AI VIET NAM – AIO2024

OOP Cơ Bản Dùng Python

Khanh Duong Quang-Vinh Dinh

Ngày 15 tháng 6 năm 2024

Abstract

Trong lập trình hướng đối tượng (OOP), việc sử dụng classes và objects đóng vai trò nền tảng, giúp cho mã nguồn được tổ chức và quản lý một cách hiệu quả. Python, với khả năng hỗ trợ mạnh mẽ cho OOP, cho phép ta mô hình hóa các thực thể trong thế giới thực thành các đối tượng với những thuộc tính và phương thức riêng biệt. Trong bài học này, chúng ta sẽ cùng nhau tìm hiểu những kiến thức cơ bản và quan trọng nhất về Classes và Objects, cùng với đó là những thao tác liên quan đến tính Kế thừa (Inheritance) trong OOP, một tính chất giúp cho những hệ thống được xây dựng dựa trên Python có khả năng tái sử dụng và mở rộng dễ dàng.

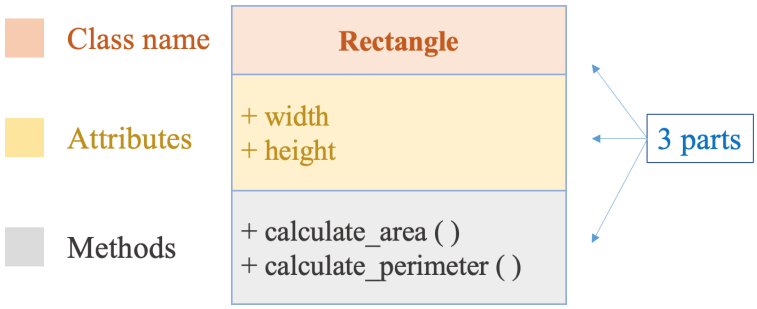
1. Classes and Objects

1.1. Khái niệm về Classes và Objects

- Classes là khuôn mẫu (template) để tạo ra các objects. Một class định nghĩa các thuộc tính (attributes) và phương thức (methods) mà objects của nó sẽ có.
- Objects là các thể hiện (instance) của classes, tức là mỗi object được tạo ra từ một class sẽ có các thuộc tính và phương thức được định nghĩa trong class đó.

1.2. Class Diagram

- Class Diagram là một loại sơ đồ thuộc về UML (Unified Modeling Language) được sử dụng để biểu diễn cấu trúc và mối quan hệ giữa các lớp trong một hệ thống phần mềm.
- Một Class Diagram bao gồm: tên của class, các thuộc tính của class, các phương thức của class.



Hình 1: Thành phần của một Class Diagram.

1.3. Cách định nghĩa một Class

Để định nghĩa một Class, ta thực hiện khai báo các thuộc tính của class bên trong phương thức **__init__()** (hay còn gọi là constructor, một phương thức đặc biệt trong Class Python sẽ được mô tả kỹ hơn trong mục 1.5).
Các phương thức của class được định nghĩa bằng cách nhận biến đặc biệt **self** làm đối số đầu tiên (một đối số được sử dụng để tham chiếu đến đối tượng hiện tại), và khởi tạo tương tự như các hàm bình thường.

```
# Create a class
class Rectangle:
    def __init__(self, my_width, my_height):
        self.width = my_width
        self.height = my_height

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

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

```

1.4. Cách tạo một Object từ một Class

Để khởi tạo một object thuộc một class, ta gọi tên class và kèm theo các giá trị tương ứng so với các biến đã được định nghĩa trong phương thức **__init__**.

```
# Create a object and
my_rec = Rectangle(4, 7)

# Call the attributes of the object
print(my_rec.width)                     # Output: 4
print(my_rec.height)                    # Output: 7

# Call the methods of the object
print(my_rec.calculate_area())          # Output: 28
print(my_rec.calculate_perimeter())     # Output: 22

```

1.5. Các phương thức đặc biệt

a. __init__()
Đây là phương thức khởi tạo (constructor), được gọi tự động mỗi khi một object mới của class được tạo ra.

```python
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(10)
print(obj.value)  # Output: 10

```

b. __str__()
Phương thức này được sử dụng để định nghĩa chuỗi đại diện của object, thường được sử dụng bởi hàm print.

```python
class MyClass:
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f"MyClass with value {self.value}"

obj = MyClass(10)
print(obj)  # Output: MyClass with value 10

```

c. __len__()
Phương thức này được sử dụng để trả về độ dài (length) của object, thường được sử dụng bởi hàm len.

```python
class MyCollection:
    def __init__(self, items):
        self.items = items

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

collection = MyCollection([1, 2, 3, 4])
print(len(collection))  # Output: 4

```

d. __call__()
Phương thức này cho phép một object có thể được gọi như một hàm.

```python
class Adder:
    def __init__(self, value):
        self.value = value

    def __call__(self, x):
        return self.value + x

add_five = Adder(5)
print(add_five(10))  # Output: 15

```

1.6. Sử dụng object của Class này làm thuộc tính của Class khác

Thuộc tính của một class có thể là bất kỳ đối tượng nào, bao gồm các biến có kiểu dữ liệu nguyên thủy (*integer, float, boolean,...*), các iterable (*list, tuple, string...*), hay thậm chí cả object của một class đã được khai báo trước đó. Trong ví dụ này, thuộc tính **birth** trong class **Person** nhận giá trị là một *object* được khởi tạo từ class **Date** đã được định nghĩa trước đó.

```python
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    def __call__(self):
        return f"{self.day:02d}/{self.month:02d}/{self.year}"

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

    def info(self):
        print(f"Name: {self.name} - Birth: {self.birth()}")

name = "Isaac Newton"
birth = Date(4, 1, 1643)
physicist = Person(name, birth)
physicist.info()
# Output: Name: Isaac Newton - Birth: 04/01/1643

```

2. Inheritance

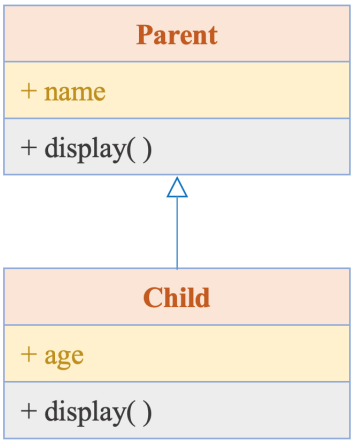
Kế thừa là một trong những tính năng mạnh mẽ nhất của lập trình hướng đối tượng. Nó cho phép một lớp (class) có thể kế thừa các thuộc tính (attributes) và phương thức (methods) của một lớp khác. Lớp kế thừa được gọi là lớp con (subclass), còn lớp bị kế thừa được gọi là lớp cha (superclass).

2.1. Định nghĩa Kế thừa

Lớp con có thể kế thừa tất cả các thuộc tính và phương thức từ lớp cha. Điều này giúp tái sử dụng mã nguồn và tạo ra mối quan hệ cha-con giữa các lớp.

Trong hình 2, quan hệ Kế thừa được mô tả bằng Class Diagram. Trong, đó, **Parent** là superclass, **Child**

là subclass, được kết nối với bằng mũi tên có đầu tam giác rỗng, trỏ vào class **Parent**, ngụ ý rằng class **Parent** là class cha của class **Child**.



Hình 2: Quan hệ kế thừa được mô tả bằng Class Diagram.

Ví dụ dưới đây mô tả cú pháp khởi tạo class kế thừa. Trong đó, class cha là Parent được gọi trong dấu ngoặc ngay khi class con là Child được định nghĩa. Phương thức **__init__()** của class cha cũng được gọi ở class con thông qua hàm **super()**.

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

    def display(self):
        print(f"Parent Name: {self.name}")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Gọi hàm khởi tạo của lớp cha
        self.age = age

    def display(self):
        super().display()  # Gọi phương thức của lớp cha
        print(f"Child Age: {self.age}")

# Tạo object của lớp Child
child = Child("Alice", 20)
child.display()

# Output:
# Parent Name: Alice
# Child Age: 20

```

2.2. Access Modifiers

Access Modifiers trong Python là cách để kiểm soát mức độ truy cập và sửa đổi các thuộc tính (attributes) và phương thức (methods) của một class. Mặc dù Python không có các từ khóa cụ thể như **private**, **protected**, **public** giống như một số ngôn ngữ lập trình khác (ví dụ: Java, C++), nhưng nó cung cấp cách thức để đạt được điều này thông qua quy ước đặt tên.

a. Public members

Các public members có thể được truy cập từ bất kỳ đâu, cả bên trong và bên ngoài class. Mặc định, tất cả các thuộc tính và phương thức trong Python đều là public.

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def display(self):
        print(self.value)

obj = MyClass(10)
print(obj.value)  # Truy cập thuộc tính công khai
obj.display()     # Gọi phương thức công khai

```

b. Protected members

Các protected members chỉ nên được truy cập bên trong class và các lớp con của nó. Để khai báo một member là protected, chúng ta sử dụng một dấu gạch dưới _ trước tên của thuộc tính hoặc phương thức.

```
class MyClass:
    def __init__(self, value):
        self._value = value  # Protected attribute

    def _display(self):  # Protected method
        print(self._value)

class SubClass(MyClass):
    def show(self):
        self._display()  # Truy cập phương thức được bảo vệ từ lớp con

obj = SubClass(20)
obj.show()  # Output: 20

```

b. Private members

Các private members chỉ có thể được truy cập bên trong class nơi chúng được định nghĩa. Để khai báo một member là private, chúng ta sử dụng hai dấu gạch dưới __ trước tên của thuộc tính hoặc phương thức.

```python
class MyClass:
    def __init__(self, value):
        self.__value = value  # Private attribute

    def __display(self):  # Private method
        print(self.__value)

    def public_method(self):
        self.__display()  # Truy cập phương thức riêng tư từ bên trong class

obj = MyClass(30)
obj.public_method()  # Output: 30
# print(obj.__value)  # Lỗi: AttributeError
# obj.__display()     # Lỗi: AttributeError

```

2.3. Override

Override là một khái niệm quan trọng trong lập trình hướng đối tượng, cho phép một lớp con cung cấp một triển khai cụ thể của một phương thức đã được định nghĩa trong lớp cha. Ghi đè phương thức cho phép bạn thay đổi hoặc mở rộng hành vi của phương thức kế thừa từ lớp cha trong lớp con.

Để ghi đè một phương thức trong lớp con, bạn chỉ cần định nghĩa lại phương thức đó với cùng tên và cùng tham số trong lớp con.

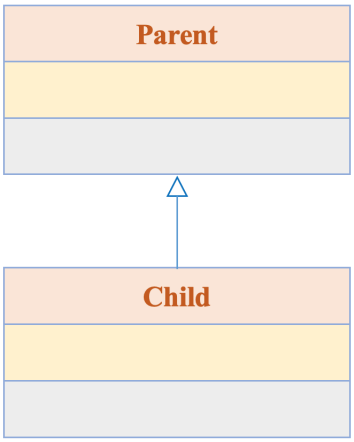
Trong ví dụ này, phương thức *display()* được ghi đè ở subclass Child.

```python
class Parent:
    def display(self):
        print("Display from Parent")

class Child(Parent):
    def display(self):
        print("Display from Child")

# Tạo object từ lớp con
child = Child()
child.display()  # Output: Display from Child

```

2.4. Các loại Kế thừa trong Python

a. Single Inheritance (Kế thừa đơn)

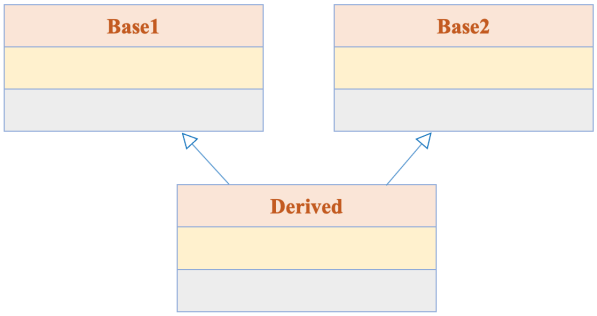
Kế thừa đơn là khi một lớp con kế thừa từ một lớp cha duy nhất.



Hình 3: Kế thừa đơn.

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

    def display(self):
        print(f"Parent Name: {self.name}")

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display(self):
        super().display()
        print(f"Child Age: {self.age}")

# Tạo object từ lớp con
child = Child("Alice", 20)
child.display()

# Output:
# Parent Name: Alice
# Child Age: 20

```

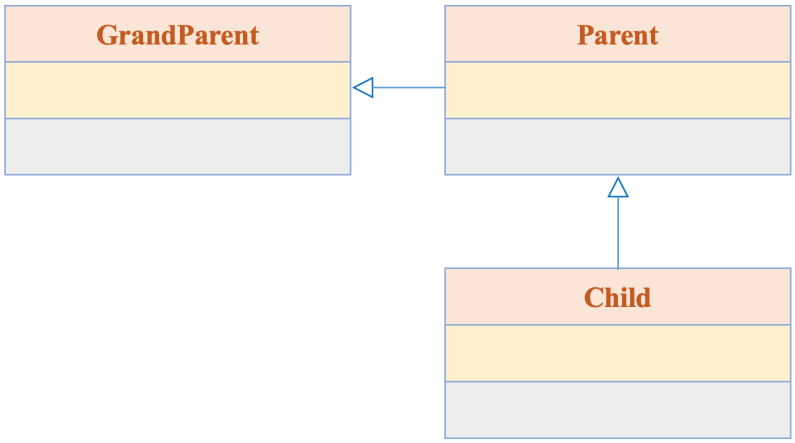
b. Multiple Inheritance (Kế thừa đa kế)
Kế thừa đa kế là khi một lớp con kế thừa từ nhiều lớp cha.



Hình 4: Kế thừa đa kế.

```python
class Base1:
    def __init__(self):
        self.str1 = "Base1"
        print("Base1 Initialized")

class Base2:
    def __init__(self):
        self.str2 = "Base2"
        print("Base2 Initialized")

class Derived(Base1, Base2):
    def __init__(self):
        Base1.__init__(self)
        Base2.__init__(self)
        print("Derived Initialized")

    def display(self):
        print(self.str1, self.str2)

# Tạo object từ lớp Derived
obj = Derived()
obj.display()

# Output:
# Base1 Initialized
# Base2 Initialized
# Derived Initialized
# Base1 Base2

```

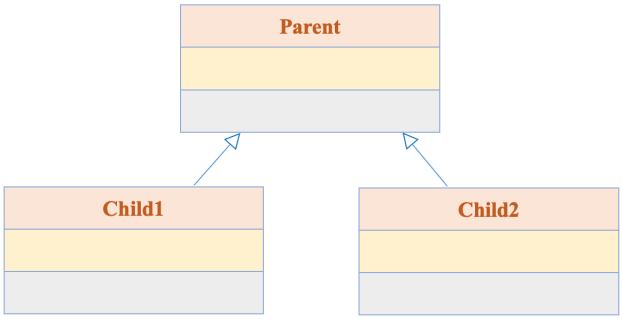
c. Multilevel Inheritance (Kế thừa đa cấp)
Kế thừa đa cấp là khi một lớp kế thừa từ lớp cha, và lớp cha đó lại kế thừa từ một lớp khác.



Hình 5: Kế thừa đa cấp.

```python
class GrandParent:
    def __init__(self, grandparent_name):
        self.grandparent_name = grandparent_name

    def display_grandparent(self):
        print(f"GrandParent Name: {self.grandparent_name}")

class Parent(GrandParent):
    def __init__(self, grandparent_name, parent_name):
        super().__init__(grandparent_name)
        self.parent_name = parent_name

    def display_parent(self):
        print(f"Parent Name: {self.parent_name}")

class Child(Parent):
    def __init__(self, grandparent_name, parent_name, child_name):
        super().__init__(grandparent_name, parent_name)
        self.child_name = child_name

    def display_child(self):
        print(f"Child Name: {self.child_name}")

# Tạo object từ lớp con
child = Child("George", "John", "Alice")
child.display_grandparent()  # Output: GrandParent Name: George
child.display_parent()       # Output: Parent Name: John
child.display_child()        # Output: Child Name: Alice

```

d. Hierarchical Inheritance (Kế thừa phân cấp)
Kế thừa phân cấp là khi một lớp cha có nhiều lớp con kế thừa từ nó.



Hình 6: Kế thừa phân cấp.

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

    def display(self):
        print(f"Parent Name: {self.name}")

class Child1(Parent):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def display(self):
        super().display()
        print(f"Child1 Age: {self.age}")

class Child2(Parent):
    def __init__(self, name, grade):
        super().__init__(name)
        self.grade = grade

    def display(self):
        super().display()
        print(f"Child2 Grade: {self.grade}")

# Tạo objects từ các lớp con
child1 = Child1("Alice", 20)
child2 = Child2("Bob", "A")

child1.display()
# Output:
# Parent Name: Alice
# Child1 Age: 20

child2.display()
# Output:
# Parent Name: Bob
# Child2 Grade: A

```

6. Bài Tập

Câu hỏi 1: Khái niệm nào sau đây mô tả chính xác nhất về một class trong Python?

a. Một biến
b. Một hàm
c. Một khuôn mẫu để tạo ra các đối tượng
d. Một phương thức đặc biệt

Câu hỏi 2: Trong sơ đồ class (class diagram), yếu tố nào sau đây không phải là thành phần của class?

a. Tên class
b. Thuộc tính (attributes)
c. Phương thức (methods)
d. Tham số (parameters)

Câu hỏi 3: Câu lệnh nào sau đây được sử dụng để định nghĩa một class trong Python?

a. def MyClass:
b. class MyClass:
c. create MyClass:
d. define MyClass:

Câu hỏi 4: Câu lệnh nào sau đây dùng để tạo một object từ class **MyClass**?

a. obj = MyClass.create()
b. obj = MyClass.new()
c. obj = MyClass()
d. obj = new MyClass()

Câu hỏi 5: Phương thức đặc biệt nào được sử dụng để khởi tạo các thuộc tính của một object trong Python?

a. __start__
b. __init__
c. __create__
d. __new__

Câu hỏi 6: Kế thừa (Inheritance) trong Python là gì?

a. Khả năng một class kế thừa các thuộc tính và phương thức từ một class khác
b. Khả năng một class thay đổi giá trị của các biến
c. Khả năng một class tạo ra các đối tượng mới
d. Khả năng một class gọi các hàm từ class khác

Câu hỏi 7: Access modifier nào sau đây được sử dụng để chỉ ra một thuộc tính hoặc phương thức là riêng tư (private) trong Python?

a. @
b. _
c. __
d. #

Câu hỏi 8: Điều nào sau đây là đúng về phương thức override trong Python?

a. Nó cho phép lớp con kế thừa phương thức từ lớp cha mà không cần thay đổi gì

b. Nó cho phép lớp con thay đổi hoặc mở rộng hành vi của phương thức kế thừa từ lớp cha

c. Nó ngăn cản lớp con kế thừa phương thức từ lớp cha

d. Nó yêu cầu lớp con phải gọi phương thức của lớp cha mà không được thay đổi

Câu hỏi 9: Loại kế thừa nào sau đây mô tả đúng khi một lớp con kế thừa từ nhiều lớp cha?

a. Single Inheritance

b. Multiple Inheritance

c. Multilevel Inheritance

d. Hierarchical Inheritance

Câu hỏi 10: Trong Python, loại kế thừa nào xảy ra khi một lớp kế thừa từ một lớp cha, và lớp cha đó lại kế thừa từ một lớp khác?

a. Single Inheritance

b. Multiple Inheritance

c. Multilevel Inheritance

d. Hierarchical Inheritance