Tuần 3: Tổng hợp kiến thức buổi học số 3 + 4

Time-Series Team

Ngày 21 tháng 6 năm 2025

Buổi học số 3 + 4 (Thứ 3 + Thứ 4, 17/06/2025 + 18/06/2025) bao gồm hai nội dung chính:

- ***Phần I: Khái niệm cơ bản về Object-Oriented Programming: Lập trình hướng đối tượng***
- ***Phần II: Các tính chất cơ bản trong Object-Oriented Programming***
- ***Phần III: Ứng dụng Custom Layer trong PyTorch***

Phần I: Khái niệm cơ bản về Object-Oriented Programming: Lập trình hướng đối tượng

1 Giới thiệu về lập trình hướng đối tượng

Lập trình hướng đối tượng (OOP) là một phương pháp lập trình phổ biến, lấy các đối tượng làm trung tâm để giải quyết vấn đề thực tế. Để hiểu rõ OOP, ta cần nắm các khái niệm cơ bản như phạm vi biến, trừu tượng hóa, lớp, đối tượng và các tính chất đặc trưng như kế thừa, đa hình, đóng gói và trừu tượng.

1.1 Bước đầu cho OOP - Khái niệm biến local và global (biến cục bộ và biến toàn cục)

Phạm vi (scope) biến là yếu tố quan trọng trong lập trình, quyết định nơi một biến có thể truy cập được.

1.1.1 Biến cục bộ (Local)

Biến cục bộ chỉ tồn tại và có thể sử dụng trong một hàm hoặc phương thức cụ thể.

Quy tắc ưu tiên (Scope resolution):

$$\text{Local} > \text{Instance} > \text{Class} > \text{Global} \tag{1}$$

Ví dụ minh họa:

```
class Cat():
    age = 1 #Class variabe
    def describe(self, age):
        print(age, age)  # Ouput: 2,2 (both are local variables)
        print(self.age)  # Output: 1 (class variable)

cat = Cat()
cat.describe(2)
```

Cả hai biến age ở lần in đầu tiên đều là biến cục bộ vì Python sẽ ưu tiên tìm biến cục bộ trước, ở đây là đối số 2 truyền vào age. Vì ta không định nghĩa self.age, nên khi gọi, Python sẽ tìm đến thứ tự tiếp theo là biến Class. Chính vì vậy, ta sẽ in ra được kết quả age = 1

1.1.2 Biến toàn cục (Global)

Biến toàn cục được khai báo bên ngoài hàm hoặc class, sử dụng được trên toàn chương trình nhưng nên hạn chế để tránh gây khó kiểm soát.

Không khuyến khích:

```
counter = 0  # Global variable

class MyClass:
    def increment(self):
        global counter
        counter += 1
```

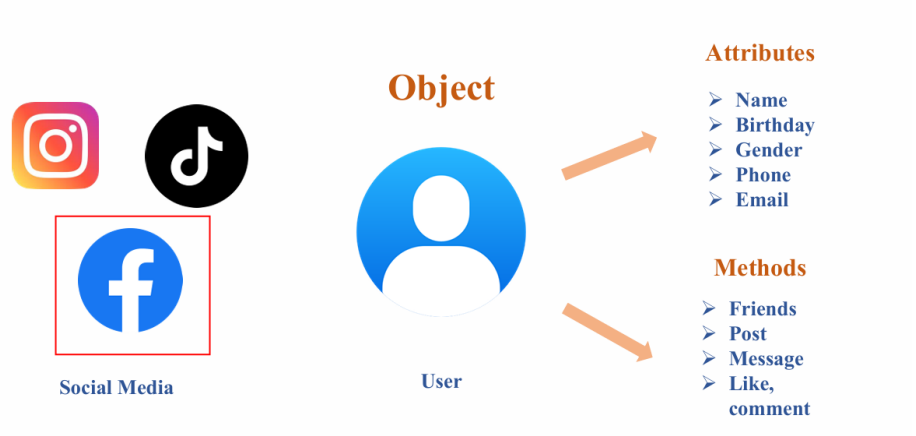
Khuyến khích:

```
class MyClass:
    counter = 0  # Class attribute

    def increment(self):
        MyClass.counter += 1
```

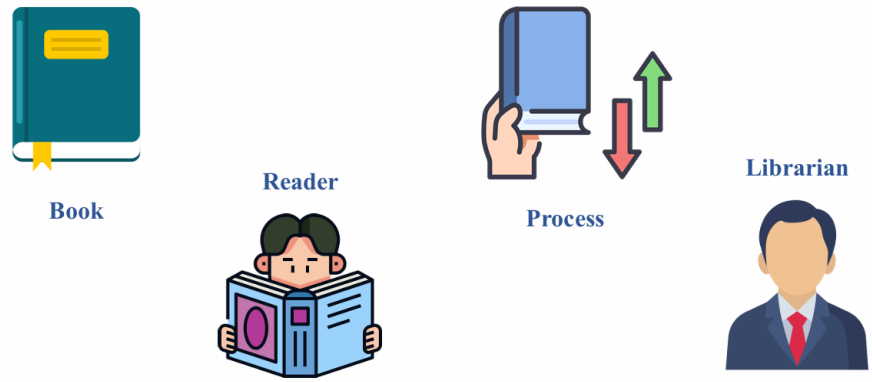
2 Động lực cho sự ra đời của OOP

Trong thực tế, chúng ta luôn tìm cách mô hình hóa các thực thể thực tế thành các đối tượng số hóa. Từ nhu cầu quản lý và tổ chức các thực thể này, lập trình hướng đối tượng ra đời để đơn giản hóa và tối ưu hóa việc xây dựng phần mềm.

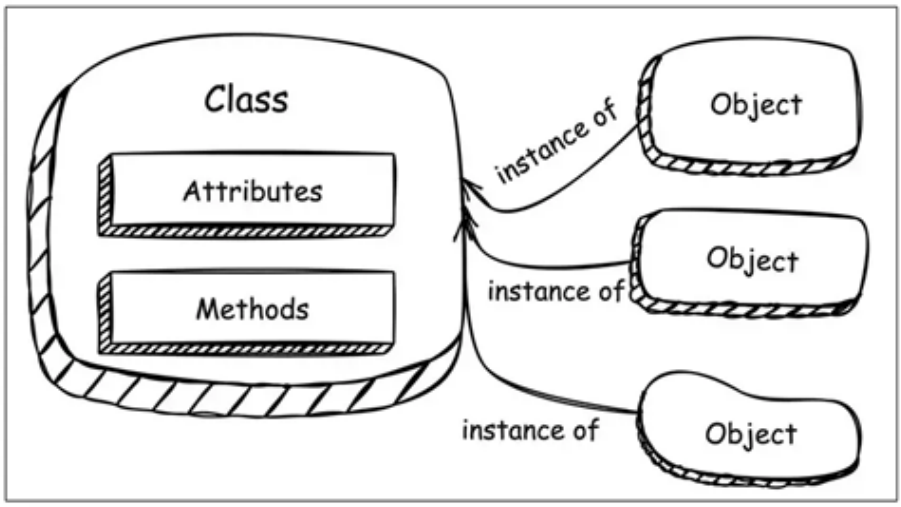
Hãy tưởng tượng một thế giới mạng xã hội, nơi mỗi người dùng chính là một đối tượng (object). Những thông tin như ngày sinh, giới tính, số điện thoại... được xem là thuộc tính (attributes) của người dùng đó. Còn các hành vi như đăng bài, thích (like), chia sẻ (share), bình luận, kết bạn... chính là những phương thức (methods) – tức là hành động mà đối tượng đó có thể thực hiện. Đây chính là cách mà lập trình hướng đối tượng mô hình hóa và tổ chức các thực thể trong thế giới số.



Tương tự:
Giả sử bạn là một người quản lý thư viện. Trong đó, sách và người đọc chính là những đối tượng (objects), còn các quy trình như mượn sách, trả sách, đăng ký thẻ được xem là phương thức (methods) – tức là hành vi của các đối tượng. Để có thể quản lý hiệu quả tất cả các đối tượng và hành vi trong hệ thống thư viện này, ta cần một cách tổ chức logic và linh hoạt. Chính từ nhu cầu đó, lập trình hướng đối tượng (OOP) ra đời – như một phương pháp giúp mô hình hóa thế giới thực thành các thành phần trong phần mềm một cách rõ ràng và dễ mở rộng.



3 Class và Object



3.1 Class (Lớp)

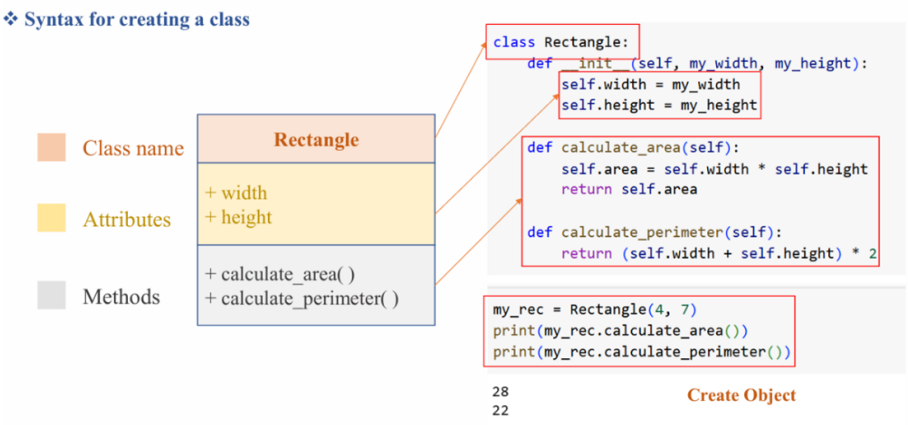
Class như một bản vẽ kỹ thuật hoặc template, dùng để định nghĩa thuộc tính (attributes) và phương thức (methods) của các đối tượng.

- **Attributes:** định nghĩa các thông tin, đặc điểm cũng như các thuộc tính của Object.
- **Method**: định nghĩa các hành vi, phương thức cũng như các hành động thường có của Object

3.2 Object (Đối tượng)

Object là một thực thể cụ thể được tạo ra từ class, chứa các giá trị thuộc tính cụ thể và khả năng thực hiện các phương thức đã định nghĩa.

4 Cách tạo một Class



Một class diagram bao gồm: tên class, attributes, và methods. Sau khi khởi tạo, ta tạo ra một object là instance của class vừa tạo.

Constructor (`__init__`): được dùng để tạo và gán giá trị ban đầu cho các thuộc tính (attributes) của đối tượng.Nói một cách đơn giản, constructor giống như bản thiết kế ban đầu giúp ta xác định: "Khi tạo ra một đối tượng mới, nó sẽ có những thông tin gì?"

Self keyword: Là tham chiếu đến instance cụ thể của class.

Ví dụ:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

my_rec = Rectangle(4, 7)
print(my_rec.area())  # Output: 28
```

Ta hình dung "self" là một vùng nhớ. Khi lớp được gọi, nó tham chiếu đến vùng nhớ gồm các thuộc tính. Khi đối số được truyền vào constructor (4 và 7), nó được gán cho các thuộc tính lần lượt là `self.width` và `self.height`, và được lưu trữ tại vùng nhớ đó. Khi gọi `my_rec` cùng với method, nó tham chiếu đến vùng nhớ `self` đã được gán thuộc tính, và `self.area` lúc này sẽ trở thành `my_rec.area`, từ đó ta tính được diện tích của đối tượng `my_rec`.

Phương thức đặc biệt: `__call__`

Trong Python, `__call__` là một **phương thức đặc biệt** (giống như `__init__`, `__str__`, v.v.) được sử dụng khi một đối tượng cần hành xử giống như một hàm. Nếu một lớp định nghĩa `__call__`, thì các *instance* của lớp đó có thể được gọi như một hàm thực sự.

Phương thức `__call__()` thường được dùng trong ba tình huống phổ biến:

1. **Function factory**: Tạo ra các đối tượng có thể xử lý logic như một hàm.

```
class SayHi:
    def __init__(self, name):
        self.name = name

    def hello(self):
        print(f'Hello {self.name}')

    def __call__(self, prefix):
        print(f'{prefix} {self.name}')

obj = SayHi("Alice")
obj.hello()
obj("Hi")       # __call__ make it becomes a function → Output: Hi Alice
```

2. **Stateful function**: Hàm có thể ghi nhớ trạng thái bên trong.

```
class Counter:
    def __init__(self):
        self.count = 0

    def __call__(self):
        self.count += 1
        return self.count

counter = Counter()

print(counter())  # 1
print(counter())  # 2
print(counter())  # 3
```

   Mỗi lần gọi `counter()` đều ghi nhớ trạng thái trước đó và cộng dồn lên, không giống như các phương thức thông thường vốn không lưu trạng thái giữa các lần gọi.

3. **Decorator hoặc Callback handler**: (nâng cao cần tìm hiểu thêm).

Phần II: Các tính chất cơ bản trong Object-Oriented Programming

5 Delegation (Ủy quyền)

Delegation (ủy quyền) trong lập trình hướng đối tượng là một kỹ thuật trong đó một đối tượng ủy thác trách nhiệm thực hiện một hành vi cụ thể cho một đối tượng khác. Thay vì kế thừa trực

tiếp hoặc xử lý toàn bộ logic nội bộ, đối tượng sẽ gọi đến phương thức của một thành phần bên trong để thực hiện nhiệm vụ. Kỹ thuật này giúp tách biệt trách nhiệm giữa các lớp, dễ bảo trì và mở rộng hệ thống và tăng tính tái sử dụng của mã nguồn.

```
class Date:
    def __init__(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year

    def get_day(self):
        return self.__day

    def get_month(self):
        return self.__month

    def get_year(self):
        return self.__year

    def describe(self):
        print(f"Day: {self.get_day()} - Month: {self.get_month()} - Year: {self.
    get_year()}")

class Person:
    def __init__(self, name, date):
        self.__name = name
        self.__date_of_birth = date  # Delegation: store a Date object

    def describe(self):
        print(f"Name: {self.__name}")
        self.__date_of_birth.describe()  # Delegate to Date


date = Date(18, 6, 2025)
peter = Person("Peter", date)
peter.describe()
```

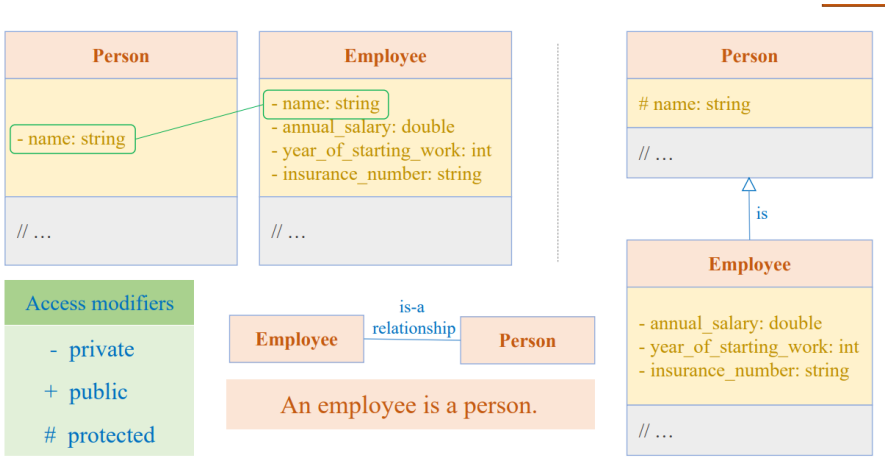
Hãy tưởng tượng bạn muốn "outsource" (thuê ngoài) một công việc thay vì tự thực hiện – ví dụ như mô tả ngày sinh. Thay vì để lớp `Person` tự xử lý chi tiết ngày tháng năm sinh, ta ủy quyền công việc đó cho lớp `Date`. Điều này giúp mã nguồn gọn gàng hơn và phân tách rõ trách nhiệm giữa các lớp.

6 Inheritance (Kế thừa)

Inheritance trong lập trình hướng đối tượng OOP là một cơ chế xây dựng class mới dựa trên các class đã có. Các class kế thừa sẽ bao gồm toàn bộ các attributes và methods từ base class (lớp cơ sở) hay parent class (lớp cha). Sử dụng Inheritance sẽ giúp các nhà phát triển tái sử dụng được các class đã có, giảm thiểu các duplication (sự trùng lặp) không cần thiết. Nó cũng giống như việc thay vì mua thêm một chiếc xe mới, thì ta dùng lại xe cũ của bố đế tiết kiệm hơn.

Trong class diagram, lớp con kế thừa từ lớp cha được biểu thị bằng mũi tên màu trắng. Khi đó mọi thuộc tính chung có thể bị lặp lại đều nằm ở lớp cha, còn lại các thuộc tính riêng cho từng

đối tượng con sẽ thuộc riêng từng đối tượng ấy.



Ví dụ minh họa:

```
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)          # Call parent constructor
        self.student_id = student_id

    def introduce(self):
        print(f"My name is {self.name}, ID: {self.student_id}")

s = Student("Alice", "S12345")
s.introduce()
```

Lớp `Student` kế thừa từ `Person`. Phương thức `super().__init__(name)` được sử dụng để gọi constructor của lớp cha vì ta không muốn việc khởi tạo thuộc tính `name` lặp lại ở cả cha và con. Lớp con còn có thêm thuộc tính riêng là `student_id`, và định nghĩa phương thức `introduce()` để hiển thị thông tin.

Trong lập trình hướng đối tượng, phạm vi truy cập của biến (*access modifier*) ảnh hưởng đến cách các lớp con sử dụng thuộc tính từ lớp cha:

- **Biến `public`** giống như tài sản công khai – bất kỳ đối tượng nào cũng có thể truy cập được.
- **Biến `private`** là thông tin tuyệt mật – chỉ lớp khai báo nó mới có quyền truy cập. Lớp con không thể "đọc nhật ký" hay can thiệp trực tiếp vào những gì cha mẹ không cho phép.
- **Biến `protected`** thì đặc biệt hơn – đây là những "di sản" được truyền lại, chỉ dành cho các lớp con và không ai khác ngoài hệ thống kế thừa được phép sử dụng.

Sự phân cấp này giúp bảo vệ dữ liệu và giữ cho hệ thống kế thừa hoạt động có tổ chức.

7 Abstraction (Trừu tượng)

Trong lập trình hướng đối tượng, @abstractmethod được sử dụng trong kế thừa như một cam kết hoặc mong muốn từ lớp cha dành cho lớp con. Lớp cha khai báo phương thức đó nhưng không tự triển khai, vì nó chỉ định hướng hành vi mong đợi. Việc hiện thực hóa chức năng được giao cho lớp con đảm nhận. Tưởng tượng lớp cha như cha mẹ đặt ra "ước muốn" – ví dụ: "Con cần phải biết tự giới thiệu bản thân", nhưng cha mẹ không thực hiện thay, mà mong lớp con tự định nghĩa và thực hiện điều đó theo cách riêng.

Ví dụ minh họa:

```
from abc import ABC, abstractmethod

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

class Square(Shape):
    def __init__(self, side):
        self.__side = side

    def compute_area(self):
        return self.__side * self.__side

square = Square(5)
print(square.compute_area())
```

Ở đây lớp `Shape()` mong muốn các lớp con của nó đều phải **bắt buộc** có phương thức tính diện tích. Nếu không, chương trình sẽ báo lỗi.

8 Encapsulation (Đóng gói)

Tính đóng gói giúp thông tin nội bộ của đối tượng và chỉ cho phép truy cập qua phương thức công khai (public methods). Điều này giúp bảo vệ dữ liệu và kiểm soát cách dữ liệu bị thay đổi.

Ví dụ minh họa:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # "__" indicates this is a private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
```

```
            print(f"Withdrew: {amount}")
        else:
            print("Invalid or insufficient funds.")

    def get_balance(self):
        return self.__balance

account = BankAccount("Alice", 1000)

account.deposit(500)              # OK
account.withdraw(300)             # OK
print(account.get_balance())      # Output: 1200

account.__balance = 100000        # Attempt to directly modify the balance (fails)
print(account.get_balance())      # Still 1200 - real balance not affected
```

9 Polymorphism (Đa hình)

Tính đa hình (Polymorphism) trong lập trình hướng đối tượng (OOP) cho phép một object có thể có nhiều hình dạng và hành vi khác nhau.

Tưởng tượng bạn có một chiếc remote điều khiển. Khi bạn bấm nút "Play", nó có thể:

- Phát nhạc nếu đang điều khiển máy nghe nhạc.
- Phát phim nếu đang điều khiển TV.
- Phát bài học nếu đang điều khiển app học online.

Tất cả đều là "Play", nhưng mỗi thiết bị hiểu và phản hồi khác nhau. Đó chính là bản chất của Polymorphism – một hành động, nhiều cách thực thi khác nhau tùy vào ngữ cảnh.

Polymorphism trong OOP được chia làm hai loại:

- **Static Polymorphism (đa hình tĩnh)**: là cơ chế định nghĩa lại các methods cùng tên, nhưng có thể khác số lượng hoặc kiểu của tham số. Còn gọi là *Method Overloading*.
- **Dynamic Polymorphism (đa hình động)**: là cơ chế định nghĩa lại các methods cùng tên, cùng tham số và kiểu trả về từ lớp cha. Còn gọi là *Method Overriding*.

Sự khác biệt chính:

- Static Polymorphism: xử lý tại thời điểm biên dịch (compile-time).
- Dynamic Polymorphism: xử lý tại thời điểm chạy (runtime).

Ghi chú quan trọng: Python là ngôn ngữ thông dịch (runtime-based), không cần khai báo kiểu biến. Kiểu dữ liệu được xác định khi chương trình chạy, không phải lúc biên dịch. Do đó, Python không có static polymorphism như Java hoặc C++. Mọi thứ xảy ra tại runtime.

Ví dụ về đa hình động:

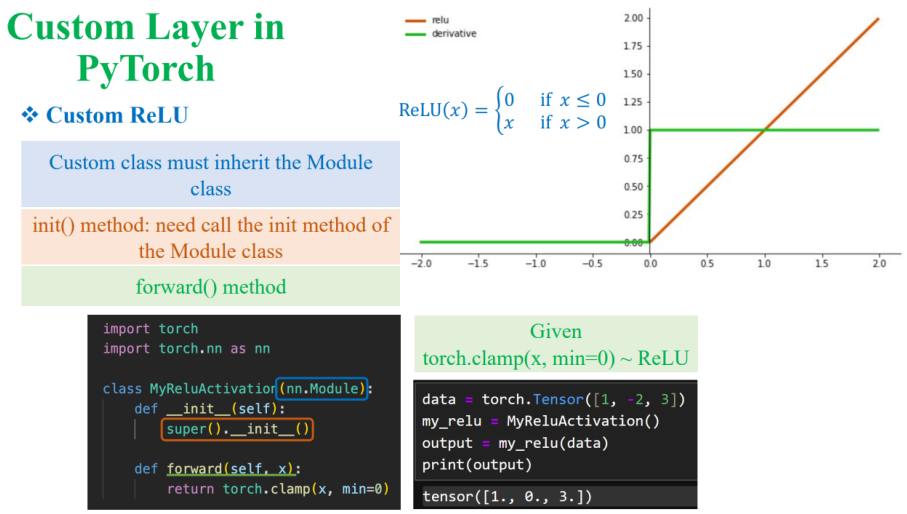
```python
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Gâu gâu"  # Vietnamese for "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meo meo"  # Vietnamese for "Meow!"

animals = [Dog(), Cat(), Animal()]

for a in animals:
    print(a.speak())  # Calls speak() appropriate to the actual object type
```

Dù gọi cùng một hàm `speak()` trên danh sách `animals`, kết quả sẽ:

- Trả về `"Some sound"` nếu là instance của `Animal`
- Trả về `"Gâu gâu"` nếu là `Dog`
- Trả về `"Meo meo"` nếu là `Cat`

Python sẽ xác định phiên bản phù hợp của phương thức `speak()` tại **thời điểm chạy chương trình (runtime)**, dựa trên kiểu thực tế của đối tượng. Đây chính là **method overriding** – khi lớp con định nghĩa lại phương thức của lớp cha với cùng tên và tham số.

Phần III: Ứng dụng Custom Layer trong PyTorch



Biến tạo ra cần phải kế thừa lớp Module (lớp này là lớp cha có sẵn trong PyTorch). Và ta cũng gọi `super().__init__()` trong constructor và method `forward()`