Python cho Khoa học dữ liệu

Bài 21: Các Khái niệm cơ bản của Hướng đối tượng trong Python

Hà Minh Tuấn
hmtuan@hcmus.edu.vn

Khoa Toán - Tin học
Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM

Ngày 29 tháng 10 năm 2025

Nội dung bài học

1. Khái niệm Lớp (Class) trong Python
2. Khái niệm Đối tượng trong python
3. Khái niệm Đóng gói (Encapsulation)
4. Kế thừa (Inheritance) trong Lập trình Hướng đối tượng
5. Đa hình (Polymorphism) trong Lập trình Hướng đối tượng
6. Trừu tượng hóa (Abstraction) trong Lập trình Hướng đối tượng
7. Object-Oriented Design (OOD)
8. Ví dụ minh họa: Class DataPreprocessor
9. Phân tích kiến trúc class DataPreprocessor

Khái niệm Lớp (Class) trong Python

- **Lớp (Class)** trong Python là một *khuôn mẫu* hoặc *bản thiết kế* để tạo ra các **đối tượng (Objects)**.
- Mỗi lớp mô tả tập hợp các **thuộc tính (Attributes)** và **phương thức (Methods)** dùng chung cho mọi đối tượng được tạo ra từ lớp đó.
- Lớp giúp tổ chức mã nguồn tốt hơn, hỗ trợ tái sử dụng và mở rộng trong lập trình hướng đối tượng.

Thành phần cơ bản của một Lớp

- **Từ khóa** `class`: dùng để định nghĩa một lớp.
- **Phương thức khởi tạo** `__init__()`: tự động được gọi khi tạo đối tượng, dùng để gán giá trị ban đầu.
- **Tham số** `self`: đại diện cho chính đối tượng đang được tạo hoặc gọi.
- **Thuộc tính (Attributes)**: đặc điểm mô tả đối tượng, thường được gán trong `__init__()`.
- **Phương thức (Methods)**: hành vi mà đối tượng có thể thực hiện.

Ví dụ: Định nghĩa một Lớp đơn giản

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")
```

Đoạn mã trên định nghĩa lớp Person với:
- Hai thuộc tính: name và age.
- Một phương thức: greet().

Tạo đối tượng từ Lớp

```
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

p1.greet()
p2.greet()
```

Kết quả:

```
Hello, my name is Alice and I'm 25 years old.
Hello, my name is Bob and I'm 30 years old.
```

Ý nghĩa và vai trò của Lớp trong Python

- **Trừu tượng hóa (Abstraction)**: Che giấu chi tiết phức tạp, chỉ cung cấp giao diện cần thiết.
- **Đóng gói (Encapsulation)**: Gộp dữ liệu và hành vi vào cùng một khối logic.
- **Tái sử dụng mã (Reusability)**: Dễ mở rộng và tái sử dụng trong các chương trình khác.
- **Kế thừa (Inheritance)**: Tạo lớp mới dựa trên lớp cũ, giảm trùng lặp mã.
- **Đa hình (Polymorphism)**: Cho phép cùng một phương thức hoạt động khác nhau trên các đối tượng khác nhau.

Khái niệm Đối tượng (Object) trong Python

- **Đối tượng (Object)** là một *thực thể cụ thể* được tạo ra từ **lớp (class)**.
- Mỗi đối tượng mang các **thuộc tính riêng** (dữ liệu cụ thể) và có thể **thực hiện các hành vi** (phương thức) được định nghĩa trong lớp.
- Có thể hiểu rằng:
  - **Lớp** là bản thiết kế (blueprint).
  - **Đối tượng** là sản phẩm cụ thể được tạo ra từ bản thiết kế đó.

Ví dụ: Tạo đối tượng từ lớp

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I'm {self.age} years old.")

# Tạo hai đối tượng từ lớp Person
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

# Gọi phương thức của đối tượng
p1.greet()
p2.greet()
```

Kết quả:

```
Hello, my name is Alice and I'm 25 years old.
```

Đặc điểm của đối tượng trong Python

- Mỗi đối tượng có **bộ nhớ riêng**, lưu trữ giá trị các thuộc tính độc lập với đối tượng khác.
- Đối tượng có thể **thực hiện các hành vi** thông qua việc gọi các phương thức.
- Python coi **mọi thứ đều là đối tượng**, kể cả số, chuỗi, danh sách, hàm, hoặc lớp.

Kiểm tra đối tượng và kiểu dữ liệu

```python
 # Kiểm tra kiểu của đối tượng
print(type(p1))
print(isinstance(p1, Person))

# Mọi thứ trong Python đều là object
print(isinstance(10, object))
print(isinstance("Hello", object))
```

Kết quả:

```
<class '__main__.Person'>
True
True
True
```

Tóm tắt: Mối quan hệ giữa Lớp và Đối tượng

- **Lớp (Class)**: Mô tả đặc điểm và hành vi chung.
- **Đối tượng (Object)**: Là phiên bản cụ thể của lớp, mang dữ liệu riêng biệt.
- Nhiều đối tượng có thể được tạo ra từ cùng một lớp.
- Ví dụ: Lớp `Person` có thể tạo ra nhiều đối tượng như `Alice`, `Bob`, `Charlie`.

Khái niệm Đóng gói (Encapsulation) trong Python

- **Encapsulation (Đóng gói)** là một trong bốn đặc trưng cơ bản của lập trình hướng đối tượng.
- Nó cho phép **ẩn giấu dữ liệu (data hiding)** và **bảo vệ thông tin nội bộ** của đối tượng khỏi việc truy cập trực tiếp từ bên ngoài.
- Thay vì thao tác trực tiếp với dữ liệu, người dùng chỉ có thể tương tác thông qua các **phương thức công khai (public methods)** do lập trình viên cung cấp.
- Mục tiêu của đóng gói là:
  - Ngăn chặn việc thay đổi dữ liệu trái phép.
  - Duy trì tính toàn vẹn của đối tượng.
  - Giúp mã dễ bảo trì, dễ mở rộng.

Cách triển khai Đóng gói trong Python

- Python không có từ khóa `private`, nhưng có quy ước đặt tên để thể hiện mức độ truy cập:
  - `public`: không có dấu gạch dưới (ai cũng truy cập được).
  - `_protected`: có một dấu gạch dưới, chỉ nên dùng trong nội bộ lớp và các lớp kế thừa.
  - `__private`: có hai dấu gạch dưới, Python sẽ đổi tên (name mangling) để ngăn truy cập trực tiếp.

Ví dụ: Sử dụng Đóng gói trong Python

```python
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # thuộc tính private

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. Remaining balance: {self.__balance}")
        else:
            print("Invalid withdrawal amount or insufficient funds.")
```

Minh họa truy cập và bảo vệ dữ liệu

```
def get_balance(self):
            return self.__balance
```

Lớp BankAccount đóng gói thông tin số dư (__balance)
và chỉ cho phép truy cập thông qua các phương thức công khai.

Minh họa truy cập và bảo vệ dữ liệu

```python
acc = BankAccount("Alice", 1000)
acc.deposit(500)
acc.withdraw(200)

# Truy cập trực tiếp thuộc tính private sẽ thất bại
print(acc.__balance)      # Gây lỗi AttributeError

# Truy cập đúng cách thông qua phương thức
print(acc.get_balance())  # 1300
```

```
Kết quả:
Deposited 500. New balance: 1500
Withdrew 200. Remaining balance: 1300
AttributeError: 'BankAccount' object has no attribute '__balance'
1300
```

Ý nghĩa của Đóng gói trong lập trình hướng đối tượng

- Giúp kiểm soát cách dữ liệu được truy cập và thay đổi.
- Tăng tính an toàn và ổn định của chương trình.
- Tạo giao diện rõ ràng giữa **nội bộ lớp** và **người sử dụng lớp**.
- Là bước đầu tiên để hướng tới lập trình theo nguyên tắc **information hiding** (ẩn thông tin).

Khái niệm Kế thừa

- **Inheritance (Kế thừa)** là cơ chế cho phép một lớp (class con) kế thừa các thuộc tính và phương thức của một lớp khác (class cha).
- Mục đích là **tái sử dụng mã nguồn** và **mở rộng chức năng** mà không cần viết lại từ đầu.
- Kế thừa giúp mô hình hóa quan hệ kiểu *"is-a"* (ví dụ: Mèo *là một* Động vật).

Kế thừa trong Python

- Trong Python, kế thừa được khai báo bằng cách truyền tên lớp cha vào trong ngoặc đơn khi định nghĩa lớp con.
- Cú pháp:

```
class ClassCon(ClassCha):
    code
```

- Lớp con có thể:
  - Dùng lại các thuộc tính và phương thức của lớp cha.
  - Ghi đè (override) phương thức của lớp cha.
  - Mở rộng thêm thuộc tính và phương thức mới.

Ví dụ về Kế thừa trong Python

```python
 # Lớp cha (ClassCha)
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

# Lớp con (ClassCon) kế thừa từ Animal
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Tạo đối tượng từ lớp con
dog = Dog("Buddy")
print(dog.name)      # Kế thừa thuộc tính từ lớp cha
print(dog.speak())   # Ghi đè phương thức speak()
```

Output:

Giải thích ví dụ

- **Animal** là lớp cha có thuộc tính `name` và phương thức `speak()`.
- **Dog** là lớp con kế thừa từ `Animal` và ghi đè lại phương thức `speak()`.
- Đối tượng dog có thể truy cập cả thuộc tính `name` từ lớp cha và phương thức `speak()` đã ghi đè.
- Điều này thể hiện sức mạnh của kế thừa: *tái sử dụng và mở rộng hành vi của lớp cha*.

Ví dụ Kế thừa Đa cấp (Multilevel Inheritance)

```
class Animal:
    def move(self):
        print("Moving...")

class Mammal(Animal):
    def has_fur(self):
        print("Has fur")

class Dog(Mammal):
    def bark(self):
        print("Woof!")

dog = Dog()
dog.move()     # Từ Animal
dog.has_fur() # Từ Mammal
dog.bark()     # Từ Dog
```

Output:

Moving...

Khái niệm Đa hình (Polymorphism)

- **Polymorphism (Đa hình)** là khả năng cho phép các đối tượng thuộc các lớp khác nhau có thể được xử lý thông qua cùng một giao diện.
- Nói cách khác, cùng một phương thức có thể có **hành vi khác nhau** tùy thuộc vào đối tượng đang gọi nó.
- Mục tiêu chính của đa hình là **tăng tính linh hoạt và khả năng mở rộng của chương trình**.

Đa hình trong Python

- Python hỗ trợ đa hình một cách tự nhiên nhờ vào đặc điểm **dynamic typing** (kiểu động).
- Nghĩa là bạn không cần khai báo kiểu dữ liệu cụ thể; chỉ cần các đối tượng có phương thức hoặc thuộc tính giống nhau, chúng có thể được sử dụng thay thế nhau.
- Đa hình trong Python được thực hiện qua:
  - Ghi đè phương thức (method overriding)
  - Dùng chung tên phương thức cho các lớp khác nhau

Ví dụ cơ bản về Đa hình trong Python

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

# Hàm có thể nhận bất kỳ đối tượng nào có phương thức speak()
def make_animal_speak(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_animal_speak(dog)
make_animal_speak(cat)
```

Output:

Giải thích ví dụ

- Hàm `make_animal_speak()` không cần biết đối tượng là `Dog` hay `Cat`.
- Chỉ cần đối tượng đó có phương thức `speak()`, hàm sẽ gọi được.
- Đây chính là biểu hiện của **đa hình động** (dynamic polymorphism): hành vi được quyết định tại thời điểm chạy.

Ví dụ Đa hình thông qua Kế thừa

```python
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

animals = [Dog(), Cat()]
for a in animals:
    print(a.speak())
```

Output:

Woof!

Ý nghĩa của Đa hình

- **Tăng khả năng mở rộng:** Dễ dàng thêm lớp mới mà không cần thay đổi mã gốc.
- **Tái sử dụng mã:** Các lớp con có thể dùng chung giao diện của lớp cha.
- **Giảm phụ thuộc:** Hàm hoặc đối tượng có thể làm việc với nhiều loại dữ liệu khác nhau.
- Đa hình là một trong bốn trụ cột chính của OOP: *Encapsulation*, *Inheritance*, *Polymorphism*, *Abstraction*.

Khái niệm Trừu tượng hóa (Abstraction)

- **Abstraction (Trừu tượng hóa)** là quá trình ẩn đi những chi tiết không cần thiết và chỉ hiển thị những thông tin quan trọng của đối tượng.
- Mục tiêu của trừu tượng hóa là giúp người lập trình **tập trung vào bản chất** của đối tượng thay vì các chi tiết triển khai cụ thể.
- Nó giúp đơn giản hóa sự phức tạp trong lập trình hướng đối tượng, đặc biệt khi xây dựng các hệ thống lớn.

Trừu tượng hóa trong Python

- Trong Python, trừu tượng hóa được thực hiện thông qua **lớp trừu tượng (Abstract Class)** và **phương thức trừu tượng (Abstract Method)**.
- Các lớp trừu tượng được định nghĩa bằng cách kế thừa từ lớp `ABC` (Abstract Base Class) trong module `abc`.
- Một lớp trừu tượng không thể được khởi tạo trực tiếp, chỉ có thể được kế thừa.

Ví dụ về Trừu tượng hóa trong Python

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass  # Phương thức trừu tượng, lớp con phải định nghĩa lại

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

animals = [Dog(), Cat()]
for a in animals:
    print(a.speak())
```

Giải thích ví dụ

- Lớp `Animal` là lớp trừu tượng, chứa phương thức trừu tượng `speak()`.
- Các lớp con (`Dog`, `Cat`) bắt buộc phải cài đặt lại phương thức này.
- Nếu không định nghĩa lại, chương trình sẽ báo lỗi khi tạo đối tượng.
- Điều này đảm bảo rằng tất cả các lớp con đều tuân theo **một giao diện chung**.

Ý nghĩa của Trừu tượng hóa

- **Giảm độ phức tạp:** Giúp lập trình viên tập trung vào những chức năng cốt lõi thay vì chi tiết triển khai.
- **Tăng tính bảo trì:** Khi cần thay đổi, chỉ cần điều chỉnh trong lớp con mà không ảnh hưởng đến phần còn lại.
- **Tăng tính linh hoạt:** Cho phép mở rộng hệ thống dễ dàng bằng cách thêm các lớp mới kế thừa cùng một giao diện.
- Trừu tượng hóa là nền tảng cho **thiết kế module, giao diện (interface)** và các mô hình phần mềm linh hoạt.

Ví dụ minh họa thực tế

```python
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class CreditCardPayment(Payment):
    def pay(self, amount):
        return f"Paid {amount} using Credit Card."

class PayPalPayment(Payment):
    def pay(self, amount):
        return f"Paid {amount} using PayPal."
```

Ví dụ minh họa thực tế

```
payments = [CreditCardPayment(), PayPalPayment()]
for p in payments:
    print(p.pay(100))
```

```
Output:
Paid 100 using Credit Card.
Paid 100 using PayPal.
```

Khái niệm Object-Oriented Design (OOD)

- **Object-Oriented Design (OOD)** là quá trình thiết kế chương trình dựa trên các **đối tượng (objects)** có **thuộc tính** và **hành vi**.
- Mục tiêu:
  - Tổ chức mã nguồn theo hướng mô hình hóa bài toán thực tế.
  - Tăng khả năng **tái sử dụng**, **bảo trì**, và **mở rộng**.
  - Ẩn chi tiết xử lý nội bộ, chỉ cung cấp giao diện (interface) cần thiết.

Các bước chính của Object-Oriented Design

- **Bước 1:** Xác định mục tiêu và phạm vi của class.
- **Bước 2:** Xác định các đối tượng và trách nhiệm (responsibilities).
- **Bước 3:** Xác định thuộc tính (attributes) và phương thức (methods).
- **Bước 4:** Xác định mối quan hệ giữa các class.
- **Bước 5:** Cài đặt và kiểm thử.

Ví dụ: Thiết kế class tiền xử lý dữ liệu

```python
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder

class DataPreprocessor:
    def __init__(self, missing_strategy='mean', scaler_type='standard'):
        self.missing_strategy = missing_strategy
        self.scaler_type = scaler_type
        self.data = None
        self.scaler = StandardScaler()
        self.encoder = LabelEncoder()

    def load_data(self, df):
        self.data = df.copy()
        return self
```

Ví dụ (tiếp): Class DataPreprocessor

```
def handle_missing(self):
        if self.missing_strategy == 'mean':
            self.data = self.data.fillna(self.data.mean())
        elif self.missing_strategy == 'drop':
            self.data = self.data.dropna()
        return self
```

Ví dụ (tiếp): Class DataPreprocessor

```
def encode_categorical(self, columns):
        for col in columns:
            self.data[col] = self.encoder.fit_transform(self.data[col])
        return self

    def scale_features(self, columns):
        self.data[columns] = self.scaler.fit_transform(self.data[columns])
        return self

    def get_processed_data(self):
        return self.data
```

Class DataPreprocessor hỗ trợ xử lý dữ liệu theo pipeline:

- Nạp dữ liệu
- Xử lý giá trị thiếu
- Mã hóa biến phân loại
- Chuẩn hóa biến số
- Trả về dữ liệu đã xử lý

Ví dụ sử dụng class DataPreprocessor

```
df = pd.DataFrame({
    'age': [25, 30, None, 45],
    'income': [50000, 60000, 55000, None],
    'gender': ['M', 'F', 'M', 'F']
})

processor = DataPreprocessor(missing_strategy='mean')
processed = (processor
             .load_data(df)
             .handle_missing()
             .encode_categorical(['gender'])
             .scale_features(['age', 'income'])
             .get_processed_data())

print(processed)
```

Chương trình nạp dữ liệu, xử lý giá trị thiếu, mã hóa giới tính và chuẩn hóa các cột numeric.

Kiến trúc tổng thể

- Class `DataPreprocessor` nằm trong tầng **Preprocessing Layer** của pipeline Machine Learning.
- Pipeline tổng quát:
  - `DataLoader` – nạp dữ liệu
  - `DataPreprocessor` – làm sạch, mã hóa, chuẩn hóa
  - `FeatureSelector` – chọn đặc trưng
  - `ModelTrainer` – huấn luyện mô hình

Vai trò của class DataPreprocessor

- **Làm sạch dữ liệu**: xử lý giá trị thiếu, loại bỏ dữ liệu lỗi.
- **Mã hóa dữ liệu phân loại**: chuyển giá trị text sang dạng số.
- **Chuẩn hóa dữ liệu số**: đưa về cùng thang đo.
- **Quản lý pipeline xử lý**: cho phép nối chuỗi các bước.
- **Lưu giữ trạng thái**: giữ dữ liệu gốc và cấu hình xử lý.

Thuộc tính (Attributes)

- `data`: *pandas DataFrame* – dữ liệu đang xử lý.
- `missing_strategy`: cách xử lý giá trị thiếu (`"mean"`, `"drop"`...).
- `scaler_type`: loại chuẩn hóa (`"standard"`, `"minmax"`...).
- `scaler`: đối tượng chuẩn hóa (VD: `StandardScaler()`).
- `encoder`: đối tượng mã hóa (VD: `LabelEncoder()`).
- `log`: lưu lại các bước xử lý (tùy chọn).

Hành vi (Methods)

- `__init__()`: khởi tạo đối tượng và cấu hình xử lý.
- `load_data(df)`: nạp dữ liệu vào class.
- `handle_missing()`: xử lý giá trị thiếu.
- `encode_categorical(columns)`: mã hóa biến phân loại.
- `scale_features(columns)`: chuẩn hóa biến số.
- `get_processed_data()`: trả về dữ liệu đã xử lý.
- `summary()`: hiển thị thông tin log (tùy chọn).

Quan hệ giữa các class

- **Composition:** chứa các đối tượng con như `Encoder`, `Scaler`.
- **Aggregation:** có thể dùng `Logger` để ghi log xử lý.
- **Dependency:** phụ thuộc vào `pandas`, `sklearn`.

Tổng kết kiến trúc OOP của class DataPreprocessor

- **Tên class:** `DataPreprocessor`.
- **Trách nhiệm chính:** Chuẩn bị dữ liệu sạch, sẵn sàng cho huấn luyện.
- **Thuộc tính:** `data`, `missing_strategy`, `scaler_type`, `encoder`, `scaler`.
- **Hành vi:** `load_data()`, `handle_missing()`, `encode_categorical()`, `scale_features()`, `get_processed_data()`.
- **Quan hệ:** Có thể mở rộng bằng các class con (Encoder, Scaler, Logger).
- **Ưu điểm:** Mở rộng dễ, hỗ trợ method chaining, rõ ràng về logic xử lý.