Model-View-Controller (MVC)

Table of contents

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[1] for abstract base classes and MRO, and python-composition[2] 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).

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