`Images are from WIRED`	09b p2, p3	Those slides' photos
`Inspired by the example of MIT6.100L slide.`	07c p10	The blueprint/house analogy — an idea, not a figure family

Placement. Credits are set in **tiny centred or bottom-right roman type** beneath the figure (09a p9, 07c p10) or in the bottom-right corner of the slide (09b p2, p3).

10b p9 refers to "the course notes" as the source of a code snippet — the course has its own written notes (delivered via Ed Lessons) which are **not** among these materials and may carry conventions this note cannot see.

Page furniture

- **Slide titles**: dark blue `#3030B0`, serif, upper-left, sentence case with title-case content words. `Optional Content:` is used as a prefix for non-examinable material (10b p5).
- **Footer**: `n / N` in tiny grey at the bottom-right. **Unreliable** — see How to read the citations.
- **Title slide**: course code and title in blue, lecture number and name in magenta italic, lecturer, date, then the attribution line. (10a p1, 09a p1, 09b p1, 10b p1, 11a p1)
- **QR-code slides**: a large centred QR code with the URL in small serif beneath, flanked by decorative 3D clip art, no title (07a p19, 07b p18, 09b p19).

Inconsistencies

Places where the course does the same thing more than one way. **Both readings are recorded; a decision is needed before drawing.**

1. Two entirely different UML idioms inside one deck

10a p4–p11 draws three-compartment black-outlined class boxes with magenta relationship lines, orthogonal routing and no fills. 10a p13 draws single-compartment pastel-filled draw.io boxes with dashed dependency arrows. Same deck, same lecture, same subject. 10b p19 adds a third variant: single-compartment boxes with **navy `#103040`** outlines. **Decide: compartments or not, fills or not, box outline colour.**

2. Class boxes are drawn four incompatible ways before week 10

Three-compartment black box (10a p6); attributes-box-above-methods-box with the name outside and magenta outlines (07c p6); attributes-left/methods-right around a central black name box (08a p2); plain filled rectangle with name only (09b p2, p4). **Decide which is canonical for pre-UML material.**

3. Inheritance connectors: three styles

Orthogonal bracket with **no arrowhead** (09b p2, p3); **straight blue lines each with its own arrowhead**, overlapping at the parent (09b p4, p5, p6); orthogonal bracket with a **filled triangular arrowhead** (10a p4, p7–p11; 10b p19). **Decide.**

4. Hierarchy box colour: uniform vs. per-child

09b p2/p3 fill every box the same steel blue. 09b p4/p5/p6 make the root black and give each child its own colour. The two appear two pages apart. And 10a p13 uses a third scheme where colour may track "has subclasses". **Decide whether fill colour means anything at all.**

5. `Warning` heading: red or purple

Red `#F00000` in 12a p10 and 04b p14. Purple `#7030A0` in 07b p6. **Decide.**

6. Two greens for the same role

`Question` / `Exercise` / `Example` / `Task` are `#109050` in the LaTeX decks (02a p5, 03b p11) but `#00B050` in the PowerPoint ones (10a p12, 11a p8). Same meaning, two hexes. Additionally 01a p25 sets `Question` in a magenta/crimson rather than either green (observed visually; my colour sample of that page missed the heading). **Decide one green.**

7. Boxed-token highlight colour is unfixed

Purple (07c p9, p16), magenta (07a p18), blue-and-purple keyed to two referents (09b p9). **Decide.**

8. Three code presentations

Ruled + line-numbered LaTeX listings (most decks); **dark-theme screenshots** of an IDE (09b p8–p13, p16; 10b p2, p4, p5; 08b p7); and plain unruled, unnumbered monospace (07a p16, p20). 09b switches between the first two within a single deck. **Decide.**

9. `Summary` styling

Purple block heading (02a p28, 07a p21, 07b p19); blue slide title (10a p14, 11a p14); magenta (09b p18). **Decide.**

10. Image credits are given inconsistently

09a p9 and 09b p2/p3 credit their stock photography. 01a p23–p29 (Ada Lovelace, Grace Hopper, the Jacquard loom, punch cards, the first computer bug), 07c p6/p10, 08a p2/p4 and 12a p4–p9 (the meme sequence) credit nothing. **Decide a credit policy.**

11. Bit-string reading order

02a p7–p9 enumerate n-bit patterns by toggling the **leftmost** cell fastest (`off-off`, `on-off`, `off-on`, `on-on`), which reads the leftmost cell as least significant. 02a p11 then interprets a word as $1\cdot2^{3\,2} + 1\cdot2^{3\,1} + \ldots + 1\cdot2$, reading the **leftmost cell as most significant**. **If you draw a numbered bit string, decide which end is the high bit.**

12. `Exercise` as heading vs. divider

A green block heading in 03b p11, 12b p5, 09b p17, 10a p12, 11a p8 — but a full magenta italic divider slide in 08b p11. **Decide.**

Not established by the source

Conventions a diagram in this course will plausibly need, that the materials simply do not settle. This is where the risk of invention lies.

No general flowchart convention. The only flow diagram in 726 pages is 07a p17's sketch-style try/except figure. There is no established shape vocabulary (no decision diamond, no terminator, no process box), no arrow style, and no rule for labelling branches. **If you need a flowchart, you are inventing one.** Note that the course's habit is to write flow as an inline `→` chain (08b p8, 10b p12, 11a p8) rather than draw it — consider doing that instead.

No MVC diagram. The entire MVC deck (11a, 14 pages) contains **no architecture diagram**. The Model/View/Controller relationship is given as a six-item numbered list (11a p6, p13) and a four-column table (11a p5). The only place the three appear as boxes is 10a p13, where they are three unremarkable draw.io rectangles among thirteen. There is **no established shape, layout, arrow direction, or colour** for an MVC diagram in this course.

No hand-drawn reference or memory diagram. The course never draws its own variable/object box-and-arrow picture — it screenshots Python Tutor (07b p7/p12, 07c p13/p14). If you need a reference diagram that Python Tutor cannot produce, there is no house style to follow.

No list or aliasing diagram. 04b (non-primitive data, 45 pages) teaches aliasing, copying and pass-by-reference **entirely through REPL transcripts** — mutating `ys` and observing `xs` change (p10, p12, p13). Not one box-and-arrow picture of a list. The idea of "two names, one list" is never drawn outside Python Tutor.

Grey-level semantics in the sorting traces are never legended. 12b p4, p7, p10 use at least three shades — white, light grey, darker grey — and no slide says which is "being compared", which is "sorted", and which is "just moved". Since these figures are Runestone's, the course has no stated rule to appeal to.

No stated meaning for fill colour in class boxes. 09b p5's brick red, teal and orange appear decorative; 10a p13's green *may* mark classes with subclasses but this is one figure and `WTView`'s orange contradicts any simple reading. **There is no colour key anywhere.**

No convention for showing a method body, call stack, or sequence of calls. Despite 12a (recursion, 47 pages) there is no call-tree or stack-frame diagram of the course's own making — recursion is taught with memes (p4–p9), induction on chessboards (p12–p15) and code. No UML sequence diagram, activity diagram, state diagram or object diagram appears anywhere in the course.

No rule for typing method returns in UML. 10a p6 writes `+info(): str`; 10a p7–p11 write `+info()`. Both are the same method on the same class in the same deck.

No hex values are ever published. Every colour in this note was sampled from the rendered PDFs. The course states no palette, so a redrawn figure matching these hexes matches the slides by measurement, not by declaration.

Abstract classes have no notation. 10b p3–p5 teaches abstract base classes in prose and code only. UML's italic-name convention for abstract classes is never used; `Shape` is never drawn.

Line weights, font sizes and box dimensions are never specified, and vary between the two slide families.

The eight unread decks. 00-welcome, 01b, 02b, 04a, 05a, 05b, 06a, 06b were not read as images. They contain no embedded raster images (except 00-welcome's six photo pages) and are in the LaTeX family, so they are unlikely to hide a new idiom — but this note cannot rule it out.

The written course notes. 10b p9 cites "the course notes" as a source. Those notes are delivered through Ed Lessons and are not part of these materials. If they contain diagrams, they may establish conventions that contradict this note.

See also

- 2025-10-07-uml[18] — Lecture 10A's content
- uml[19] — the UML reference note
- python-pep8-style-guide[20] — the naming rules cited above
- csse1001[21] — course index

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`).

[18] courses/csse1001/2025-10-07-uml.pdf
[19] courses/csse1001/uml.pdf
[20] courses/csse1001/python-pep8-style-guide.pdf
[21] courses/csse1001/csse1001.pdf

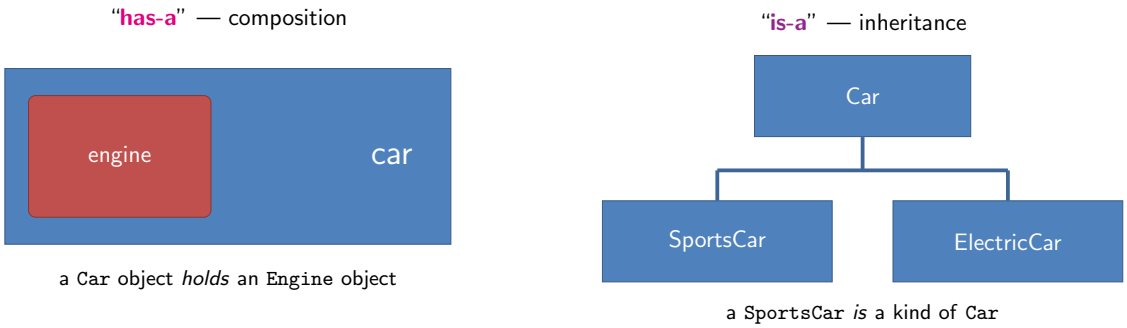

Figure 7: The two ways of reusing a class, side by side. On the left, composition is **containment**: the part sits inside the whole, and there is no connecting line. On the right, inheritance is a **hierarchy**: parent above, children below, joined by a stem-and-bus connector with no arrowhead. *has-a* 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[22] 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.

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

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):
        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[23] and 2025-09-23-inheritance[24].

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`.

Python Dunder (Magic) Methods

Dunder ("double underscore") or **magic** methods are special methods, named `__like_this__`, that Python calls automatically to implement built-in behaviour for a type — instantiation, printing, equality, arithmetic operators, and more. You don't call them directly; Python invokes them on your behalf when the corresponding syntax/built-in is used.

See 2025-09-16-dunder-methods[25] for the lecture note, and python-classes-and-objects[26] for classes generally.

Common dunder methods

Method	Called by	Purpose
`__init__(self, ...)`	`ClassName(...)`	Initializes a new instance. Returns `None` — it fills in an object that already exists rather than producing one.
`__str__(self)`	`print(obj)`, `str(obj)`	User-friendly display string.
`__repr__(self)`	the REPL, `repr(obj)`	Unambiguous, developer-facing string — ideally a valid Python expression that recreates the object via `eval()`.
`__eq__(self, other)`	`obj == other`	Custom equality (instead of identity).

[23] courses/csse1001/python-inheritance.pdf
[24] courses/csse1001/2025-09-23-inheritance.pdf
[25] courses/csse1001/2025-09-16-dunder-methods.pdf
[26] courses/csse1001/python-classes-and-objects.pdf

Method	Called by	Purpose
`__add__(self, other)`	`obj + other`	Addition.
`__sub__(self, other)`	`obj - other`	Subtraction.
`__neg__(self)`	`-obj`	Unary negation.
`__len__(self)`	`len(obj)`	Length.

Every binary method takes the left operand as `self` and the right one as `other`.

`__repr__` vs. `__str__`

- `__repr__` is for the programmer: unambiguous, meant for debugging/logging, ideally `eval(repr(obj))` reconstructs an equal object. The round trip is `obj` → `repr()` → source text → `eval()` → a second object, whose `repr` matches the first's.
- `__str__` is for the user: a friendly, readable string, used by `print()`.
- If only `__repr__` is defined, `print()` falls back to it. If neither is defined, printing an object shows its default `<... object at 0x...>` representation.

A consequence that catches people out: at the interactive prompt, `>>> obj` and `>>> print(obj)` can produce completely different text from the same object, because they go through different methods.

Type-hinting a class inside its own body

Methods like `__add__` want to say they return the class being defined, but the class object doesn't exist yet while the body is executing:

```
from __future__ import annotations   # makes annotations lazy (strings)

class Fraction():
    def __add__(self, other) -> Fraction:   # legal because of the import
        ...
```

Without that import, `-> Fraction` is evaluated immediately and raises `NameError`; the alternative is to quote it, `-> "Fraction"`.

Overloadable operators

Binary	Method	Unary	Method	Comparison	Method
`+`	`__add__`	`-`	`__neg__`	`<`	`__lt__`
`-`	`__sub__`	`abs`	`__abs__`	`<=`	`__le__`
`*`	`__mul__`	`~`	`__invert__`	`==`	`__eq__`
`**`	`__pow__`			`!=`	`__ne__`
`//`	`__floordiv__`			`>`	`__gt__`
`/`	`__truediv__`			`>=`	`__ge__`

Operators can be defined in terms of each other rather than from scratch — `Fraction.__sub__` is just `return self + -other`, i.e. `__add__` composed with `__neg__`.

Instance variables vs. class variables

An **instance variable** (`self.x = ...`) belongs to one specific object — each instance has its own copy. A **class variable** (declared directly in the class body) is shared by *all* instances of the class, and can be accessed either via an instance (`obj.class_var`) or via the class itself (`ClassName.class_var`), without needing any instance at all.

```
class Clicker():
    _all_clicks = 0   # class variable -- shared by every instance

    def __init__(self) -> None:
        self._clicks = 0   # instance variable -- unique per object

    def click(self) -> None:
        self._clicks += 1
        Clicker._all_clicks += 1
```

Why `Clicker._all_clicks += 1` and not `self._all_clicks += 1`. `self._all_clicks += 1` expands to `self._all_clicks = self._all_clicks + 1`. The read on the right-hand side finds the *class* attribute (attribute lookup falls back to the class when the instance has no such name), but the assignment on the left always writes an *instance* attribute. So the first click on each object creates a private `_all_clicks` on that object that shadows the shared one forever, and the class-level total stays at 0. Naming the class explicitly is what makes the update land on the shared cell.

Reading through an instance is fine — `c._all_clicks` returns the class's value, which is why the transcript in 2025-09-16-dunder-methods[27] shows `(6, 6, 6)`. It is only *assignment* through an instance that breaks the sharing.

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

Worked example: `Vector2D`

A fuller worked example combining several dunder methods together. The length is Euclidean: for `(2, 3)` it is `sqrt(13)`, `3.605551275463989`.

```
import math

class Vector2D():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __repr__(self):
        return f"Vector2D(x={self.x}, y={self.y})"

    def __str__(self):
        return f"2D Vector: ({self.x}, {self.y}) --- length: {self.length()}"

    def __eq__(self, other):
        return isinstance(other, Vector2D) and self.x == other.x and self.y == other.y

    def __add__(self, other):
        if not isinstance(other, Vector2D):
            return NotImplemented
        return Vector2D(self.x + other.x, self.y + other.y)

    def __len__(self):
        return 2
```

```
>>> u = Vector2D(2, 3)
>>> v = Vector2D(4, 5)
>>> print(u + v)
2D Vector: (6, 8) --- length: 10.0
>>> print(u)          # calls __str__
2D Vector: (2, 3) --- length: 3.605551275463989
>>> repr(u)            # calls __repr__
'Vector2D(x=2, y=3)'
>>> u == v              # calls __eq__
False
```

```
>>> len(u)               # calls __len__
2
```

`print(u + v)` uses two magic methods in one expression: `__add__` builds the new vector, then `print` displays it through `__str__`.

Returning `NotImplemented` (rather than raising or returning `False`) from a method like `__add__` is the standard way to signal "I don't know how to combine these two types" — it lets Python fall back to the other object's reflected method, or raise a clean `TypeError` if nothing handles it.

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[28]'s **"has-a"** relationship.

Syntax

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

# inherit from the Parent class
class Child(Parent):
    # Child class' implementation
```

`Child` inherits everything `Parent` defines, without naming any of it.

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.

Those three, plus plain inheritance, are the four things a subclass can do with any given member:

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

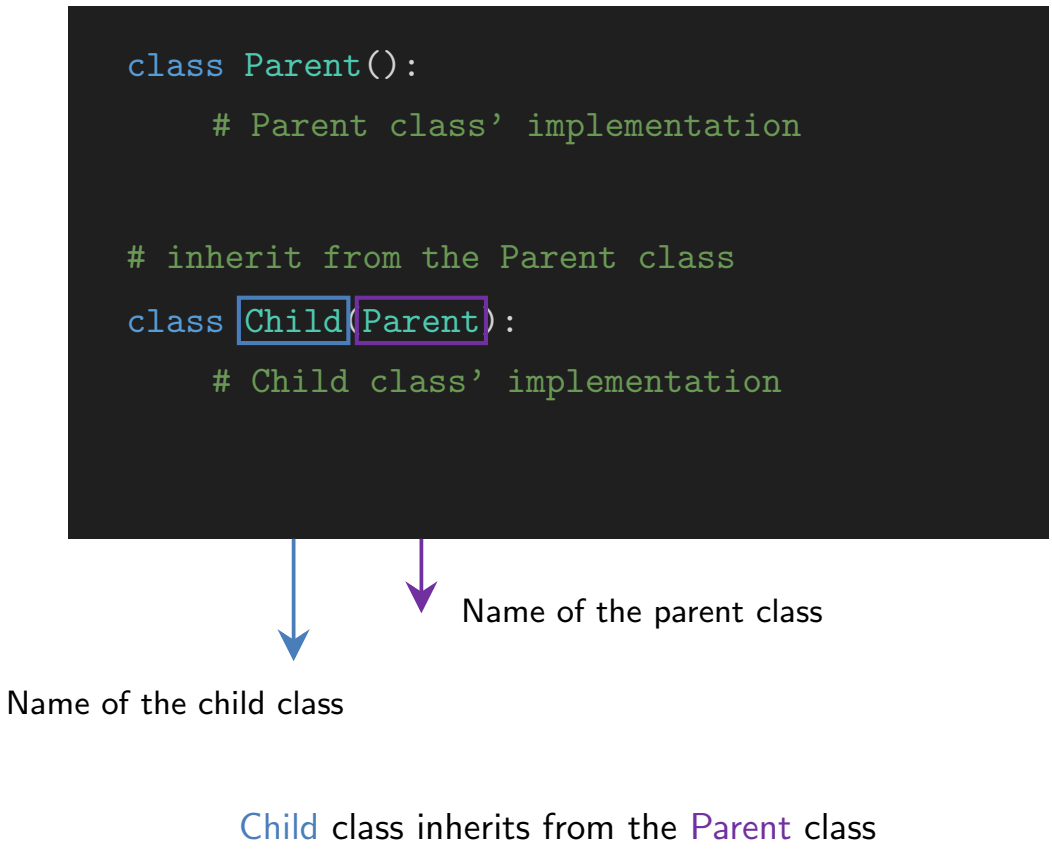

Figure 8: The `class Child(Parent):` line, annotated: the name being *defined* comes first, the name being *inherited from* goes in the parentheses. The box round each name, the arrow leading from it, and the matching word in the sentence beneath are all one colour: blue tracks the child and purple the parent throughout.

The subclass…	…does this	Effect on the parent's version
says nothing about the member	**inherits** it	used unchanged
declares a member the parent doesn't have	**adds** it	none — the parent has no such member
declares a member with the same name	**overrides** it	shadowed; never runs for this class
declares it and calls `super()` inside	**extends** it	runs, plus whatever the subclass adds

A subclass that declares no `__init__` of its own uses the parent's, so constructing it takes the parent's arguments. Declaring one *replaces* the parent's — which is why an extending `__init__` almost always opens with `super().__init__(...)`.

`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.

Which method actually runs is decided by the object, not by the calling code, and is resolved by walking up the hierarchy until the name is found: `object's own class → parent → grandparent → ...`. A subclass that overrides the name stops that walk at itself; one that doesn't lets it continue to the parent. Nothing in the caller has to know or ask which case it is — that is what makes a loop over a mixed list of subclasses work, and why adding another subclass later needs no change to the loop.

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[29] instead. See 2025-09-23-inheritance[30] for the worked `Robot` hierarchy and 2025-09-23-composition[31] for the `DroneFlight` counter-example.

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:

[29] courses/csse1001/python-composition.pdf
[30] courses/csse1001/2025-09-23-inheritance.pdf
[31] courses/csse1001/2025-09-23-composition.pdf

```
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.

Python Representation Invariants

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[32] for the lecture note.

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

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.

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[33] 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.

[33] courses/csse1001/python-exceptions.pdf

- **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[34] 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)
```

This is the aliasing behaviour of python-lists[35] 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[36]).
- 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.

[34] courses/csse1001/2025-09-16-representation-invariants.pdf
[35] courses/csse1001/python-lists.pdf
[36] courses/csse1001/python-testing.pdf