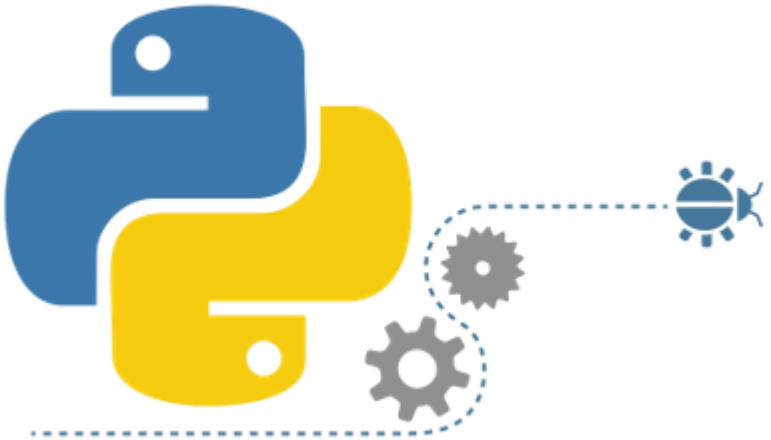
LẬP TRÌNH TRÍ TUỆ NHÂN TẠO

THS. VƯƠNG XUÂN CHÍ

VXCHI@NTT.EDU.VN

0903 270 567

CHƯƠNG 6

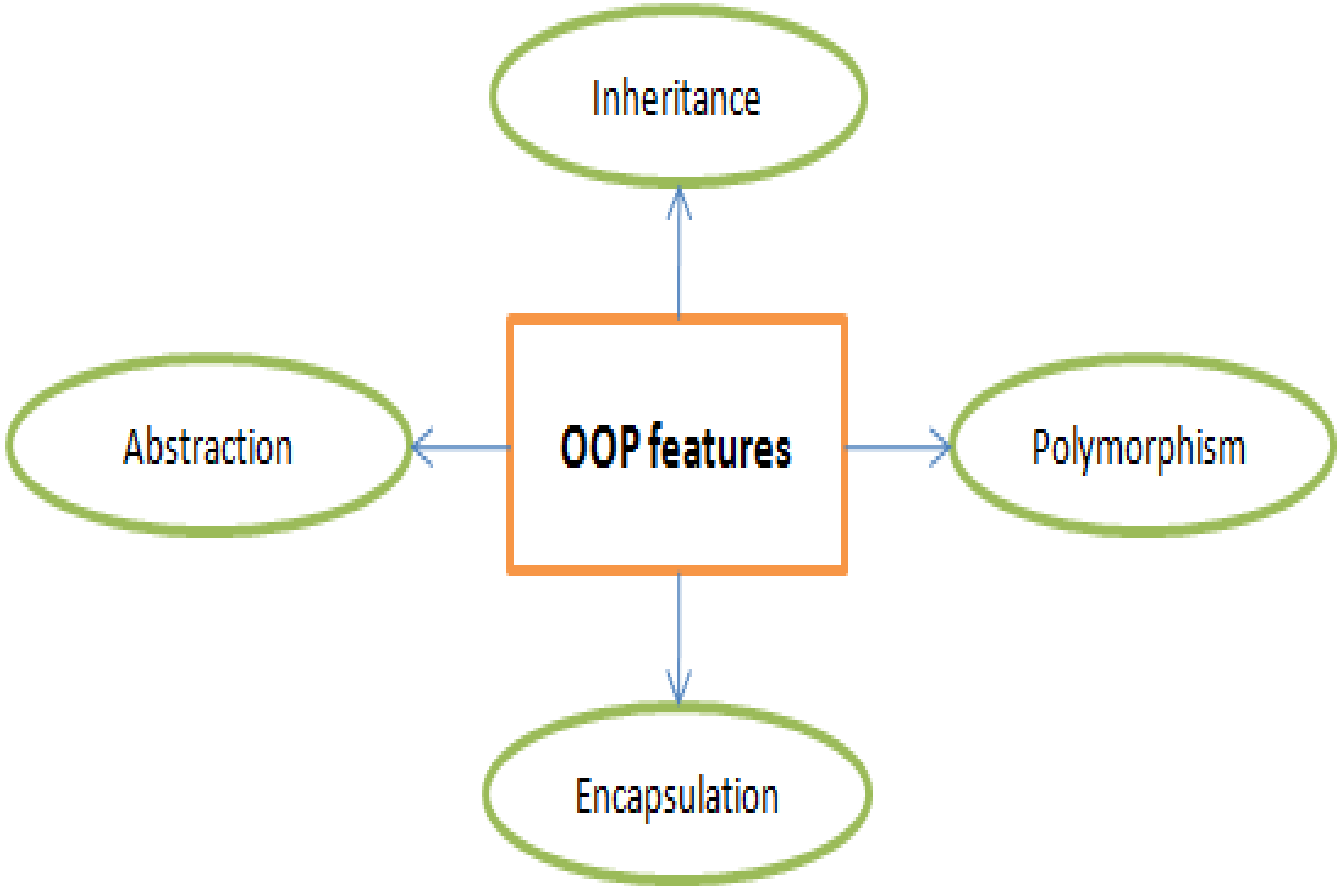


OOPs TRONG PYTHON

- Đối tượng (object)
- Lớp (class)
- Phương thức (method)
- *Kế thừa*
- *Đa hình*
- *Trừu tượng*
- *Đóng gói*

Các khái niệm OOPs trong Python

- Đối tượng (object)
- Lớp (class)
- Phương thức (method
- *Kế thừa*
- *Đa hình*
- *Trừu tượng*
- *Đóng gói*



Các khái niệm OOPs trong Python

Đối tượng (object)

Đối tượng là một thực thể có trạng thái và hành vi. Nó có thể là bất kỳ đối tượng trong thế giới thực như chuột, bàn phím, ghế, bàn, bút, v.v.

Tất cả các hàm đều có thuộc tính **__doc__** tích hợp, trả về chuỗi **doc** được xác định trong mã nguồn của hàm.

Lớp (class)

Lớp có thể được định nghĩa là một tập hợp các đối tượng. Nó là một thực thể logic có một số thuộc tính và phương thức cụ thể.

Ví dụ: có một lớp nhân viên thì nó phải chứa một thuộc tính và phương thức, tức là một địa chỉ, tên, tuổi, lương, v.v.

Phương thức

- Phương thức là một hàm được liên kết với một đối tượng.
- Một phương thức không phải là duy nhất cho các thể hiện của lớp.

Các khái niệm OOPs trong Python

- **Kế thừa (Inheritance)**

Xác định rằng đối tượng con có được tất cả các thuộc tính và hành vi của đối tượng cha.

Tạo một lớp sử dụng tất cả các thuộc tính và hành vi của lớp khác.

Lớp mới được biết đến như là một lớp dẫn xuất hoặc lớp con và lớp còn lại gọi là lớp cơ sở hoặc lớp cha.

Kế thừa giúp tái sử dụng lại mã nguồn.

- **Đa hình (Polymorphism)**

Đa hình chứa hai từ "poly" và "morphs". Poly có nghĩa là nhiều và Morphs có nghĩa là hình thức, hình dạng. Bằng đa hình, chúng ta hiểu rằng một nhiệm vụ có thể được thực hiện theo những cách khác nhau.

Ví dụ: một lớp động vật và tất cả các con vật đều biết kêu. Nhưng chúng kêu khác nhau. Ở đây, hành vi "kêu" là đa hình theo nghĩa và phụ thuộc vào động vật. Vì vậy, khái niệm "động vật" trừu tượng không thực sự "nói", nhưng các động vật cụ thể (như chó và mèo) có một triển khai cụ thể của hành động "kêu".

Các khái niệm OOPs trong Python

- **Đóng gói (Encapsulation)**

Đóng gói cũng là một khía cạnh quan trọng của lập trình hướng đối tượng. Nó được sử dụng để hạn chế quyền truy cập vào các phương thức và biến. Trong đóng gói, mã và dữ liệu được gói cùng nhau trong một đơn vị.

- **Trừu tượng (Abstraction)**

Trừu tượng hóa dữ liệu và đóng gói cả hai thường được sử dụng như từ đồng nghĩa. Cả hai đều gần như đồng nghĩa vì sự trừu tượng hóa dữ liệu đạt được thông qua việc đóng gói.

Trừu tượng được sử dụng để ẩn chi tiết nội bộ và chỉ hiển thị các chức năng. Trừu tượng hóa một cái gì đó có nghĩa là đặt tên cho những thứ để cái tên nắm bắt cốt lõi của những gì một chức năng hoặc toàn bộ chương trình làm.

Tạo lớp (Class) trong Python

- Một lớp là một thực thể ảo và có thể được xem như một bản thiết kế của một đối tượng.
- Giả sử một lớp là một nguyên mẫu của một ***tòa nhà***. Một tòa nhà chứa tất cả các chi tiết về *sàn nhà, cửa ra vào, cửa sổ*, v.v. chúng ta có thể tạo ra nhiều tòa nhà như chúng ta muốn, dựa trên những chi tiết này.
- Đối tượng là thể hiện của một Class.
- Quá trình tạo một đối tượng có thể được gọi là khởi tạo.

Tạo lớp (Class) trong Python

- Cú pháp

```
class ClassName:
    # tập lệnh
```

Ví dụ: tạo một lớp Employee có chứa hai trường là Id và name.
Lớp này cũng chứa một hàm show() được sử dụng để hiển thị thông tin của Employee.

```
1class Employee:
2    id = 10
3    name = "Van Anh"
4
5    def display (self):
6        print(self.id, self.name)
```

self *được sử dụng như một biến tham chiếu tham chiếu đến đối tượng lớp hiện tại*

Tạo lớp (Class) trong Python

Ví dụ : tạo ra thể hiện của lớp Employee được định nghĩa trong ví dụ trước.

```
class  Employee:
    id  =  10;
    name =  "Van Anh"
    def  display (self):
        print("ID: %d \nName: %s"  %  (self.id, self.name))
emp =  Employee()
emp.display()
```

```
Kết quả:

ID: 10
Name: Van Anh
```

Constructor trong Python

Constructor trong Python là một loại phương thức (hàm) đặc biệt được sử dụng để khởi tạo các thể hiện của lớp. Constructor có thể có hai loại.

1. Constructor tham số.
2. Constructor không tham số.

Định nghĩa contructor được thực thi khi chúng ta tạo đối tượng của lớp này.

Constructor trong Python

Tạo contructor trong Python

- *Phương thức __ **init** __ mô phỏng contructor của lớp.*
- *Phương thức này được gọi khi lớp được khởi tạo.*
- *Chúng ta có thể chuyển bất kỳ số lượng đối số nào tại thời điểm tạo đối tượng lớp, tùy thuộc vào định nghĩa __ init __.*
- *Nó chủ yếu được sử dụng để khởi tạo các thuộc tính của lớp.*
- *Mỗi lớp phải có một constructor.*

Constructor trong Python

Ví dụ: khởi tạo các thuộc tính của lớp Employ

```
class  Employee:
      def  __init__(self, name, id):
         self.id  =  id
         self.name =  name
      def  display (self):
         print("ID: %d \nName: %s"  %  (self.id, self.name))
emp1 =  Employee("Van Anh", 301)
emp2 =  Employee("Lan Em", 302)
# gọi phương thức display() để hiển thị thông tin employee 1
emp1.display();
# gọi phương thức display() để hiển thị thông tin employee 2
emp2.display();
```

```
Kết quả:

ID: 301
Name: Van Anh
ID: 302
Name: Vân Em
```

Constructor trong Python

Ví dụ: constructor không tham số trong Python

```
class Employee :
    # Constructor không tham số
    def __init__(self):
        print("Đây là Hàm khởi tạo không tham số")
    def show(self, name):
        print("Xin chào", name)
employee = Employee()
employee.show("Van Anh")
```

```
Kết quả:

Đây là Hàm khởi tạo không tham số
Xin chào Van Anh
```

Các hàm lớp dựng sẵn của Python

	Hàm	Mô tả
1	**getattr(obj,name,default)**	Nó được sử dụng để truy cập thuộc tính của đối tượng.
2	**setattr(obj, name,value)**	Nó được sử dụng để đặt một giá trị cụ thể cho thuộc tính cụ thể của một đối tượng.
3	**delattr(obj, name)**	Nó được sử dụng để xóa một thuộc tính cụ thể.
4	**hasattr(obj, name)**	Nó trả về true nếu đối tượng chứa một số thuộc tính cụ thể.

Các hàm lớp dựng sẵn của Python

Ví dụ:

```
class  Student:
    def  __init__(self, name, id, age):
        self.name =  name;
        self.id  =  id;
        self.age =  age
# Tạo đối tượng của lớp Student
s =  Student("Van Anh", 301, 22)
# in thuộc tính name của đối tượng s
print(getattr(s, 'name'))
# Gán giá trị của age cho 23
setattr(s, "age", 23)
# In giá trị của age
print(getattr(s, 'age'))
# true nếu student chứa thuộc tính id
print(hasattr(s, 'id'))
# Xóa thuộc tính age
delattr(s, 'age')
```

Kết quả:

```
Van Anh
23
True
```

Các thuộc tính lớp tích hợp

	Hàm	Mô tả
1	**__dict__**	Nó trả về dictionary chứa namespace của lớp.
2	**__doc__**	Nó chứa một chuỗi về tài liệu lớp.
3	**__name__**	Nó được sử dụng để truy cập tên lớp.
4	**__module__**	Nó được sử dụng để truy cập mô-đun trong đó, lớp này được định nghĩa.
5	**__bases__**	Nó chứa một tuple bao gồm tất cả các lớp cơ sở.

Các thuộc tính lớp tích hợp

Ví dụ:

```
class  Student:
        def  __init__(self, name, id, age):
            self.name =  name;
            self.id  =  id;
            self.age =  age
        def  display_details(self):
            print("Name:%s, ID:%d, age:%d"  %
(self.name, self.id))
  s =  Student("Van anh", 301, 22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

Kết quả:

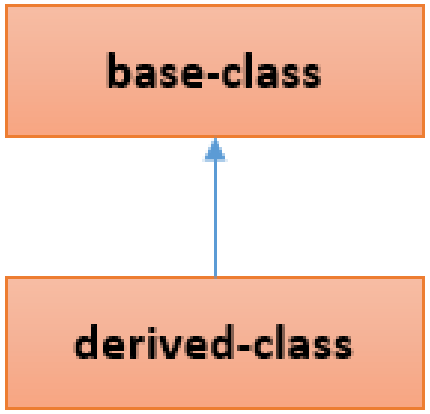
```
None
{'name': 'Van Anh', 'id': 301, 'age': 22}
__main__
```

Kế thừa trong Python

Inheritance

Inheritance

- Kế thừa cung cấp khả năng sử dụng lại mã cho chương trình vì có thể sử dụng một lớp hiện có để tạo một lớp mới thay vì tạo nó từ đầu.
- Lớp con có được các thuộc tính và có thể truy cập tất cả các thành viên dữ liệu và các hàm được định nghĩa trong lớp cha.
- Một lớp con cũng có thể cung cấp việc triển khai cụ thể cho các hàm của lớp cha. Trong bài này, chúng ta sẽ thảo luận chi tiết về kế thừa.



Inheritance

Cú pháp đơn kế thừa

```
class derived-class(base class):
    <class-suite>
```

Ví dụ:

```
class Animal:
    def speak(self):
        print("Animal Speaking")

# lớp con Dog kế thừa lớp Animal
class Dog(Animal):
    def bark(self):
        print("Gou gou!")
d = Dog()
d.bark()
d.speak()
```

Kết quả:

```
Gou gou!
Animal Speaking
```

Inheritance

Kế thừa đa lớp trong Python

- Kế thừa đa lớp khi một lớp dẫn xuất kế thừa một lớp dẫn xuất khác.
- Không có giới hạn về số lượng cấp độ, kế thừa đa cấp trong python.

Cú pháp:

```
class  class1:
    <class-suite>
class  class2(class1):
    <class  suite>
class  class3(class2):
    <class  suite>
...
class  classN(classN-1):
    <class  suite>
```

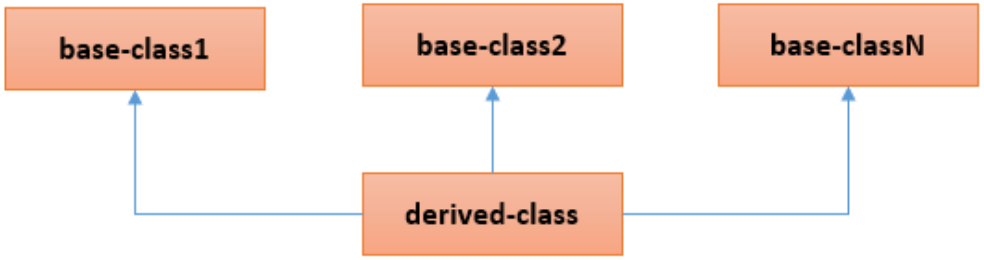
Ví dụ:

```
class  Animal:
    def  speak(self):
        print("Animal Speaking")
# lớp con Dog kế thừa lớp Animal
class  Dog(Animal):
    def  bark(self):
        print("Gou gou!")
# lớp con Dogchild kế thừa lớp Dog
class  DogChild(Dog):
    def  drink(self):
        print("Drink milk...")
d =  DogChild()
d.bark()
d.speak()
d.drink()
d.speak()
```

```
Kết quả:
Gou gou!
Animal Speaking
Drink milk...
Animal Speaking
```

Inheritance

Đa thừa kế trong Python



Cú pháp:

```
class Base1:
    <class-suite>
class Base2:
    <class-suite>
…
class BaseN:
    <class-suite>
class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

Ví dụ:

```
class Calculation1:
    def Summation(self, a, b):
        return a + b;
class Calculation2:
    def Multiplication(self, a, b):
        return a * b;
class Derived(Calculation1, Calculation2):
    def Divide(self, a, b):
        return a / b;
d = Derived()
print(d.Summation(10, 20))
print(d.Multiplication(10, 20))
print(d.Divide(10, 20))
```

```
Kết quả:
30
200
0.5
```

Inheritance

Phương thức isinstance (obj, class)

- Phương thức isinstance() được sử dụng để kiểm tra mối quan hệ giữa các đối tượng và các lớp.
- Nó trả về true nếu tham số đầu tiên, tức là obj là thể hiện của tham số thứ hai, tức là lớp.

Ví dụ:

```
class Calculation1:
    def Summation(self, a, b):
        return a + b;
class Calculation2:
    def Multiplication(self, a, b):
        return a * b;
class Derived(Calculation1, Calculation2):
    def Divide(self, a, b):
        return a / b;
d = Derived()
print("Đối tượng d là thể hiện của lớp
      Derived: ", isinstance(d,Derived))
```

```
Kết quả:
Đối tượng d là thể hiện của lớp Derived:  True
```

Method Overriding

- Phương thức lớp cha được định nghĩa trong lớp con với một số triển khai cụ thể, thì khái niệm này được gọi là ghi đè phương thức trong Python.

Ví dụ:

```
class Animal:
    def speak(self):
        print("Speaking...")
class Dog(Animal):
    def speak(self):
        print("Barking...")
class Cat(Animal):
    def speak(self):
        print("Meo meo...")
d = Dog()
d.speak()
c = Cat()
c.speak()
```

```
Kết quả:
Barking...
Meo meo...
```

Đa hình trong Python

Polymorphism

Polymorphism

- **Đa hình** tức là nhiều hình dạng, có nghĩa là cùng một đối tượng nhưng lại thể hiện khác nhau trong các thời điểm khác nhau.
- Đa hình là một tính năng trong OOP, nó sử dụng một giao diện chung cho nhiều kiểu dữ liệu khác nhau.
- Giả sử, cần tô màu cho một hình dạng, có nhiều lựa chọn hình dạng (hình chữ nhật, hình vuông, hình tròn). Tuy nhiên, có thể sử dụng cùng một phương thức để tô màu bất kỳ hình dạng nào.

→ Khái niệm này được gọi là đa hình.

Polymorphism

- Đa hình trong khi tạo các phương thức của lớp vì Python cho phép các lớp khác nhau có các phương thức có cùng tên.

Ví dụ:

```
class SinhVienIT:
    def in_thong_tin(self):
        print("Sinh viên trường CNTT")

class SinhVienKT:
    def in_thong_tin(self):
        print("Sinh viên trường Kinh tế")

def vi_du(sv):
    sv.in_thong_tin()

sv1 = SinhVienKT()
sv2 = SinhVienIT()
vi_du(sv1)
vi_du(sv2)
```

```
Sinh viên trường Kinh tế
Sinh viên trường CNTT
```

Đóng gói trong Python

Encapsulation

Encapsulation

- Sử dụng OOP trong Python, có thể hạn chế quyền truy cập vào trạng thái bên trong của đối tượng.
- Điều này ngăn chặn dữ liệu bị sửa đổi trực tiếp, được gọi là đóng gói.
- Trong Python, biểu thị thuộc tính private này bằng cách sử dụng dấu gạch dưới làm tiền tố: “_” hoặc “__“.

Encapsulation

Ví dụ:

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Giá bán sản phẩm là:{}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price
c = Computer()
c.sell()
# thay đổi giá:
c.__maxprice = 1000
c.sell()
# sử dụng hàm để thay đổi giá:
c.setMaxPrice(1000)
c.sell()
```

```
Giá bán sản phẩm là: 900
Giá bán sản phẩm là: 900
Giá bán sản phẩm là: 1000
```

Trừu tượng trong Python

Abstraction

Abstraction

- Abstract class là một class mà bên trong nó chứa một hoặc nhiều phương thức trừu tượng.
- Phương thức trừu tượng chỉ được phép khai báo nó và không được phép viết code thực thi nó.
- Khi một class được khai báo ở dạng abstract thì nó sẽ không thể nào khởi tạo được, mà chỉ có thể khởi tạo được thông qua các class con của nó.
- Một class kế thừa lại abstract class thì phải khai báo lại toàn bộ các phương thức trừu tượng bên trong abstract class mà nó kế thừa.

Abstract

- Abstract là một class, và class này chứa một hoặc nhiều phương thức trừu tượng.
- Abstract class không thể khởi tạo trực tiếp được.

Ví dụ:

```
from abc import ABC, abstractmethod
class PersonAbstact(ABC):
    name = None
    age = 0
    def getName(self):
        print(self.name)
    def getAge(self):
        print(self.age)
    @abstractmethod
    def getFull(self):
        pass
class Person(PersonAbstact):
    name = 'Vuong Xuan Chi'
    age = 35
    def getFull(self):
        self.getName()
        self.getAge()

Person().getFull();
```

```
# Kết Quả:
# Vuong Xuan Chi
# 35
```

Abstract

- Thực hiện ẩn dữ liệu bằng cách thêm dấu gạch dưới kép (__) làm tiền tố cho thuộc tính cần ẩn.
- Thuộc tính sẽ không hiển thị bên ngoài lớp thông qua đối tượng.

Ví dụ:

```
class Employee:
    __count = 0;
    def __init__(self):
        Employee.__count = Employee.__count + 1
    def display(self):
        print("Số lượng nhân viên: ",
Employee.__count)
emp1 = Employee()
emp2 = Employee()
try:
    print(emp1.__count)
finally:
    emp1.display()
```

```
Kết quả:
Số lượng nhân viên:  2
AttributeError: 'Employee' object has no attribute '__count'
```

THANK YOU

THS. VƯƠNG XUÂN CHÍ
VXCHI@NTT.EDU.VN
0903 270 567