CSSE1001 — Week 11 Notes

Table of contents

Model View Controller

See csse1001[1] for course logistics — this note covers Lecture 11A's technical content. See [mvc] for the full reference on the Model-View-Controller pattern.

Advanced inheritance (recap)

A quick recap table from the end of last week, tying MRO complexity to each inheritance model (see python-inheritance[2] for the full detail):

Inheritance type	MRO complexity
Single	Simple linear order
Multilevel	Chain-like resolution

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

Inheritance type	MRO complexity
Multiple	C3 linearization
Hybrid (e.g. diamond)	C3 linearization

A few extra facts about MRO worth reinforcing:

- 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 entirely 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 after the current one on `MyClass`'s MRO* (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.

Anatomy of MVC

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

> The Model knows nothing about the outside world — no printing, no argument parsing, no reading input. It's pure data and logic.

MVC flow

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.

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 the MVC design pattern:

- `Task` + `TodoList` → **Model** (domain)
- `TodoView` → **View** (presentation only)
- `TodoApp` + run-loop → **Controller** (input parsing & orchestration)

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)
```

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)
```

Controller

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

    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")
```

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
```

Tracing through `add Buy groceries` against the MVC flow above:

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** method (`show_task`, or `show_list` for `list`) 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.

   Every `[X]` in the output 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.

Worked example: a larger system

`game.py` (a small pygame grid-world) is a good example of these same ideas showing up in a messier, more realistic setting:

- **Model**: `Grid`, `Cell`, `Robot`, and the `Enemy` hierarchy hold all the state and rules — cell rewards, slipperiness, robot/enemy position, and score.
- `Enemy` is itself an **abstract base class** in the unenforced style from python-inheritance[3]: `act()` just raises `NotImplementedError`, and the two concrete subclasses `RandomEnemy` and `ChaserEnemy` each provide their own `act()` — `RandomEnemy` picks a random legal direction, `ChaserEnemy` greedily moves toward whichever direction minimises squared distance to the robot.

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

- **View + Controller**: `Game` owns *both* of these — `draw()`/`_draw_grid()`/`_draw_hud()`/`_draw_actor()` render the current state (View), while `handle_events()` reads keyboard input and directly drives the Model (`self.robot.move(...)`, `self.enemy.act(...)`, `self.enemy.check_collision(...)`) (Controller).

  This is a useful contrast with the todo list example above: real GUI/game loops very often merge View and Controller into one class for pragmatic reasons (rendering and input handling are both tied to the same per-frame loop), even though the stricter three-way split is what's taught here. The important invariant that *is* preserved is the one in the "MUST NOT" table: the Model (`Grid`, `Cell`, `Robot`, `Enemy`) never calls `print` or `input()`, and knows nothing about pygame at all.

  One subtlety in `handle_events()`: on every keypress the order is always robot moves, then the enemy acts, then collision is checked — so it's possible for the robot to step onto the enemy's *old* square and the enemy to then step away in that same tick, without a collision ever being registered. Not necessarily a bug, but worth noticing if the game feels like it "should" have caught you.

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

Next: further design patterns (Lecture 11B).

Reference material

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.

The three components

Layer	Core question	Owns	MUST NOT
Model	What *is* the data?	Data & business 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

The Model knows nothing about the outside world — it never prints, reads input, or otherwise interacts with the user directly. It's 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.

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 (the Model especially, since it has no I/O).
- **Separation of concerns** — each component has exactly one job.

Minimal skeleton

```
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
        ...
```

In practice

Strict three-way separation is common in textbook examples (e.g. a todo list manager: `Task`/`TodoList` as Model, `TodoView` as View, `TodoApp` as Controller), but real GUI and game applications very often merge View and Controller into a single class — rendering and input handling both naturally live inside the same per-frame/per-event loop. The property that's still 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.

See python-inheritance[4] for abstract base classes and MRO, and python-composition[5] for is-a vs has-a — both are commonly combined with MVC (e.g. an abstract `Enemy` base class within a game's Model layer).

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

Syntax

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

[4] courses/csse1001/python-inheritance.pdf
[5] courses/csse1001/python-composition.pdf
[6] courses/csse1001/python-composition.pdf

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

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

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

`super()`

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

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

Terminology cheat-sheet

Given `class Student(Person):`

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

Polymorphism

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

When to use inheritance

Only when there's a genuine **"is-a"** relationship between the two classes. If the relationship is really "has-a" (one object *contains* or *uses* another), prefer python-composition[7] instead.

Abstract base classes

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

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

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

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

```
from abc import ABC, abstractmethod

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

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

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

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

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.