LẬP TRÌNH PYTHON TỪ CƠ BẢN ĐẾN NÂNG CAO

Thực hiện: TS.Trần Bình Long

Class and Object

1. Lập trình hướng đối tượng
2. Class và Object
3. Kế thừa (Inheritance)
4. Đa kế thừa (Multiple Inheritance)
5. Đóng gói
6. Đa hình
7. Toán tử đa dụng

Lập trình hướng đối tượng

Lập trình hướng đối tượng (Object-oriented programming, viết tắt: OOP) là kỹ thuật hỗ trợ, cho phép làm việc trực tiếp với các đối tượng được định nghĩa.

Kĩ thuật này giúp tăng năng suất, đơn giản hoá độ phức tạp khi bảo trì và mở rộng phần mềm. Có nhiều ngôn ngữ lập trình hướng đối tượng như C++, Java, PHP,... , Python.

OOP tập trung vào việc tạo code sử dụng lại, còn được gọi là DRY (Don't Repeat Yourself).

Các nguyên tắc OOP

OOP phải theo nguyên tắc: *tính đóng gói, tính kế thừa và tính đa hình*.

Tính đóng gói (Encapsulation): các trạng thái bên trong của đối tượng được bảo vệ và tránh truy cập từ bên ngoài.

Tính kế thừa (Inheritance): cho phép một lớp (class) kế thừa các thuộc tính, phương thức từ lớp khác.

Tính đa hình (Polymorphism): hai hay nhiều lớp có những phương thức giống nhau nhưng thực thi theo cách thức khác nhau.

Lớp (Class) và Đối tượng (Object)

Class và Object là hai khái niệm cơ bản trong lập trình hướng đối tượng.

Lớp (Class) là kiểu dữ liệu do người dùng định nghĩa, chứa các thuộc tính đặc trưng cho đối tượng.

Thuộc tính là các giá trị của lớp. Khi đối tượng được tạo ra, thì thuộc tính của lớp sẽ trở thành các đặc điểm của đối tượng.

Đối tượng (Object) là thực thể có hành vi.
Ví dụ đối tượng là hình học có thuộc tính tên hình, số cạnh, không gian, hành vi cộng, trừ, nhân, chia...

Đối tượng còn được gọi là một thể hiện (instance) của lớp và tạo đối tượng được gọi là instantiation

Khai báo Class

Khai báo lớp sử dụng từ khóa class.
Dòng kí tự đầu tiên gọi là *docstring* - mô tả ngắn gọn lớp. *Docstring* không bắt buộc nhưng nên có.

```
>>> class MyNewClass:
        """Đây là docstring, lớp mới được khai báo."""
        pass
```

Đây là cách khai báo class đơn giản.
Class tạo ra một local namespace mới để khai báo các thuộc tính.

Thuộc tính có thể là hàm hoặc dữ liệu.

Khai báo Class

Thuộc tính đặc biệt bắt đầu với dấu gạch dưới kép (__). Ví dụ: ***__doc__*** trả về chuỗi docstring mô tả lớp.

Ngay khi khai báo lớp, môt đối tượng trong lớp được tạo ra cùng tên. Đối tượng lớp này cho phép truy cập các thuộc tính khác nhau cũng như khởi tạo các đối tượng mới của lớp.

```
>>> class MyClass:
        “”” Class thứ 2 được khởi tạo”””
        a = 10
        def func(self):
            print('Xin chào')
```

Khai báo Class

```
>>> print(MyClass.a)
    # Output: 10
    print(MyClass.func)
    # Output: <function MyClass.func at
0x0000000003079BF8>
    print(MyClass.__doc__)
    # Output: ‘Class thứ 2 được khởi tạo'
```

Tạo đối tượng

Đối tượng trong class được sử dụng để truy cập các thuộc tính khác nhau và tạo các instance mới của lớp đó. Để tạo một đối tượng tương tự như cách gọi hàm.

```
obj = MyClass()
```

Lệnh này tạo ra một đối tượng mới có tên là *obj.*
Tạo lớp bao gồm các thuộc tính, phương thức:

```
>>> class MyClass:
        """ Class thứ 3 được khởi tạo"""
        a = 10
        def func(self):
            print('Xin chào')
>>> obj = MyClass()
```

Tạo đối tượng

```
>>> print(MyClass.func)
    # Output: <function MyClass.func at
0x000000000335B0D0>
    print(obj.func)
    # Output: <bound method MyClass.func of
<__main__.MyClass object at 0x000000000332DEF0>
    # Gọi hàm func()
    obj.func()
    # Output: Xin chào
```

Tạo đối tượng

Khi định nghĩa hàm trong class, tham số (parameter) là *self,* nhưng khi gọi hàm *obj.func()* không cần tham số. Bởi vì, khi đối tượng (object) gọi phương thức, đối tượng sẽ tự pass qua tham số đầu tiên. Nghĩa là *obj.func()* tương đương với *MyClass.func(obj)*

Constructor

Hàm trong Class bắt đầu với dấu gạch dưới kép (__) là các hàm đặc biệt, có ý nghĩa đặc biệt.
Hàm *__init__():* được gọi khi khởi tạo một đối tượng, một biến mới trong class và được gọi là constructor.

```
>>>class SoPhuc:
        def __init__(self,r = 0,i = 0):
            self.phanthuc = r
            self.phanao = i
        def getData(self):
            print("{}+{}j".format(self.phanthuc,self.phanao))
```

Constructor

```
>>> # Tạo đối tượng số phức mới
    c1 = SoPhuc(2,3)
    # Gọi hàm getData()
    c1.getData()
    # Output: 2+3j
    # Tạo đối tượng số phức mới
    c2 = SoPhuc(5)
    # tạo thêm một thuộc tính mới (new)
    c2.new = 10
    print((c2.phanthuc, c2.phanao, c2.new))
    # Output: (5, 0, 10)
```

Constructor

Ví dụ trên, khai báo lớp mới để biểu diễn số phức.

Hàm *__init__()* khởi tạo các biến (mặc định là 0)

Hàm *getData()* để hiển thị.

Các thuộc tính thêm vào của đối tượng được tạo ra nhanh chóng, ví dụ tạo thuộc tính mới *'new'* cho đối tượng *c2* và có thể gọi ngay lập tức. Tuy nhiên thuộc tính mới này sẽ không áp dụng với các đối tượng đã khai báo trước như *c1.*

Xóa thuộc tính và đối tượng

Thuộc tính của đối tượng có thể bị xóa bằng lệnh *del*

```
>>> c1 = SoPhuc(2,3)
>>> del c1.phanao
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'SoPhuc' object has no attribute 'phanao'
>>> del SoPhuc.getData
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'SoPhuc' object has no attribute 'getData'
```

Xóa thuộc tính và đối tượng

Xóa chính đối tượng đó bằng câu lệnh *del.*

```
>>> c1 = SoPhuc(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Sau khi xóa, object vẫn tồn tại trên bộ nhớ, nhưng sau đó phương thức destruction của Python (hay còn gọi là garbage collection) sẽ loại bỏ hoàn toàn các dữ liệu này trên bộ nhớ

Ví dụ về Class và Object

```
>>> class Hinhhoc:
        # thuộc tính lớp
        loaiHH = “Hình đa giác"
        # thuộc tính đối tượng
        def __init__(self, tenHinh, socanh, khonggian):
            self.tenHinh = tenHinh
            self.socanh = socanh
            self.khonggian = khonggian
>>> # instantiate the Hinhhoc class
    tamgiac = Hinhhoc("Tamgiac", “Ba cạnh", “Ơ clit")
    chunhat = Hinhhoc(“Chunhat", “Bốn cạnh", “Ơ clit")
```

Ví dụ về Class và Object

```
>>> lucgiac = Hinhhoc(“Lucgiac", “Sáu cạnh", “Đe cat")
    # access the class attributes
    print(“Lucgiac là {}.".format(lucgiac.__class__.loaiHH))
    print("Tamgiac là {}.".format(tamgiac.__class__.loaiHH))
    print(“Chunhat là {}.".format(chunhat.__class__.loaiHH))
```

Ví dụ về Class và Object

```
>>> # access the instance attributes
      print("Hình {} có số cạnh là {}. {} là không gian mô tả.".format(tamgiac.tenHinh, tamgiac.socanh, tamgiac.khonggian))
      print("Hình {} có số cạnh là {}. {} là không gian mô tả.".format(chunhat.tenHinh,chunhat.socanh,chunhat.khonggian))
      print("Hình {} có số cạnh là {}. {} là không gian mô tả.".format(lucgiac.tenHinh, lucgiac.socanh, lucgiac.khonggian))
```

Ví dụ về Class và Object

Kết quả:

```
>>> Lục giác là Hình đa giác.
    Tam giác là Hình đa giác.
    Chử nhật cũng là Hình đa giác.
    Hình Tam giác có số cạnh là Ba cạnh. Ơ clit là không gian mô tả.
    Hình Chử nhật có số cạnh là Bốn cạnh. Ơ clit là không gian mô tả.
    Hình Lục giác có số cạnh là Sáu cạnh. Đề cat là không gian mô tả.
```

Ví dụ về Class và Object

Lớp *Hinhhoc* được tạo, sau đó xác định các thuộc tính, đặc điểm của đối tượng.

Truy cập thuộc tính class bằng cú pháp *__class__.loaiHH.*

Thuộc tính lớp được chia sẻ cho các đối tượng của lớp.

Truy cập các thuộc tính instance bằng cú pháp *tamgiac.tenHinh, tamgiac.socanh* và *tamgiac.khonggian.*

Mỗi đối tượng của lớp có thuộc tính instance khác nhau.

Phương thức

Phương thức (Method) là các hàm được định nghĩa trong lớp, dùng để xác định hành vi của đối tượng.

```
>>> class Hinhhoc:
        # thuộc tính đối tượng
        def __init__(self, tenHinh, socanh, khonggian):
            self.tenHinh = tenHinh
            self.socanh = socanh
            self.khonggian = khonggian
        # phương thức
        def DienTich(self, mucdich):
            return " Tính diện tích {} để
{}".format(self.tenHinh, mucdich)
```

Phương thức

```
>>>     def ChuVi(self):
            return “Tính chu vi hình {}  ".format(self.tenHinh)
        def Canh(self):
            return “Tính độ dài cạnh {}".format(self.tenHinh)
    # call our instance methods
    print(tamgiac.DienTich('xây nhà'))
    print(chunhat.ChuVi())
    print(lucgiac.Canh())
Kết quả:
>>> Tính diện tích Tam giác để xây nhà
    Tính chu vi hình Chử nhật
    Tính độ dài cạnh Lục giác
```

Phương thức

Ba phương thức *DienTich(), ChuVi()* và *Canh()* được gọi là phương thức instance vì gọi trên đối tượng instance *(tamgiac, chunhat, lucgiac)*

Kế thừa (Inheritance)

Kế thừa (Inheritance) cho phép một lớp (class) kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa.
Lớp đã có gọi là lớp cha (base class hoặc parent class), lớp mới gọi là lớp con (child class hoặc derived class). Lớp con kế thừa tất cả thành phần của lớp cha, có thể mở rộng các thành phần kế thừa và bổ sung thêm các thành phần mới.

Cú pháp của kế thừa

```
>>> class BaseClass:
        Body of base class
    class DerivedClass(BaseClass):
        Body of derived class
```

Kế thừa

Đa giác là một hình khép kín có 3 cạnh trở lên. Tạo lớp *DaGiac* như sau

```
>>> class DaGiac:
        def __init__(self, socanh):
            self.n = socanh
            #self.canh = [0 for i in range(socanh)]
        def nhapcanh(self):
            self.canh = [float(input("nhập giá trị
cạnh"+str(i+1)+" : ")) for i in range(self.n)]
        def hienthicanh(self):
            for i in range(self.n):
                print("Giá trị cạnh",i+1,"là",self.canh[i])
```

Kế thừa

Class *DaGiac* có thuộc tính n để định nghĩa số cạnh và *canh* để lưu giá trị mỗi cạnh.

Hàm *nhapcanh()* độ lớn các cạnh

Hàm hienthicanh() hiện thị danh sách các cạnh của đa giác.

Hình tam giác là đa giác có ba cạnh, tạo lớp *TamGiac* kế thừa từ *DaGiac.* Class mới này sẽ thừa kế tất cả các thuộc tính sẵn có trong lớp cha. Lớp*TamGiac* được khai báo như sau:

```
>>> class TamGiac(DaGiac):
        def __init__(self):
            DaGiac.__init__(self,3)
```

Kế thừa

```
>>>     def dientich(self):
            a, b, c = self.canh
            # Tính nửa chu vi
            p = (a + b + c) / 2
            s= (p*(p-a)*(p-b)*(p-c)) ** 0.5
            print('Diện tích của hình tam giác là %0.2f'
%s)
```

Lớp *TamGiac* định nghĩa thêm hàm *dientich().*

```
>>> t = TamGiac()
>>> t.nhapcanh()
    nhập giá trị cạnh 1 : 3
    nhập giá trị cạnh 2 : 5
    nhập giá trị cạnh 3 : 4
```

Kế thừa

```
>>> t.hienthicanh()
      Giá trị cạnh 1 là 3.0
      Giá trị cạnh 2 là 5.0
      Giá trị cạnh 3 là 4.0
>>> t.dientich()
      Diện tích của hình tam giác là 6.00
```

Các hàm *nhapcanh(), hienthicanh()* đều không có trong class TamGiac, nhưng vẫn được class sử dụng.

Kế thừa (Inheritance)

ví dụ:

```
>>> # Lớp cha
      class Hinhhoc:
         # Constructor
         def __init__(self, loaihinh, tenhinh, khonggian):
            # Lớp Hinhhoc có 3 thuộc tính: loaihinh,
tenhinh, khonhgian
            self.loaihinh = loaihinh
            self.tenhinh = tenhinh
            self.khonggian = khonggian
```

Kế thừa

```
>>>        # phương thức
           def ChuVi(self):
               print (“Tính chu vi hình{}".format(self.tenhinh))
           def DienTich(self, mucdich):
               print (“Tính diện tích hình {} để
{}".format(self.tenhinh, mucdich))
>>> # Lớp Tamgiac mở rộng từ lớp Hinhhoc.
    class Tamgiac(Hinhhoc):
        def __init__(self, loaihinh, tenhinh, khonggian,
socanh)
            # Gọi tới constructor của lớp cha (Hinhhoc)
            # để gán giá trị vào thuộc tính của lớp cha.
            super().__init__(loaihinh,tenhinh,khonggian)
```

Kế thừa

```
>>>         self.socanh=socanh
          # kế thừa phương thức cũ
           def ChuVi(self):
              print ('Tính chu vi hình {}'.format(self.tenhinh))
          # ghi đè (override) phương thức cùng tên của
lớp cha.
           def DienTich(self,mucdich):
               print('Tính diện tích hình {} để
{}'.format(self.tenhinh, mucdich))
               print('Hình {} có
{}'.format(self.tenhinh,self.socanh))
          # bổ sung them thành phần mới
```

Kế thừa

```
>>>     def Canh(self):
            print('Tính cạnh của {}'.format(self.tenhinh))
>>> tamgiacthuong=Tamgiac('Tam giác', 'Tam giác thường', 'Ơ clit', 'ba cạnh không bằng nhau')
    tamgiaccan= Tamgiac('Tam giác', 'Tam giác cân', 'Ơ clit', 'hai cạnh bằng nhau')
    tamgiacdeu= Tamgiac('Tam giác', 'Tam giác đều', 'Ơ clit', 'Ba cạnh bằng nhau')
    tamgiacthuong.DienTich('xây nhà')
    tamgiaccan.ChuVi()
    tamgiacdeu.Canh()
```

Kế thừa

Kết quả:

```
>>> Tính diện tích hình Tam giác thường để xây nhà
    Hình Tam giác thường có ba cạnh không bằng nhau
    Tính chu vi hình Tam giác cân
    Tính cạnh của Tam giác đều
```

Khai báo constructor để gán giá trị vào thuộc tính của lớp Hinhhoc.

Hàm *super()* đứng trước phương thức *__init__* để gọi tới nội dung *__init__* của Hinhhoc.

Lớp Tamgiac kế thừa hàm *ChuVi()* và *DienTich()* của lớp Hinhhoc đồng thời sửa đổi một hành vi thể hiện ở phương thức *DienTich()*. Sau đó lớp con bổ sung thêm thành phần mới là *Canh()* để mở rộng kế thừa.

Overriding (Ghi đè)

Python cho phép ghi đè lên các phương thức của lớp cha.

Class *TamGiac* sử dụng lại hàm *__init__()* từ class *DaGiac,* nếu muốn override (ghi đè) lại định nghĩa của hàm *__init__()* trong class cha, dùng hàm super().
super () .__ *init* __ *(3)* tương đương với *DaGiac.__ init __ (self, 3)*
Python có hai hàm *isinstance()* và *issubclass()* dùng để kiểm tra mối quan hệ của hai lớp và instance.

Hàm *issubclass(classA, classB)* trả về True nếu class A là lớp con của class B.

Overriding (Ghi đè)

```
>>> issubclass(DaGiac,TamGiac)
    False
>>> issubclass(TamGiac,DaGiac)
    True
```

Hàm *isinstance(obj, Class)* trả về True nếu obj là instance của lớp Class hoặc là một instance của lớp con của Class

```
>>> isinstance(t,TamGiac)
    True
>>> isinstance(t,DaGiac)
    True
>>> isinstance(t,int)
    False
```

Đa kế thừa (Multiple Inheritance)

Một lớp con được kế thừa từ nhiều lớp cha được gọi là đa kế thừa

Cú pháp:

```
>>> class LopCha1:
        pass
    class LopCha2:
        pass
    class LopCon(LopCha1, LopCha2):
        pass
```

Các lớp cha có thể có thuộc tính hoặc phương thức giống nhau. Lớp con sẽ ưu tiên thừa kế thuộc tính, phương thức của lớp đứng đầu tiên trong danh sách thừa kế.

Kế thừa đa cấp (Multilevel Inheritance)

Ngoài việc kế thừa từ các lớp cha, lớp con mới còn kế thừa các lớp con trước đó. Đây gọi là kế thừa đa cấp (Multilevel Inheritance).

Trường hợp này, các đặc tính của lớp cha và lớp con trước đó sẽ được lớp con mới kế thừa.

Cú pháp:

```
>>> class LopCha:
        pass
    class LopCon1(LopCha):
        pass
    class LopCon2(LopCon1):
        pass
```

LopCon1 kế thừa *LopCha*, *LopCon2* kế thừa *LopCon1*

Thứ tự truy xuất phương thức (Method Resolution Order)

Trong đa thừa kế, khi truy xuất thuộc tính, đầu tiên sẽ tìm kiếm trong lớp hiện tại. Nếu không tìm thấy, tìm kiếm tiếp tục lớp cha đầu tiên và từ trái qua phải.
Thứ tự truy xuất: *[LopCon, LopCha1, LopCha2, object].*
Thứ tự này được gọi là tuyến tính hóa của *LopCon* và quy tắc sử dụng để tìm theo thứ tự này được gọi là Thứ tự truy xuất phương thức (MRO).

MRO dùng để hiển thị danh sách các class cha của một class nào đó.

Thứ tự truy xuất phương thức (MRO)

MRO được sử dụng theo hai cách:
__mro__: trả về một tuple
mro(): trả về một danh sách.

```
>>> LopCon.__mro__
      (<class '__main__.LopCon'>,
<class'__main__.LopCha1'>,<class'__main__.LopCha
2'>, <class 'object'>)
>>> LopCon.mro()
      [<class '__main__.LopCon>,
      <class '__main__.LopCha1'>,
      <class '__main__.LopCha2'>,
      <class 'object'>]
```

Thứ tự truy xuất phương thức (MRO)

ví dụ thừa kế phức tạp và hiển thị MRO.

```
>>> class X: pass
      class Y: pass
      class Z: pass
      class A(X,Y): pass
      class B(Y,Z): pass
      class M(B,A,Z): pass
# Output:
>>> # [<class '__main__.M'>, <class '__main__.B'>,
      # <class '__main__.A'>, <class '__main__.X'>,
      # <class '__main__.Y'>, <class '__main__.Z'>,
      # <class 'object'>]
```

Đóng gói (Encapsulation)

Đóng gói: Hạn chế truy cập vào bên trong đối tượng, ngăn chặn dữ liệu bị sửa đổi.

Trong Python, sử dụng thuộc tính private bằng dấu gạch dưới: "_" hoặc "__".

```
>>> class Computer:
        def __init__(self):
        # Thuộc tính private ngăn chặn sửa đổi trực tiếp
            self.__maxprice = 900
        def sell(self):
            print("Giá bán sản phẩm: {}".format
(self.__maxprice))
        def setMaxPrice(self, price):
            self.__maxprice = price
```

Đóng gói

```
>>> c = Computer()
        c.sell()
        # thay đổi giá.
        c.__maxprice = 1000
        c.sell()
        # sử dụng hàm setter setMaxPrice để thay đổi giá.
        c.setMaxPrice(1000)
        c.sell()
```

Kết quả:

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

Đóng gói

Tạo class Computer, sử dụng *__init__()* để lưu trữ giá bán tối đa của máy tính.

Sau khi sử dụng, nếu cần sửa đổi giá, nhưng không thể sửa đổi theo cách bình thường vì *__maxprice* là thuộc tính private. Vậy để thay đổi giá trị, sử dụng hàm **setter setMaxPrice()**.

Đa hình (Polymorphism)

Tính đa hình là khái niệm mà hai hoặc nhiều lớp có những phương thức giống nhau nhưng thực hiện theo cách thức khác nhau.

```
>>> class Tamgiac:
        def DienTich(self):
            print("Tính diện tích tam giác")
        def Canh(self):
            print("Tam giac có ba cạnh")
>>> class LucGiac:
        def DienTich(self):
          print("Lục giác đều bằng tổng 6 tam giác đều")
        def Canh(self):
          print("Lục giác đều có sáu cạnh bằng nhau")
```

Đa hình

```
>>> # common interface
    def kiemtra_DienTich(Hinhhoc):
        Hinhhoc.DienTich()
    # instantiate objects
    tamgiac = Tamgiac()
    lucgiac = Lucgiac()
    # passing the object
    kiemtra_DienTich(tamgiac)
    kiemtra_DienTich(lucgiac)
```

Đa hình

Hai lớp *Tamgiac* và *LucGiac,* đều có phương thức *DienTich().* Truy nhiên thực hiện khác nhau.

Tính đa hình được sử dụng để tạo hàm chung cho hai lớp, đó là *kiemtra_DienTich().* Tiếp theo, truyền đối tượng *tamgiac* và *lucgiac* vào hàm vừa tạo,

kết quả:

>>>Tính diện tích tam giác

Lục giác đều bằng tổng 6 tam giác đều

Toán tử đa dụng

Một toán tử có thể được sử dụng để thực hiện nhiều việc khác nhau. Ví dụ: toán tử ' + ', dùng cộng hai số với nhau, kết hợp hai danh sách, hoặc nối hai chuỗi khác nhau …

Tính năng này gọi là toán tử đa dụng, cho phép cùng một toán tử được sử dụng khác nhau tùy từng ngữ cảnh.

Khi sử dụng toán tử đa dụng với đối tượng của một lớp do người dùng khai báo. ví dụ mô phỏng một điểm trong hệ tọa độ hai chiều sau:

Toán tử đa dụng

```
>>> class Point:
        def __init__(self, x = 0, y = 0):
            self.x = x
            self.y = y
```

Thực hiện chương trình và nhập các điểm:

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Point'
and 'Point'
```

Toán tử đa dụng

Chương trình báo lỗi *TypeError:* không thể nhận hai đối tượng *Point* cùng lúc. Để xử lý vấn đề này, thực hiện các bước sau:

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

Nên khai báo phương thức ***__str__()*** trong class để kiểm soát cách hiển thị kết quả được in ra.

```
>>> class Point:
        def __init__(self, x = 0, y = 0):
            self.x = x
            self.y = y
        def __str__(self):
            return "({0},{1})".format(self.x,self.y)
```

Toán tử đa dụng

Dùng hàm *print()*

```
>>> p1 = Point(2,3)
>>> print(p1)
      (2,3)
```

Sử dụng ***__str__()*** làm kết quả hiển thị chuẩn hơn. Ngoài ra có thể in ra kết quả bằng hàm tích hợp sẵn trong Python là str() hoặc format().

```
>>> str(p1)
      '(2,3)'
>>> format(p1)
      '(2,3)'
```

Khi sử dụng str() và format(), Python thực hiện lệnh gọi p1.__str__() nên kết quả được trả về tương tự.

Toán tử đa dụng ' + '

Với toán tử đa dụng ' + ', dùng hàm ***__add__()*** trong class.

```
>>> class Point:
        def __init__(self, x = 0, y = 0):
            self.x = x
            self.y = y
        def __str__(self):
            return "({0},{1})".format(self.x,self.y)
        def __add__(self,other):
            x = self.x + other.x
            y = self.y + other.y
            return Point(x,y)
```

Toán tử đa dụng' + '

Thực hiện chương trình và nhập vào các điểm:

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
      (1,5)
```

Khi thực hiện *p1 + p2*, Python sẽ gọi *p1.__add__(p2).*

Hàm đặc biệt

Hàm trong Class được bắt đầu với hai dấu gạch dưới liền nhau *(__)* là các hàm đặc biệt, có ý nghĩa đặc biệt.

Có nhiều hàm đặc biệt một trong đó là hàm *__init__()*

Hàm này được gọi khi khởi tạo một đối tượng, một biến mới trong class.

Các hàm đặc biệt này giúp những hàm của người dùng tương thích với các hàm có sẵn trong Python.

Một số hàm đặc biệt dùng cho toán tử đa dụng trong bảng dưới đây:

Toán tử đa dụng

TOÁN TỬ	BIỂU DIỄN	HÀM
Phép cộng	p1 + p2	p1.__add__(p2)
Phép trừ	p1 - p2	p1.__sub__(p2)
Phép nhân	p1 * p2	p1.__mul__(p2)
Lũy thừa	p1 ** p2	p1.__pow__(p2)
Phép chia	p1 / p2	p1.__truediv__(p2)
Phép chia lấy phần nguyên (Floor Division)	p1 // p2	p1.__floordiv__(p2)
Số dư (modulo)	p1 % p2	p1.__mod__(p2)

Toán tử đa dụng

Thao tác trên bit: phép dịch trái	p1 << p2	p1.__lshift__(p2)
Thao tác trên bit: phép dịch phải	p1 >> p2	p1.__rshift__(p2)
Thao tác trên bit: phép AND	p1 & p2	p1.__and__(p2)
Thao tác trên bit: phép OR	p1 | p2	p1.__or__(p2)
Thao tác trên bit: phép XOR	p1 ^ p2	p1.__xor__(p2)
Thao tác trên bit: phép NOT	~p1	p1.__invert__()

Toán tử đa dụng

Python không chỉ có phép toán tử đa dụng toán học, mà còn có phép toán tử đa dụng so sánh.
Các toán tử so sánh được hỗ trợ bởi Python: <, >, <=, >=, ==,...
Sử dụng toán tử đa dụng này khi muốn so sánh các đối tượng trong lớp với nhau.
Ví dụ muốn so sánh các điểm trong class *Point,* hãy so sánh độ lớn của các điểm này bắt đầu từ gốc tọa độ, thực hiện như sau:

Toán tử đa dụng

```
>>> class Point:
        def __init__(self, x = 0, y = 0):
            self.x = x
            self.y = y
        def __str__(self):
            return "({0},{1})".format(self.x,self.y)
        def __lt__(self,other):
            self_mag = (self.x ** 2) + (self.y ** 2)
            other_mag = (other.x ** 2) + (other.y ** 2)
            return self_mag < other_mag
```

Toán tử đa dụng

Thực hiện chương trình, nhập các điểm và toán tử dùng để só sánh:

```
>>> Point(1,1) < Point(-2,-3)
    True
>>> Point(1,1) < Point(0.5,-0.2)
    False
>>> Point(1,1) < Point(1,1)
    False
```

Tương tự, có thể có nhiều toán tử đa dụng so sánh khác. Một số hàm đặc biệt dùng cho toán tử đa dụng so sánh trong bảng dưới đây:

Toán tử đa dụng so sánh

TOÁN TỬ	BIỂU DIỄN	HOẠT ĐỘNG
Nhỏ hơn	p1 < p2	p1.__lt__(p2)
Nhỏ hơn hoặc bằng	p1 <= p2	p1.__le__(p2)
Bằng	p1 == p2	p1.__eq__(p2)
Khác	p1 != p2	p1.__ne__(p2)
Lớn hơn	p1 > p2	p1.__gt__(p2)
Lớn hơn hoặc bằng	p1 >= p2	p1.__ge__(p2)

OOP

Ưu điểm của OOP. Qua các ví dụ, có thể rút ra một số nhận xét như này:

- Lập trình trở nên dễ dàng và hiệu quả hơn.
- Class có thể chia sẻ được nên code dễ dàng được sử dụng lại.
- Năng suất của chương trình tăng lên
- Dữ liệu an toàn và bảo mật với trừu tượng hóa dữ liệu

Bài tập

1. Định nghĩa một class có 2 method:
getString: để nhận một chuỗi do người dùng nhập vào.
printString: in chuỗi vừa nhập sang chữ hoa.

2. Định nghĩa một lớp gồm có tham số lớp và có cùng tham số instance
(**tham số instance, thêm vào __init__**)

Bài tập

3. Định nghĩa class tên Vietnam và class con của nó là Hanoi

(dùng Subclass(ParentClass) để định nghĩa class con)

4. Định nghĩa class có tên là Circle được xây dựng từ bán kính. Circle có một method tính diện tích

5. Định nghĩa class tên Hinhchunhat được xây dựng bằng chiều dài và chiều rộng. Class Hinhchunhat có method để tính diện tích

Bài tập

6. Định nghĩa class Shape và class con là Square. Square có hàm init để lấy đối số là chiều dài. Cả 2 class đều có hàm area để in diện tích của hình, diện tích mặc định của Shape là 0

7. Định nghĩa class Nguoi và 2 class con của nó: Nam, Nu. Tất cả các class có method "getGender" có thể in "Nam" cho class Nam và "Nữ" cho class Nu.