Week 2 — Advanced OOP, Design Patterns & Architecture

Félix Chavelli ✉ felix.chavelli@inria.fr

February 25, 2026 · Semester 2

PSL UNIVERSITÉ PARIS

Today's Agenda

S1 Recap & Motivation

Inheritance

Polymorphism

Abstract Classes & Interfaces

Operator Overloading

Design Patterns

UML Class Diagrams

Summary & What's Next

Part 1

S1 Recap & Motivation

What You Already Know from S1

Covered in S1 (Week 5):

- Classes, objects, instances
- Properties (attributes) and methods
- Constructors (`__init__`, C++ constructors)
- Destructors (`__del__`, `~Class()`)
- `self` (Python) / `this` (C++)
- Static members and methods
- Copy constructors (C++)
- Basic encapsulation (`public`/`private`)
- Introduction to inheritance & polymorphism

→ Today: Going Deeper

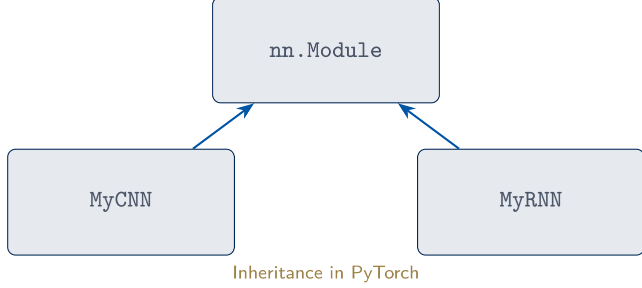
- Inheritance hierarchies
- Abstract classes & interfaces
- Operator overloading
- Design patterns
- UML class diagrams

Why Advanced OOP Matters for AI

Motivation

Modern AI frameworks are **built on OOP**. Understanding inheritance and polymorphism is essential to use — and extend — them effectively.

- **PyTorch:** every model subclasses `torch.nn.Module`
- **scikit-learn:** estimators follow the `fit`/`predict` interface
- **TensorFlow/Keras:** layers, callbacks, losses are all class hierarchies
- Design patterns help write modular, testable, extensible code



Inheritance in PyTorch

Part 2

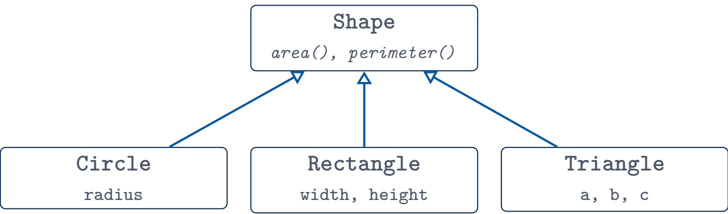
Inheritance

Inheritance — Core Concepts

Inheritance

A **subclass** (derived class) inherits the properties and methods of its **superclass** (base class). It can:

- **Add** new attributes and methods
- **Override** existing methods to specialize behavior



Inheritance in Python

Python

```
import math

class Shape:
    def area(self):
        raise NotImplementedError

    def perimeter(self):
        raise NotImplementedError

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

    def perimeter(self):
        return 2 * math.pi * self.radius
```

- Syntax: `class Sub(Base):`
- Use `super()` to call parent methods
- All methods are virtual by default
- `isinstance(obj, Cls)` checks the hierarchy

Python

```
c = Circle(5)
print(c.area())
# 78.539...
print(isinstance(c, Shape))
# True
```

Inheritance in C++

C++

```
#include <cmath>
#include <iostream>

class Shape {
public:
    virtual double area() const = 0;          // pure virtual
    virtual double perimeter() const = 0;     // pure virtual
    virtual ~Shape() = default;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return M_PI * radius * radius;
    }
    double perimeter() const override {
        return 2 * M_PI * radius;
    }
};
```

Access Specifiers in C++

Access	Same class	Derived class	Outside
`public`	✓	✓	✓
`protected`	✓	✓	✗
`private`	✓	✗	✗

C++

```cpp
class Shape {
protected:
    std::string color;  // accessible in derived classes
public:
    virtual double area() const = 0;
private:
    int internal_id;    // only in Shape itself
};
```

Key Idea

Rule of thumb: make data `protected` or `private`, expose behavior via `public` methods.

Calling the Parent: `super()` and Scope Resolution

Python:

Python

```python
class Rectangle(Shape):
    def __init__(self, w, h):
        self.width = w
        self.height = h

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2*(self.width+self.height)

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)
```

C++:

C++

```cpp
class Rectangle : public Shape {
    double w, h;
public:
    Rectangle(double w, double h)
        : w(w), h(h) {}

    double area() const override {
        return w * h;
    }
    double perimeter() const override {
        return 2 * (w + h);
    }
};

class Square : public Rectangle {
public:
    Square(double s)
        : Rectangle(s, s) {}
};
```

Multiple Inheritance & MRO (Python)

Python

```
class Flyable:
    def fly(self):
        return "I can fly!"

class Swimmable:
    def swim(self):
        return "I can swim!"

class Duck(Flyable, Swimmable):
    def quack(self):
        return "Quack!"

d = Duck()
print(d.fly())     # "I can fly!"
print(d.swim())    # "I can swim!"
```

- Python supports multiple inheritance
- **MRO** (Method Resolution Order): determines which method is called
- Uses **C3 linearization**
- Check with `Duck.__mro__`

Diamond Problem

When two base classes share a common ancestor → `super()` follows MRO

Part 3

Polymorphism

Polymorphism — The Big Picture

Polymorphism

The ability to use objects of **different types** through a **uniform interface**. The actual behavior depends on the **runtime type** of the object.

Static typing (C++):

- Achieved via virtual functions
- Base class pointer/reference calls derived method
- Resolved at **runtime** (vtable)

Dynamic typing (Python):

- Duck typing: "If it walks like a duck. . . "
- No need to declare a common base class
- Method lookup at **runtime**

Polymorphism in Action

Python:

Python

```python
def print_info(shape):
    """Works with any Shape."""
    print(f"Area: {shape.area():.2f}")
    print(f"Perim: {shape.perimeter():.2f}")

shapes = [Circle(3), Rectangle(4,5),
          Square(2)]

for s in shapes:
    print_info(s)
```

C++:

C++

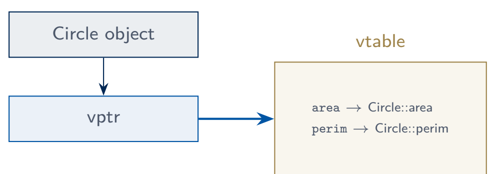
```cpp
void print_info(const Shape& s) {
    std::cout << "Area: "
              << s.area() << "\n";
    std::cout << "Perim: "
              << s.perimeter() << "\n";
}

// Must use pointers or references
std::vector<Shape*> shapes = {
    new Circle(3),
    new Rectangle(4, 5),
    new Square(2)
};
for (auto* s : shapes)
    print_info(*s);
```

Key Idea

One function, many behaviors — that is the power of polymorphism.

Virtual Functions & Vtables (C++)

- A **`virtual`** method enables dynamic dispatch
- The compiler builds a **vtable** (virtual table) for each class
- Each object holds a hidden pointer to its class's vtable
- At call time, the vtable is consulted → correct method



C++

```cpp
Shape* s = new Circle(5);
s->area(); // calls Circle::area()
           // via vtable lookup
```

Part 4

Abstract Classes & Interfaces

Abstract Classes

Abstract Class

A class that **cannot be instantiated** directly. It defines an **interface** that derived classes must implement.

Python

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Shape() -> TypeError!
```

C++

```cpp
class Shape {
public:
    virtual double area()
        const = 0;      // = 0 => pure
    virtual double perimeter()
        const = 0;
    virtual ~Shape() = default;
};

// Shape s; -> compile error!
```

Key Idea

Abstract classes define a **contract**: "every concrete shape must provide `area()` and `perimeter()`".

Interfaces — Python Protocols

- Python 3.8+ introduces `Protocol` (structural subtyping)
- No need to inherit — just implement the required methods

Python

```python
from typing import Protocol

class Drawable(Protocol):
    def draw(self) -> None: ...

class Circle:
    def draw(self) -> None:
        print("Drawing circle")

def render(obj: Drawable) -> None:
    obj.draw()  # works: Circle has draw()

render(Circle())  # OK, no inheritance needed
```

Key Idea

Protocols formalize **duck typing** with type-checker support.

Part 5

Operator Overloading

Operator Overloading — Overview

Operator Overloading

Redefine the behavior of built-in operators (+, ∗, ==, [], . . .) for user-defined types.

Operation	Python	C++
`a + b`	`__add__(self, other)`	`operator+(const T& other)`
`a * b`	`__mul__(self, other)`	`operator*(const T& other)`
`a == b`	`__eq__(self, other)`	`operator==(const T& other)`
`a[i]`	`__getitem__(self, i)`	`operator[](size_t i)`
`str(a)`	`__str__(self)`	`operator<<(ostream&, T)`
`len(a)`	`__len__(self)`	*(no equivalent)*

Building a `Vector` Class — Python

Python

```
class Vector:
    def __init__(self, *components):
        self._data = list(components)

    def __add__(self, other):
        return Vector(*(a+b for a,b in zip(self._data, other._data)))

    def __mul__(self, scalar):
        return Vector(*(x * scalar for x in self._data))

    def __rmul__(self, scalar):
        return self.__mul__(scalar)

    def __eq__(self, other):
        return self._data == other._data

    def __repr__(self):
        return f"Vector({', '.join(map(str, self._data))})"

    def __len__(self):
        return len(self._data)

    def dot(self, other):
        return sum(a*b for a,b in zip(self._data, other._data))
```

Building a Vector Class — C++

C++

```
class Vector {
    std::vector<double> data;
public:
    Vector(std::initializer_list<double> vals) : data(vals) {}
    Vector operator+(const Vector& other) const {
        Vector result = *this;
        for (size_t i = 0; i < data.size(); ++i)
            result.data[i] += other.data[i];
        return result;
    }
    Vector operator*(double scalar) const {
        Vector result = *this;
        for (auto& x : result.data) x *= scalar;
        return result;
    }
    double dot(const Vector& other) const {
        return std::inner_product(
            data.begin(), data.end(), other.data.begin(), 0.0);
    }
    friend Vector operator*(double s, const Vector& v) {
        return v * s;
    }
    friend std::ostream& operator<<(std::ostream& os, const Vector& v) {
        os << "Vector(";
        for (size_t i = 0; i < v.data.size(); ++i)
            os << (i ? ", " : "") << v.data[i];
        return os << ")";
    }
};
```

Part 6

Design Patterns

What Are Design Patterns?

Design Pattern

A **reusable solution** to a commonly occurring problem in software design. Not code — a **template** for solving a type of problem.

- Coined by the "Gang of Four" (GoF, 1994): Gamma, Helm, Johnson, Vlissides
- Three categories:
  1. **Creational:** how objects are created
  2. **Structural:** how objects are composed
  3. **Behavioral:** how objects interact
- We focus on three patterns especially relevant to algorithms & AI:

Strategy	**Iterator**	**Decorator**
Behavioral	Behavioral	Structural

Strategy Pattern

Interchangeable algorithms at runtime

Problem: you want to switch between different algorithms (e.g., loss functions, sorting strategies) without changing client code.

Python

```
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    @abstractmethod
    def sort(self, data: list) -> list:
        pass

class BubbleSort(SortStrategy):
    def sort(self, data):
        # ... bubble sort logic ...
        return sorted_data

class MergeSort(SortStrategy):
    def sort(self, data):
        # ... merge sort logic ...
        return sorted_data
```

Python

```
class Sorter:
    def __init__(self, strategy):
        self._strategy = strategy

    def set_strategy(self, strategy):
        self._strategy = strategy

    def sort(self, data):
        return self._strategy.sort(
            data
        )

s = Sorter(BubbleSort())
s.sort([3,1,2])
s.set_strategy(MergeSort())
s.sort([3,1,2])
```

→ AI example: interchangeable loss functions in a training loop.

Iterator Pattern

Traversing a collection without exposing its internals

Python (built-in support):

Python

```
class BinaryTree:
    def __init__(self, val, left=None,
                 right=None):
        self.val = val
        self.left = left
        self.right = right

    def __iter__(self):
        """In-order traversal."""
        if self.left:
            yield from self.left
        yield self.val
        if self.right:
            yield from self.right

tree = BinaryTree(2,
    BinaryTree(1), BinaryTree(3))
for v in tree:
    print(v)  # 1 2 3
```

- Separates how you traverse from what you traverse
- Python uses `__iter__` + `__next__`
- `yield` / `yield from` make generators
- C++: STL iterators (`begin()`, `end()`)

Key Idea

The `for` loop in Python automatically calls `__iter__()`. This is the Iterator pattern built into the language!

Decorator Pattern

Adding behavior without modifying existing classes

Python

```python
import time

def timer(func):
    """Measure execution time."""
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - start
        print(f"{func.__name__}: "
              f"{elapsed:.4f}s")
        return result
    return wrapper

@timer
def slow_sort(data):
    return sorted(data)

slow_sort(list(range(10**6, 0, -1)))
# slow_sort: 0.2345s
```

- Python `@decorator` syntax wraps a function
- The GoF Decorator pattern wraps objects:

Python

```python
class LoggingModel:
    """Decorator for any model."""
    def __init__(self, model):
        self._model = model

    def predict(self, x):
        print(f"Input: {x}")
        result = self._model.predict(x)
        print(f"Output: {result}")
        return result
```

→ AI: add logging, caching, or validation around any model.

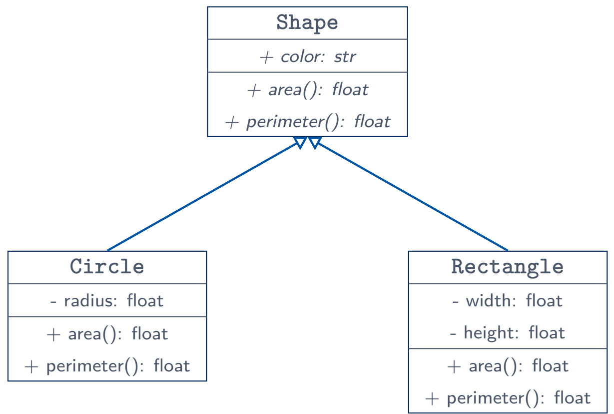
Part 7

UML Class Diagrams

Introduction to UML Class Diagrams

UML Class Diagram

A **visual representation** of the structure of a system: classes, attributes, methods, and relationships between classes.



Notation: + public, − private, # protected, *italic* = abstract

UML Relationships

Relationship	Arrow	Meaning
Inheritance	▷——	"is-a" (Circle *is a* Shape)
Association	⟶	"uses" (Sorter uses SortStrategy)
Composition	◆⟶	"owns" (Car owns Engine)
Aggregation	◇⟶	"has" (Department has Professors)

Key Idea

Composition vs. Inheritance: Prefer composition ("has-a") over inheritance ("is-a") when the relationship is about **behavior**, not **identity**.
Example: `Sorter` *has a* `SortStrategy`, not *is a* `SortStrategy`.

Part 8

Summary & What's Next

Summary — Key Takeaways

1. **Inheritance** lets you build class hierarchies; use `super()` / scope resolution
2. **Polymorphism** provides uniform interfaces with varying behavior
   - C++: `virtual` + vtable   Python: duck typing
3. **Abstract classes** define contracts that subclasses must fulfill
   - Python: `ABC` + `@abstractmethod`   C++: pure virtual (`= 0`)
4. **Operator overloading** makes user types feel natural
5. **Design patterns** are reusable blueprints:
   - Strategy: swap algorithms
   - Iterator: traverse uniformly
   - Decorator: add behavior transparently
6. **UML diagrams** help communicate design visually

What's Next — Week 3

→ Next week: Heaps, Heapsort & Priority Queues (CLRS Ch. 6)

- Heap property, array representation
- MAX-HEAPIFY, BUILD-MAX-HEAP
- Heapsort: $\mathcal{O}(n \log n)$ in-place
- Priority queues: `heapq` (Python), `std::priority_queue` (C++)

To prepare: CLRS Chapter 6; review today's shape hierarchy exercises.

References

- **Gamma, Helm, Johnson, Vlissides** — *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- **Ramalho, L.** — *Fluent Python*, 2nd ed., O'Reilly, 2022.
- **Stroustrup, B.** — *The C++ Programming Language*, 4th ed.

PSL University · Bachelor of Science in AI · 2025–2026