CSSE1001 — Week 11 Notes

Table of contents

Model View Controller **1**
- Advanced inheritance (recap) . . . 1
- MVC . . . 2
- Exercise: a minimal todo list manager . . . 3
- Summary . . . 8

Reference material **9**
- CSSE1001 Diagram Notation . . . 9
- Model-View-Controller (MVC) . . . 26
- Python Classes and Objects . . . 31
- Python Inheritance . . . 33
- Python Lists . . . 38
- Python Representation Invariants . . . 45

Model View Controller

Lecture 11A, 14 October 2025. See csse1001[1] for course logistics and [mvc] for the full reference on the Model-View-Controller pattern.

Advanced inheritance (recap)

MRO complexity for each inheritance model (see python-inheritance[2] for the full detail):

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

Inheritance type	MRO complexity
Single	Simple linear order
Multilevel	Chain-like resolution
Multiple	C3 linearization
Hybrid (e.g. diamond)	C3 linearization

A few extra facts about MRO:

- Every class has an MRO, computed the moment the class is *defined* (not when it's first used) — inspect it with `ClassName.mro()` or `ClassName.__mro__`.
- The MRO is built by the **C3 linearization** algorithm and depends on the inheritance structure.
- If two base classes define the same attribute/method, whichever appears **earlier** in the MRO wins.
- Inside a method of an object of type `MyClass`, every `super()` call — whether it's in `MyClass` itself or in one of its parents/grandparents — refers to the *next class on `MyClass`'s MRO* (i.e. the one after the current class, not the next class up from wherever the method happens to be defined).

MVC

The **Model-View-Controller (MVC)** pattern separates an application into three interconnected components:

- **Model**: stores the data and business rules of the application; responsible for every state change.
- **View**: some (usually visual) representation of the data, without altering it.
- **Controller**: an interface that receives user input (commands), tells the Model what to do, then selects a View to present the result.

There is an asymmetry buried in that third bullet: the Controller is the only component that talks to both of the others. The View is never described as reading from the Model, and the Model is never described as talking to anyone.

Anatomy of MVC

Layer	Core question	Owns	MUST NOT
Model	What *is* the data?	Data & rules	`print`, parse args, call `input()`

Layer	Core question	Owns	MUST NOT
View	How is info *shown*?	Formatting & rendering	change state, validate data
Controller	What happens *next*?	Workflow & commands	store raw data, format output

Model knows nothing about the outside world.

Unpacked against the Model row of the table, that means no printing, no argument parsing and no reading input — pure data and logic. [mvc] draws the dependency shape it implies.

MVC flow

The flow is a six-step cycle:

1. **User** issues a command.
2. **Controller** parses input and calls the appropriate **Model** method.
3. **Model** mutates state and returns domain data.
4. **Controller** selects a **View** and passes the data.
5. **View** formats the data and pushes it to stdout.
6. Control returns to the main loop, waiting for the next user action.

The two things easiest to lose from that list are that step 3 hands data *back to the Controller* — not to the View — and that the View is chosen by the Controller *after* the Model has already run.

Why MVC?

- **Modularity** — swap in new models, views, or controllers without breaking the application.
- **Testability** — each component can be tested in isolation.
- **Separation of concerns** — each component has a distinct responsibility.

Exercise: a minimal todo list manager

Implement a minimal todo list manager with MVC design pattern.

The classes are assigned to layers as follows:

- `Task` + `TodoList` → **Model** (domain)
- `TodoView` → **View** (presentation only)

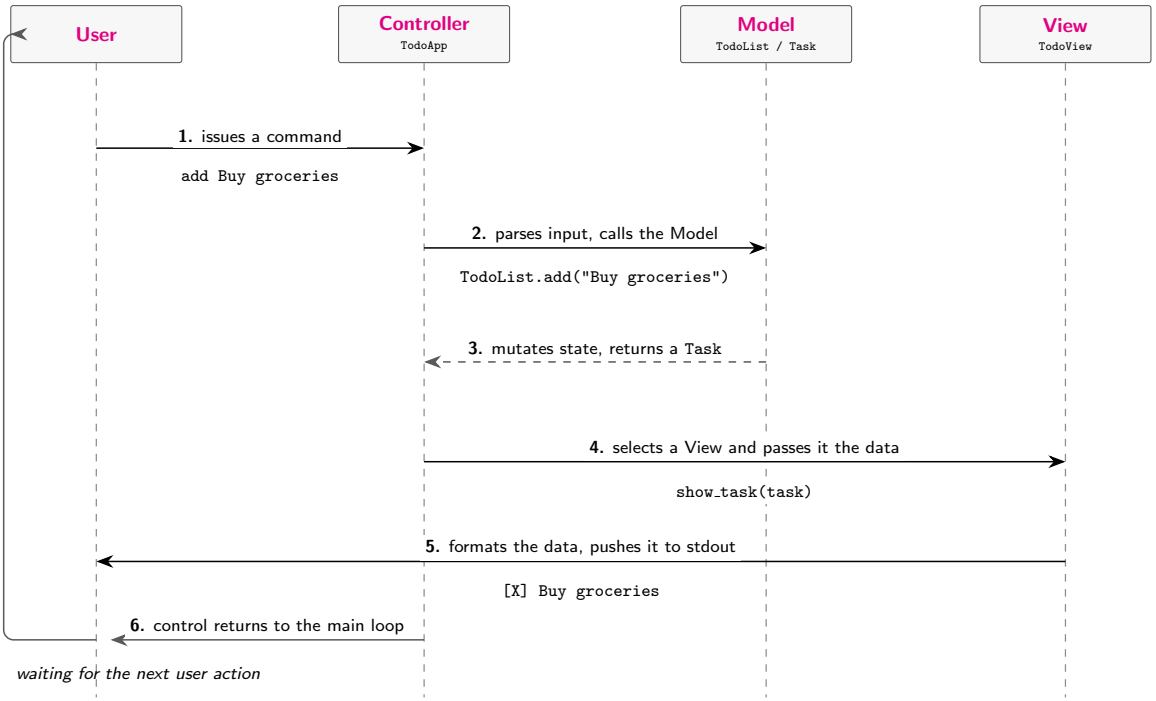

Figure 1: The six-step MVC flow with the todo-list example's names filled in. Reading down is reading forward in time. Step 3 is dashed because it is a return, not a call: the Model hands its domain data back to the Controller, which only then picks a View. The monospace lines beneath arrows 1, 2, 4 and 5 give the concrete values for the command `add Buy groceries`: the call to `TodoList.add`, which returns a `Task`, and then `show_task`. Steps 4 and 5 show the pattern, not a trace of the program: `[X] Buy groceries` is what `show_task` would print if the Controller called it, which for an `add` command the code never does.

- `TodoApp` + run-loop → **Controller** (input parsing & orchestration)

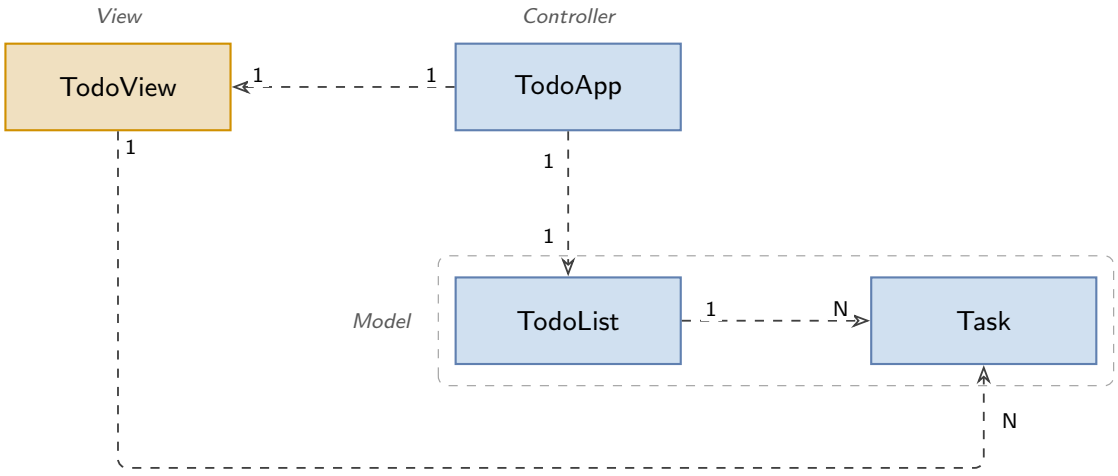

Figure 2: The four exercise classes and the one-way dependencies between them. Each class is a single-compartment box. A dashed line with a small open arrowhead means a class uses another class, with a bare multiplicity at each end. The Controller and Model classes are filled blue and the View class orange. The dashed boundary groups the two Model classes, `TodoList` and `Task`, and the three italic names label the layers. Every dependency shown is read off the constructor and method bodies of the listings below.

The direction of every dashed arrow matters. `TodoApp` knows about `TodoList` and `TodoView`; `TodoList` knows about `Task`; and `TodoView` knows about `Task` too, because `show_task(self, task: Task)` names the type and reads two of its attributes. Nothing points back the other way — no arrow from `TodoView` to `TodoList`, and none from either Model class to the Controller or the View. That one-way shape is the whole point of the *Anatomy of MVC* table, and it is drawn out in [mvc].

The four listings below are one program, split by layer.

Model

```
class Task:
    def __init__(self, title: str) -> None:
        self._title = title
        self._done = False

class TodoList:
    def __init__(self) -> None:
```

```
        self._tasks = []

    def add(self, title: str) -> Task:
        task = Task(title)
        self._tasks.append(task)
        return task

    def all(self) -> list[Task]:
        return list(self._tasks)
```

Two details worth noticing. `add` both mutates state *and* returns the domain object it created — that return value is step 3 of the flow. And `all` returns `list(self._tasks)`, a fresh copy rather than the internal list, so a View that iterates it cannot accidentally mutate the Model (see python-lists[3] on aliasing, and python-representation-invariants[4] on why handing out the internal list would be a leak).

View

```
class TodoView:
    def show_task(self, task: Task) -> None:
        mark: str = "/" if task._done else "X"
        print(f"[{mark}] {task._title}")

    def show_list(self, tasks: list[Task]) -> None:
        print("\n My Tasks\n--------")
        for t in tasks:
            self.show_task(t)
```

`TodoView` is pure presentation: it holds no state at all, and neither method returns anything. `show_list` delegates per-row formatting to `show_task` so the row format is defined in exactly one place.

Controller

```
class TodoApp:
    def __init__(self, todo: TodoList, view: TodoView) -> None:
        self._todo = todo
        self._view = view
```

[3] courses/csse1001/python-lists.pdf
[4] courses/csse1001/python-representation-invariants.pdf

```
    def handle(self, command: str) -> None:
        parts = command.strip().split(maxsplit=1)
        if not parts:
            return
        cmd: str = parts[0]
        if cmd == "add" and len(parts) == 2:
            self._todo.add(parts[1])
            print("Task added!")
        elif cmd == "list":
            self._view.show_list(self._todo.all())
        else:
            print("Commands: add <title>, list, quit")
```

`TodoApp` is handed its Model and its View in `__init__` rather than constructing them — that's what makes the modularity claim from *Why MVC?* actually true here; you change the argument at the call site and nothing inside `TodoApp` changes.

`split(maxsplit=1)` is what lets a title contain spaces: `add Buy groceries` splits into exactly `["add", "Buy groceries"]`, never three parts.

Run loop

```
app = TodoApp(TodoList(), TodoView())
while True:
    user_cmd: str = input("$")
    if user_cmd.strip() in ("quit", "exit"):
        break
    app.handle(user_cmd)
```

```
$add Work on A2
Task added!
$add buy groceries
Task added!
$list
 My Tasks
--------
[X] Work on A2
[X] buy groceries
```

`print("\n My Tasks\n--------")` would actually emit a blank line before `My Tasks`, and the transcript above does not show one.

Tracing `add Buy groceries` through the flow (Figure 1):

1. **User** issues `add Buy groceries`.
2. **Controller** (`TodoApp.handle`) parses the input and calls `TodoList.add`.
3. **Model** (`TodoList`) mutates state and returns a `Task`.
4. **Controller** selects a **View** (e.g. `show_task`) and passes the data.
5. **View** formats the data and pushes it to stdout.
6. Control returns to the main loop, waiting for the next command.

Steps 4 and 5 there describe the *pattern*, not this program: for an `add` command the Controller calls no View at all — it prints `Task added!` from inside `handle` and returns, which is also the first of the three rule-bends below. `show_task` runs only by way of `show_list`, on a `list` command.

Every `[X]` in the transcript above is correct, if a little misleading at first glance — `X` just means "not done" here (`mark = "/" if task._done else "X"`), and nothing in this exercise ever sets `_done = True`. It's not a bug, just an unused feature stub left for extension.

Three places the exercise bends its own rules

Read against the *Anatomy of MVC* table, the listings break its rules in three places. They don't change the answer to the exercise. They're worth noticing because the pattern is easier to remember once you've seen where a small example fails to honour it.

- **The Controller prints.** `handle` calls `print("Task added!")` and `print("Commands: ...")` directly, and the run loop's `input("$")` prompt is written in the Controller layer too. By the table's own third row a Controller "MUST NOT … format output". Doing it properly would mean a `TodoView.show_message` and routing both strings through it.
- **The View reaches into the Model's private attributes.** `show_task` reads `task._done` and `task._title`, both single-underscore "private by convention" names (python-classes-and-objects[5]). The View is behaving correctly in spirit — it only *reads* — but a `Task` with public accessors would keep the boundary honest.
- **`quit` is handled outside `handle`.** The help string advertises `quit`, but `handle` has no branch for it; the run loop intercepts `quit`/`exit` before `handle` ever sees them. Typing `quit` therefore works, while asking `handle` about it prints the help text.

Summary

- MVC is a pattern, typically used in web/GUI applications.
- Separation of concerns makes code more organised.
- It's easier to test, maintain, and extend code built this way.

[5] courses/csse1001/python-classes-and-objects.pdf

- It provides a solid foundation for future enhancements.
- Numerous variants of MVC exist.

> **The diagrams are not course notation**
>
> There is no established shape, layout, arrow direction, or colour for an MVC diagram in this course (see csse1001-diagram-notation[6]). Figure 2 uses the course's class-diagram conventions, apart from the dashed Model boundary and the three layer names. The layout of Figure 1 is not a course convention, and neither is the dependency figure in [mvc]. Don't reproduce them in an exam answer as though they were the course's notation.

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.

[6] courses/csse1001/csse1001-diagram-notation.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[7] — Lecture 10A's content
- uml[8] — the UML reference note
- python-pep8-style-guide[9] — the naming rules cited above
- csse1001[10] — course index

Model-View-Controller (MVC)

Model-View-Controller (MVC) is a design pattern that separates an application into three interconnected components, each with a single, distinct responsibility.

- **Model** — stores the *data* and *business rules* of the application; responsible for **every state change**.
- **View** — some (usually visual) *representation* of the data, **without altering it**.
- **Controller** — an *interface* that receives *user input* (commands), tells the Model what to do, then selects a View to present the result.

[7] courses/csse1001/2025-10-07-uml.pdf
[8] courses/csse1001/uml.pdf
[9] courses/csse1001/python-pep8-style-guide.pdf
[10] courses/csse1001/csse1001.pdf

The asymmetry in that third bullet is the pattern: the Controller is the only component that knows about both of the others. The View does not read from the Model, and the Model does not know about either of the others.

The three components

Layer	Core question	Owns	MUST NOT
Model	What *is* the data?	Data & rules	`print`, parse args, call `input()`
View	How is info *shown*?	Formatting & rendering	change state, validate data
Controller	What happens *next*?	Workflow & commands	store raw data, format output

Model knows nothing about the outside world.

The Model never prints, reads input, or otherwise interacts with the user directly — it is pure data and logic.

Flow of control

1. **User** issues a command.
2. **Controller** parses the input and calls the appropriate **Model** method.
3. **Model** mutates its own state and returns domain data.
4. **Controller** selects a **View** and passes it the data.
5. **View** formats the data and presents it (e.g. prints to stdout).
6. Control returns to the main loop, waiting for the next user action.

Two steps are routinely misremembered. Step 3 returns the domain data **to the Controller**, not to the View — that's why the return arrow in Figure 3 points back left. And step 4 means the View is chosen **after** the Model has already run, which leaves room for which View gets used to depend on what the Model returned.

See 2025-10-14-model-view-controller[11] for the same six steps against a worked example, drawn out step by step — including the one place that example's code departs from them.

[11] courses/csse1001/2025-10-14-model-view-controller.pdf

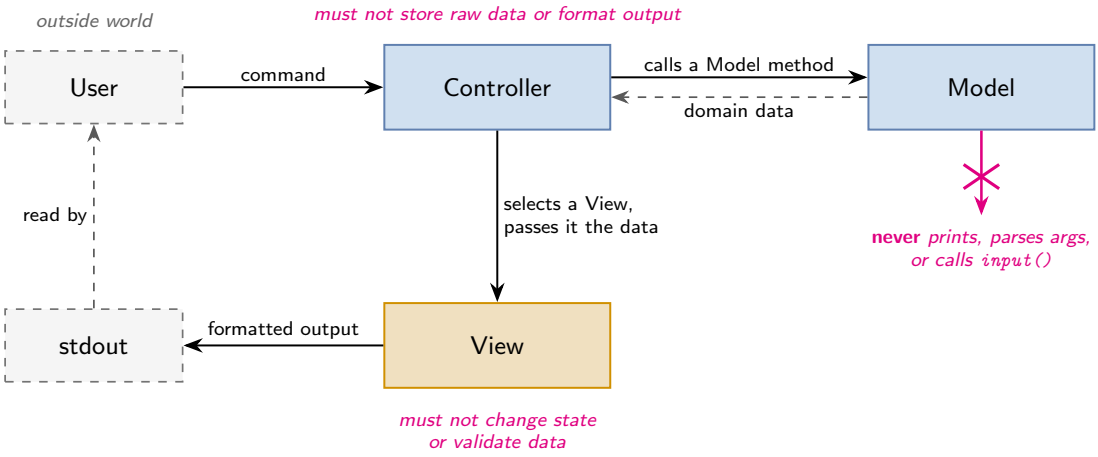

Figure 3: The dependency shape the three-component table implies. Solid arrows are calls; the two dashed arrows are not calls but returns - the Model's domain data going back to the Controller and not to the View, and the output the User reads off stdout; the dashed grey boxes are the outside world, not classes. The magenta text beside each layer is its MUST NOT rule, and the crossed magenta stub under the Model marks that it has no route to the outside world. The box fill colours carry no meaning.

Why use it?

- **Modularity** — swap in a new model, view, or controller without breaking the rest of the application.
- **Testability** — each component can be tested in isolation.
- **Separation of concerns** — each component has exactly one job.

The Model is the easiest of the three to test, because the table above forbids it any I/O at all. Modularity in particular depends on one implementation habit: the Controller should be **given** its Model and View rather than constructing them itself, so that swapping either is a change at the call site and not a change inside the Controller. The worked example does this: its run loop builds `TodoApp(TodoList(), TodoView())` and hands both in.

Minimal skeleton

This is the worked example with the todo-list domain taken out, so that the shape of the three classes is all that is left.

```
class Model:
    def __init__(self):
        self._state = ...

    def do_something(self, ...):
        # mutate self._state, return domain data
        ...

class View:
    def show(self, data):
        # format `data` and present it -- no mutation, no logic
        ...

class Controller:
    def __init__(self, model, view):
        self._model = model
        self._view = view

    def handle(self, command):
        # parse `command`, call the right Model method,
        # then pass its result to the View
        ...

# the run loop belongs with the Controller
```

```
controller = Controller(Model(), View())
while True:
    command = input("$ ")
    if command.strip() in ("quit", "exit"):
        break
    controller.handle(command)
```

The skeleton has no `print` in `Model`, no `input()` anywhere but the run loop, and no reference to `Controller` from inside `View` or `Model`.

Smells that break the pattern

Warning signs that a layer has taken on work belonging to another. Each row is one cell of the MUST NOT column above, turned into something you can spot while reading code.

Smell	Which rule it breaks
A `print` or an f-string inside a Model method	Model must not format output
A Model method that calls `input()` to ask a follow-up question	Model must not read input
A View method that assigns to an attribute of the object it was passed	View must not change state
A View that rejects bad data instead of displaying it	View must not validate
A Controller that keeps its own copy of the task list, scores, etc.	Controller must not store raw data
A Controller that builds the output string itself and prints it	Controller must not format output

In practice

The worked example keeps the three layers in three separate classes (a todo list manager: `Task`/`TodoList` as Model, `TodoView` as View, `TodoApp` as Controller). Real GUI and game applications often merge View and Controller into a single class, since rendering and input handling both live inside the same per-frame/per-event loop. The property worth preserving even then is the Model's isolation: whatever owns rendering and input, the Model itself should stay free of `print`/`input()`/UI-framework calls.

Small teaching examples frequently bend the Controller and View rules too — the worked exercise in 2025-10-14-model-view-controller[12] has its Controller printing status messages

[12] courses/csse1001/2025-10-14-model-view-controller.pdf

directly and its View reading private attributes off Model objects — which is worth knowing about, because "does this obey the table?" is a much sharper question than "does this look like MVC?".

Numerous variants of MVC exist. Treat the version above as the base form.

See python-inheritance[13] for abstract base classes and MRO, and python-composition[14] for is-a vs has-a.

Python Classes and Objects

An **object** bundles data (**fields/attributes**, comprising its state) with **methods** (class-scoped functions that act on that data). An object can reference and change its own state, and so has a notion of *self*.

A **class** is the blueprint; an **object** is an instance of that class. Keeping the data inside the class and reaching it only through the class's own attributes and methods is **encapsulation**.

Introduced in 2025-09-09-object-oriented-programming[15] (Lecture 7C).

Defining a class

```
class ClassName():
    def __init__(self, ...):
        self.attribute = ...

    def some_method(self, ...):
        return ...
```

- Class names use `CamelCaps` by convention (`ClassNamesAreLikeThis`).
- `__init__` is the **initializer**, run automatically when an instance is created.
- Every method's first parameter is `self` — Python supplies it automatically; never pass an argument for it explicitly.
- Think of the class body as a blueprint with placeholders: `self` has an `x`, `self` has a `y`. On **instantiation**, `self` becomes the particular object being built and each placeholder becomes a real attribute stored in it, so many instances can share one blueprint while each holds its own values.

[13] courses/csse1001/python-inheritance.pdf
[14] courses/csse1001/python-composition.pdf
[15] courses/csse1001/2025-09-09-object-oriented-programming.pdf

Instantiation and attributes

```
>>> p = ClassName(...)       # creates an instance, passing args to __init__
>>> p.attribute              # dot notation accesses instance variables
>>> p
<__main__.ClassName object at 0x10a9e0dd8>
>>> type(p)
<class '__main__.ClassName'>
```

A method is called the same way: `<object_variable>.<method>(<parameters>)`. Calling it as a bare name is a `NameError` — it only exists on the object.

Equality vs. aliasing

- Two separately-constructed instances with identical attributes are **not** `==` by default: the default comparison is identity, so equal-looking objects at two different addresses compare `False`. Defining what equality *means* for a class is the job of `__eq__` — see python-dunder-methods[16].
- Assigning `q = p` makes `q` an **alias** for the same object as `p` — mutating one is visible through the other, and `p == q` holds because they're literally the same object.

Private variables, getters, and setters

A leading underscore (`self._value`) signals that an attribute is intended for internal use only — a *convention*, not an enforced restriction; nothing stops external code from reading or writing it directly (which is bad practice).

Prefer exposing controlled access via methods:

```
def get_value(self) -> int:            # getter
    return self._value

def set_value(self, x: int) -> None:   # setter
    if x < 0:
        raise ValueError
    self._value = x
```

[16] courses/csse1001/python-dunder-methods.pdf

The `raise` is the reason the setter exists: it is the one place the class can reject a bad value before it becomes state (see python-exceptions[17]).

Python's `@property`/`@<name>.setter` decorators let a getter/setter pair be used with ordinary attribute syntax (`obj.name` / `obj.name = value`) instead of explicit method calls, so validation still runs while call sites stay readable.

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[18]'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:

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

[17] courses/csse1001/python-exceptions.pdf
[18] courses/csse1001/python-composition.pdf

The subclass…	…does this	Effect on the parent's version
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.

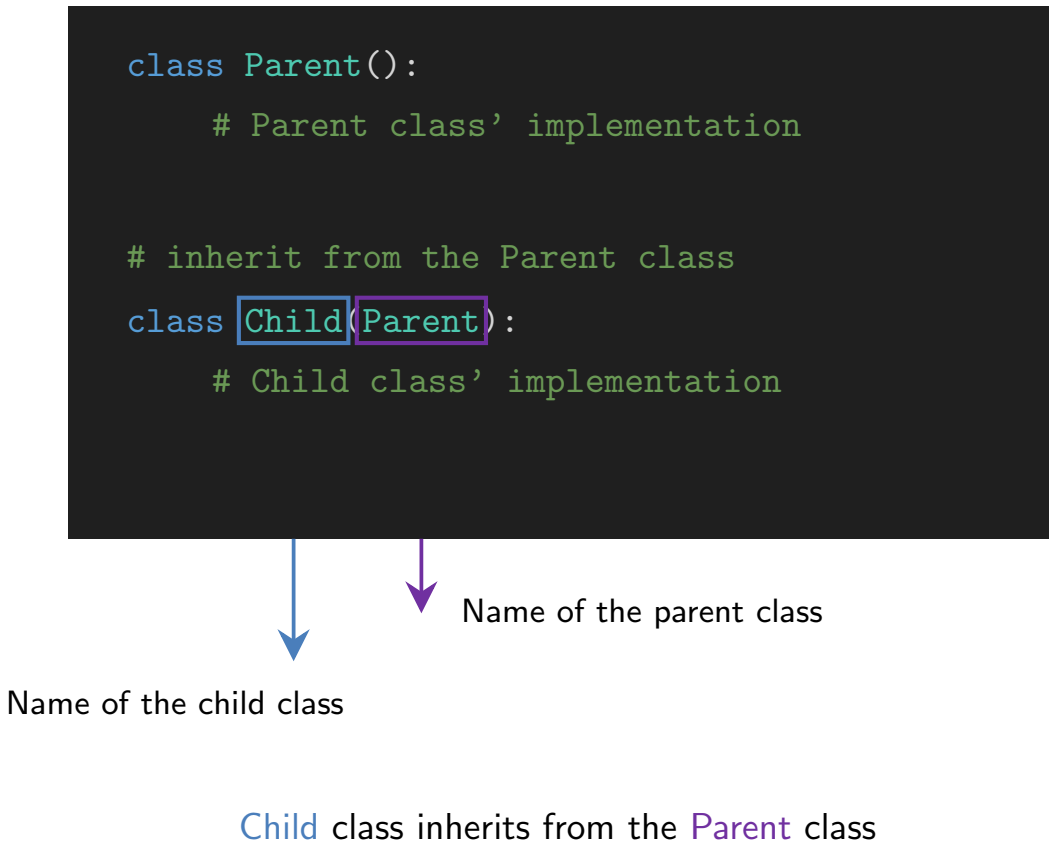

Figure 4: The `class Child(Parent):` line, annotated: the name being *defined* comes first, the name being *inherited from* goes in the parentheses. The box round each name, the arrow leading from it, and the matching word in the sentence beneath are all one colour: blue tracks the child and purple the parent throughout.

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[19] instead. See 2025-09-23-inheritance[20] for the worked `Robot` hierarchy and 2025-09-23-composition[21] 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:

```
from abc import ABC, abstractmethod

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

[19] courses/csse1001/python-composition.pdf
[20] courses/csse1001/2025-09-23-inheritance.pdf
[21] courses/csse1001/2025-09-23-composition.pdf

```
    # 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 Lists

A **list** is a **mutable** ordered collection of elements. These elements are *not necessarily* of the same type. Square brackets `[]` are used to create lists in Python.

```
>>> xs = [1, "apple"]
>>> type(xs)
<class 'list'>
>>> xs[0]
1
>>> xs[0] = 2*xs[1]
>>> xs[0]                  # note: lists are mutable
'appleapple'
```

Comparison

Lists compare point-wise from position zero:

```
>>> [1, 2, 3] < [4, 5, 6]
True
>>> [7, 2, 3] < [4, 5, 6]
False
>>> [] < [1]
True
```

The second line is the one to read carefully: the comparison stops at position zero, where `7 < 4` is `False`, and never looks at positions one and two.

Membership

```
>>> 1 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>> 123 in [1, 2, 3]
False
>>> [1] in [1, 2, 3]
False
```

`in` tests for an *element*, not a sub-list — the last line is `False` because `[1, 2, 3]` contains the integer `1`, not the list `[1]`.

Adding elements: `append`, `extend`, concatenation

`xs + [y]` **creates a new list**, which must be reassigned back to `xs` to have an effect:

```
>>> xs = [0, 1, 2]
>>> xs = xs + [3]
>>> xs
[0, 1, 2, 3]              # "this is the original list updated"
```

Read the comment as a statement about the *value* of `xs`, not about the object: `xs = xs + [...]` builds a **new** list and rebinds `xs` to it (see the aliasing transcript below), which is exactly why it breaks an alias where `append` does not.

Alternatively, using a method that `list` provides — `xs.append(y)` mutates `xs` **in place**:

```
>>> xs = [0, 1, 2]
>>> xs.append(3)
>>> xs
[0, 1, 2, 3]
```

`append` adds its argument as a *single new element*, even when that argument is itself a list:

```
>>> xs = [1, 2, 3]
>>> xs.append([4,5])
>>> xs
[1, 2, 3, [4, 5]]
```

`xs.extend(ys)` instead appends every element of `ys` individually — code similar to `xs = xs + ys`:

```
>>> xs = [1, 2, 3]
>>> xs.extend([4,5])
>>> xs
[1, 2, 3, 4, 5]
```

Aliasing

Take care when (trying to) copy lists. Assigning one list variable to another does **not** copy it — both names refer to the *same* list object:

```
>>> xs = [1, 2, 3]
>>> ys = xs
>>> ys[-1] = 9
>>> ys
[1, 2, 9]
>>> xs
[1, 2, 9]
```

Whether an operation preserves this aliasing relationship depends on whether it mutates the list in place or creates a new one:

```
>>> xs = [1, 2, 3]
>>> ys = xs           # ys is an alias of xs
>>> xs.append(4)      # appending to a list
>>> ys
[1, 2, 3, 4]
>>> xs += [5]         # appending to a list
>>> ys
[1, 2, 3, 4, 5]
>>> xs = xs + [6]     # concatenating two lists
>>> ys
[1, 2, 3, 4, 5]       # aliasing relationship broken!
```

The trap is that `xs += [5]` and `xs = xs + [6]` look like the same operation and are not: the first mutates the object both names share, the second builds a *new* object and rebinds only `xs`.

`.copy()` breaks the alias for a flat list:

```
>>> xs = [1, 2, 3]
>>> ys = xs.copy()
>>> ys[-1] = 9
>>> xs
[1, 2, 3]
>>> ys
[1, 2, 9]
```

Passing lists to functions

A more nuanced case is when passing lists (but really *list references*) to functions — mutating a parameter inside a function mutates the caller's list too:

```
>>> def foo(ys):
...     ys[0] = -100
>>> xs = [1, 2, 3]
>>> foo(xs)
>>> xs
[-100, 2, 3]
```

This is the same picture as (a) in Figure 5, with `ys` being a name in `foo`'s frame rather than the global one.

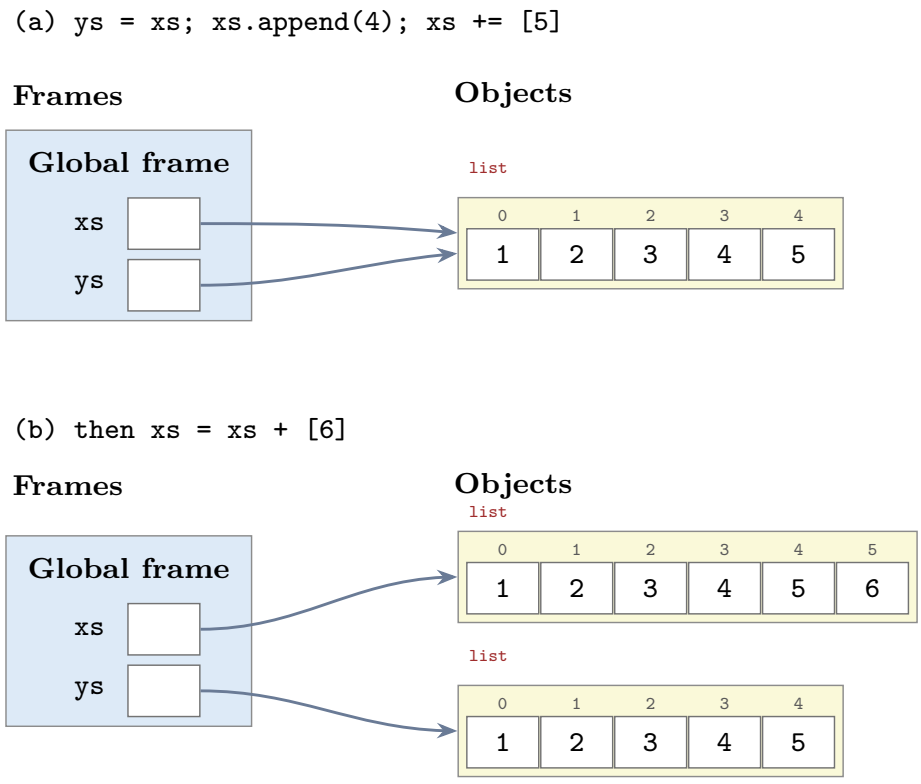

Figure 5: What the transcript above is doing to memory. In (a) `xs` and `ys` are two names for one list object, so the in-place operations `append` and `+=` are visible through both. In (b) `xs = xs + [6]` builds a second list object and rebinds `xs` to it; `ys` still refers to the original, which is what *aliasing relationship broken* means. The layout is Python Tutor's: frames on the left in pale blue, objects on the right in pale yellow, a reference as an arrow, and aliasing as two arrows converging on one object.

Nested lists and matrices

A list can have another list as an element:

```
>>> xs = [ [1,2], [5,6,7] ]
>>> len(xs)
2
>>> len(xs[-1])
3
```

`len` counts the *outer* elements — two, not five.

A **matrix** can therefore be represented by a list-of-lists in Python:

$$\mathbb{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \equiv \texttt{[[1, 2, 3], [4, 5, 6], [7, 8, 9]]}$$

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass[-1]
[7, 8, 9]
>>> ass[1:3]
[[4, 5, 6], [7, 8, 9]]
>>> ass[1]
[4, 5, 6]
>>> ass[1][-1]
6
>>> ass[1,-1]
TypeError: list indices must be integers or slices, not tuple
```

The last line is the one worth remembering: a nested list is indexed with *repeated* brackets, `ass[1][-1]`. The comma form `ass[1,-1]` is a tuple and is a `TypeError` for a list.

Deep copying

`.copy()` only performs a **shallow copy** — nested mutable elements (like inner lists) are still shared between the original and the copy:

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> bss = ass.copy()
>>> bss[0][0] = 0
>>> bss
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]   # A was changed!
```

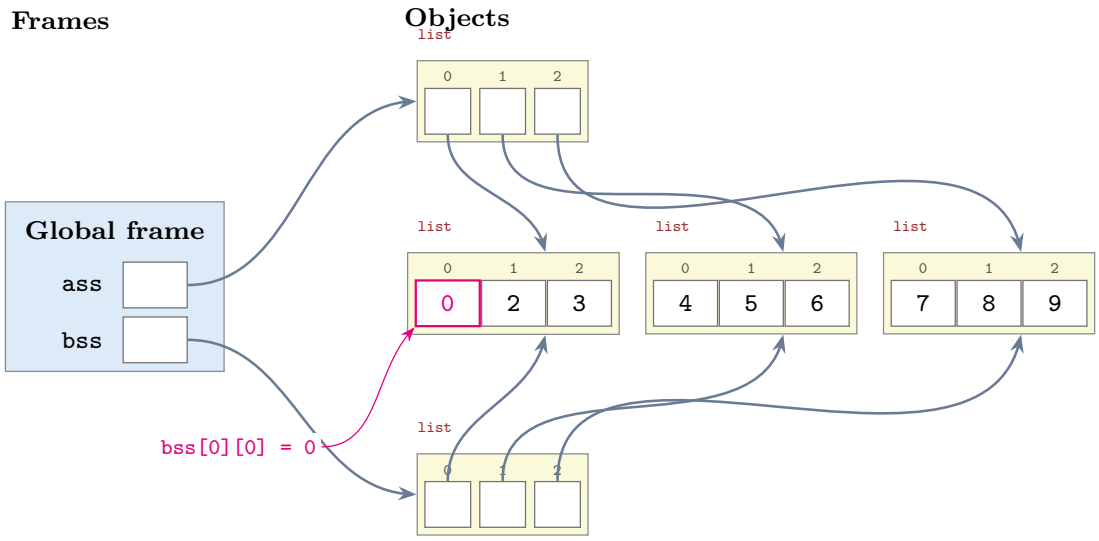

Figure 6: Why `ass` changed. `bss = ass.copy()` made a genuinely new *outer* list — that is why `bss` and `ass` are two distinct objects — but it copied the outer list's three references, not the objects they point at, so both outer lists share the same three inner lists. Writing `bss[0][0] = 0` reaches through a shared reference and the change shows up through `ass` as well. For this to work as expected we need a *deep-copy*, which is not built-in — see `copy.deepcopy`. Magenta marks the mutated cell.

A true independent copy of nested structures needs a **deep copy**, which is *not* built into `list` — see `copy.deepcopy`.

Slicing

```
>>> xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xs[3:6]
[3, 4, 5]
>>> xs[::2]
[0, 2, 4, 6, 8]
```

```
>>> xs[7:2:-2]
[7, 5, 3]
```

The three fields are `start:stop:step` with `stop` exclusive: `xs[3:6]` takes positions 3, 4 and 5; `xs[::2]` takes every second element of the whole list; and a negative step makes `xs[7:2:-2]` walk *backwards* from position 7, stopping before position 2.

Type hints

> **Warning.** Type-hinting lists (e.g. `list[int]`) is a new feature in Python 3.9 and greater.

Unpacking into function arguments

It is possible to "unbracket" a list when passing it to functions, with `*`:

```
>>> def f(x, y, z):
...     return x + y + z
>>> x = [1, 2, 3]
>>> f(x)
TypeError: f() missing 2 required positional arguments: 'y' and 'z'
>>> f(*x)
6
```

Tuples

Tuples are **immutable ordered collections** whereas lists are **mutable ordered collections**. Both can be sliced and indexed. See python-primitive-data-types[22]; a tuple's immutability is what makes it usable as a dictionary key, where a list is not — see python-dictionaries[23].

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[24] for the lecture note.

[22] courses/csse1001/python-primitive-data-types.pdf
[23] courses/csse1001/python-dictionaries.pdf
[24] 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[25] 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.

[25] 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[26] 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[27] 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[28]).
- 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.

[26] courses/csse1001/2025-09-16-representation-invariants.pdf
[27] courses/csse1001/python-lists.pdf
[28] courses/csse1001/python-testing.pdf