



Edited with the trial version of
Foxit Advanced PDF Editor
To remove this notice, visit:
www.foxitsoftware.com/shopping

Ngôn Ngữ Lập Trình Python

Object-Oriented Programming (OOP)

Nội Dung

- Introduction
- Namespaces and Dataclasses
- Classes in Python
- Inheritance in Python
- Multiple Inheritance
- Operator Overloading
- Example

OOP

- Trong Python, khái niệm về OOP tuân theo một số nguyên lý cơ bản là *tính đóng gói, tính kế thừa và tính đa hình*.
 - *Tính kế thừa (Inheritance)*: cho phép một lớp (class) có thể kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa.
 - *Tính đóng gói (Encapsulation)*: là quy tắc yêu cầu trạng thái bên trong của một đối tượng được bảo vệ và tránh truy cập được từ code bên ngoài.
 - *Tính đa hình (Polymorphism)*: là khái niệm mà hai hoặc nhiều lớp có những phương thức giống nhau nhưng có thể thực thi theo những cách thức khác nhau.

Lớp (Class) và Đối tượng (Object)

- **Đối tượng (Object)** là những thực thể tồn tại có hành vi.
- **Lớp (Class)** là một kiểu dữ liệu đặc biệt do người dùng định nghĩa, tập hợp nhiều thuộc tính đặc trưng cho mọi đối tượng được tạo ra từ lớp đó.
- Một đối tượng là **một thực thể (instance)** của một lớp
- **Phương thức (Method)** là các hàm được định nghĩa bên trong phần thân của một lớp. Chúng được sử dụng để xác định các hành vi của một đối tượng.

Kế thừa (Inheritance)

- **Tính kế thừa** cho phép một lớp (class) có thể kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa.
- Lớp đã gọi là lớp cha, lớp mới phát sinh gọi là lớp con. 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.

Đóng gói (Encapsulation)

- Sử dụng OOP trong Python, chúng ta 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, chúng ta biểu thị thuộc tính private này bằng cách sử dụng dấu gạch dưới làm tiền tố: “_” hoặc “__”.

Đa hình (Polymorphism)

- Tính đa hình là khái niệm mà hai hoặc nhiều lớp có những phương thức giống nhau nhưng có thể thực thi theo những cách thức khác nhau.
- Giả sử, chúng ta cần tô màu một hình khối, có rất nhiều lựa chọn cho hình của bạn như hình chữ nhật, hình vuông, hình tròn. Tuy nhiên, bạn có thể sử dụng cùng một phương pháp để tô màu bất kỳ hình dạng nào.

Methods vs Functions

- Các phương thức (method) thường được sử dụng là s.f() thay vì f(s).

```
s = 'hello world'  
print(len(s)) # len là hàm  
print(s.upper()) # upper is a string method, called using the . notation  
                  # gọi phương thức upper cho chuỗi s  
print(s.replace('hello', 'hi')) # vài method có thêm đối số
```

Methods vs Functions

- Ví dụ về lỗi do cách sử dụng method và function không đúng

```
n = 123 print(len(n)) # TypeError: object of type 'int' has no len()
```

```
n = 123 print(n.upper()) # AttributeError: 'int' object has no attribute 'upper'
```

Classes and Instances

- Classes cũng thường được gọi là "Types" trong Python. Ví dụ các classes là int, float, str, bool.
- Instances là các giá trị cụ thể của một class hoặc một kiểu nhất định. Ví dụ 'hello' là một thực thể string (hay còn gọi là một string)

```
x = 5  
print(type(x))          # <class 'int'>  
print(type('hello'))    # <class 'str'>
```

Namespaces

- Chúng ta có thể dùng namespaces để tạo các đối tượng có thể thay đổi (mutable objects). Tuy nhiên, sẽ không thuận lợi để lưu trữ một tập hợp các thuộc tính (fields or attributes) trong một đối tượng đơn lẻ.
- Ví dụ: (xem slide sau)

Namespaces

Don't forget this import:

```
from types import SimpleNamespace
```

Now we can create new object representing dogs:

```
dog1 = SimpleNamespace(name='Dino', age=10, breed='shepherd')
```

```
print(dog1) # prints: namespace(age=10, breed='shepherd', name='Dino')
```

```
print(dog1.name) # prints: Dino
```

```
dog1.name = 'Fred'
```

```
print(dog1) # prints: namespace(age=10, breed='shepherd', name='Fred')
```

```
print(dog1.name) # prints: Fred
```

```
dog2 = SimpleNamespace(name='Spot', age=12, breed='poodle')
```

```
dog3 = SimpleNamespace(name='Fred', age=10, breed='shepherd')
```

```
print(dog1 == dog2) # prints: False
```

```
print(dog1 == dog3) # prints: True
```

```
print(type(dog1)) # prints <class 'types.SimpleNamespace'>
```

Dataclasses

- A **Dataclass** tương tự như SimpleNamespace, có sự cải tiến là nó yêu cầu các thuộc tính (fields or attributes).
- Ví dụ: (xem slide sau)

```
from dataclasses import make_dataclass
# Now we can create a new class named Dog where # instances (individual dogs) have 3 properties
# (fields): name, age, and breed
Dog = make_dataclass('Dog', ['name', 'age', 'breed'])
# Now we can create an instances of the Dog class:
dog1 = Dog(name='Dino', age=10, breed='shepherd')
print(dog1)      # prints: Dog(name='Dino', age=10, breed='shepherd')
print(dog1.name) # prints: Dino

dog1.name = 'Fred'
print(dog1)      # prints: Dog(name='Fred', age=10, breed='shepherd')
print(dog1.name) # prints: Fred

try:
    dog2 = Dog(name='Dino', age=10)
except Exception as e:
    print(e) # prints: missing 1 required positional argument: 'breed'

dog2 = Dog(name='Spot', age=12, breed='poodle')
dog3 = Dog(name='Fred', age=10, breed='shepherd')
print(dog1 == dog2) # prints: False
print(dog1 == dog3) # prints: True

print(type(dog1))      # prints <class 'types.Dog'>
print(isinstance(dog1, Dog)) # prints True
```

Objects and Aliases

- Alias là khả năng mà tại 1 ô nhớ có nhiều đối tượng cùng trả tới

```
# Objects are mutable so aliases change!
from types import SimpleNamespace
import copy

dog1 = SimpleNamespace(name='Dino', age=10, breed='shepherd')
dog2 = dog1      # this is an alias
dog3 = copy.copy(dog1) # this is a copy, not an alias

dog1.name = 'Spot'
print(dog2.name) # Spot (the alias changed, since it is the same object)
print(dog3.name) # Dino (the copy did not change, since it is a different object)
```

Classes

- Khai báo lớp trong Python sử dụng từ khóa **class**.

```
class MyNewClass:  
    """This is a docstring. I have created a new class"""  
    pass
```

a class must have a body, even if it does nothing, so we will use 'pass' for now...

```
class MyNewClass(object):  
    """This is a docstring. I have created a new class"""  
    pass
```

- Class tạo ra một local namespace mới trở thành nơi để các thuộc tính của nó được khai báo. Thuộc tính có thể là hàm hoặc dữ liệu.
- Ngoài ra còn có các thuộc tính đặc biệt bắt đầu với dấu gạch dưới kép (_). Ví dụ: __doc__ sẽ trả về chuỗi docstring mô tả của lớp đó.

Classes

- Ngay khi khai báo một lớp, một đối tượng trong lớp mới sẽ được tạo ra với cùng một tên. Đối tượng lớp này cho phép chúng ta truy cập các thuộc tính khác nhau cũng như để khởi tạo các đối tượng mới của lớp đó.

```
class Person:
```

```
    "This is a person class"
```

```
    age = 10
```

```
def greet(self):
```

```
    print('Hello')
```

```
# Output: 10
```

```
print(Person.age)
```

```
# Output: <function Person.greet>
```

```
print(Person.greet)
```

```
10
```

```
<function Person.greet at 0x000001C7AAE7DD30>
```

```
This is a person class
```

```
# Output: "This is a person class"
```

```
print(Person.__doc__)
```

Objects

- Đối tượng trong class có thể được sử dụng để truy cập các thuộc tính khác nhau và tạo các instance mới của lớp đó. Thủ tục để tạo một đối tượng tương tự như cách chúng ta gọi hàm.

```
harry = Person()
```

Lệnh này đã tạo ra một đối tượng mới có tên là *harry*.

Ví dụ

- Khi định nghĩa hàm trong class, ta có parameter là **self**, nhưng khi gọi hàm `harry.greet()` không cần parameter, vẫn không gặp lỗi.
- Bởi vì, bất cứ khi nào, object gọi các phương thức, object sẽ tự pass qua parameter đầu tiên. Nghĩa là `harry.greet()` tương đương với `Person.greet(harry)`

```
class Person:  
    "This is a person class"  
    age = 10  
  
    def greet(self):  
        print('Hello')  
  
# create a new object of Person class  
harry = Person()  
  
# Output: <function Person.greet>  
print(Person.greet)  
  
# Output: <bound method Person.greet of  
<__main__.Person object>>  
print(harry.greet)  
  
# Calling object's greet() method  
# Output: Hello  
harry.greet()
```

```
<function Person.greet at 0x000001C7AAE7D940>  
<bound method Person.greet of <__main__.Person object at 0x000001C7AAE7E4F0>>  
Hello
```

Ví dụ:

```
# Create our own class: # Tất cả các lớp trong Python đều kế thừa từ lớp  
object. (class object)
```

```
class Dog(object):
```

```
    # a class must have a body, even if it does nothing, so we will
```

```
    # use 'pass' for now...
```

```
    pass
```

```
# Create instances of our class:
```

```
d1 = Dog()
```

```
d2 = Dog()
```

```
# Verify the type of these instances:
```

```
print(type(d1))      # Dog (actually, class '__main__.Dog')
```

```
print(isinstance(d2, Dog)) # True
```

```
# Set and get properties (aka 'fields' or 'attributes') of these instances:
```

```
d1.name = 'Dot'
```

```
d1.age = 4
```

```
d2.name = 'Elf'
```

```
d2.age = 3
```

```
print(d1.name, d1.age) # Dot 4
```

```
print(d2.name, d2.age) # Elf 3
```

```
<class '__main__.Dog'>  
True  
Dot 4  
Elf 3
```

Constructors (hàm khởi tạo) *__init__()*

- Hàm trong Class được bắt đầu với dấu gạch dưới kép (_) là các hàm đặc biệt, mang các ý nghĩa đặc biệt.
- Hàm *__init__()*. Hàm này được gọi bất cứ khi nào khởi tạo một đối tượng, một biến mới trong class và được gọi là constructor trong lập trình hướng đối tượng.

Constructors

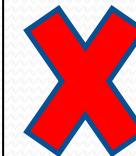
- Chúng ta mong muốn khởi tạo giá trị cho đối tượng trong lớp Dog (như ví dụ trước)

```
d1 = Dog('fred', 4) # now d1 is a Dog instance with name 'fred' and age 4
```

```
# Create our own class:  
class Dog:  
    pass
```

- Python không dùng tên hàm thông thường 'constructor' as the constructor

```
def constructor(dog, name, age):  
    # pre-load the dog instance with the given name and age:  
    dog.name = name  
    dog.age = age
```



Constructors

- Thay vào đó dùng hàm `__init__()`:

```
def __init__(dog, name, age):
    # pre-load the dog instance with the given name and age:
    dog.name = name
    dog.age = age
```



- Qui ước chuẩn (standard convention) chúng ta dùng tham số là `self`.
LUU Y: Các tham số `self` là tham chiếu đến lớp chính nó, và được sử dụng để truy cập các biến thuộc về lớp

```
def __init__(self, name, age):
    # pre-load the dog instance with the given name and age:
    self.name = name
    self.age = age
```

Tham chiếu self

- self giống như this trong các ngôn ngữ hướng đối tượng khác. Đối với các ngôn ngữ khác thì không cần phải truyền this hoặc self vào. Nhưng Python yêu cầu phải như thế.
- Các tham số self là tham chiếu đến lớp chính nó, và được sử dụng để truy cập các biến đó thuộc về lớp.
- Nó không nhất thiết được đặt tên self, ta có thể gọi nó là bất tên tùy thích, nhưng nó phải là tham số đầu tiên của bất kỳ hàm nào trong lớp:

Ví dụ

Sử dụng các từ **mysillyobject** và **abc** thay vì **self**:

```
class Person:  
    def __init__(mysillyobject, name, age):  
        mysillyobject.name = name  
        mysillyobject.age = age  
    def myfunc(abc):  
        print("Hello my name is " + abc.name)
```

```
p1 = Person("John", 36)  
p1.myfunc()
```

Hello my name is John

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def myfunc(self):  
        print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)  
p1.myfunc()
```

Hello my name is John

Constructors

- Phương thức khởi tạo nằm trong class.

```
class Dog(object):
    def __init__(self, name, age):
        # pre-load the dog instance with the given name and age:
        self.name = name
        self.age = age

    # Create instances of our class, using our new constructor
    d1 = Dog('Dot', 4)
    d2 = Dog('Elf', 3)

    print(d1.name, d1.age) # Dot 4
    print(d2.name, d2.age) # Elf 3
```

Methods (Phương thức)

- ❖ Ví dụ một function sayHi() nằm ngoài class

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Here is a function we will turn into a method:
    def sayHi(dog):
        print(f'Hi, my name is {dog.name} and I am {dog.age} years old!')

    d1 = Dog('Dot', 4)
    d2 = Dog('Elf', 3)

    sayHi(d1) # Hi, my name is Dot and I am 4 years old!
    sayHi(d2) # Hi, my name is Elf and I am 3 years old!
```

Methods

- Đưa function vào class khi đó gọi là method

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Now it is a method (simply by indenting it inside the class!)
    def sayHi(dog):
        print(f'Hi, my name is {dog.name} and I am {dog.age} years old!')

d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)

# Notice how we change the function calls into method calls:

d1.sayHi() # Hi, my name is Dot and I am 4 years old!
d2.sayHi() # Hi, my name is Elf and I am 3 years old!
```

Methods

- Sử dụng `self` như tham số cho phương thức trong class theo qui tắc chuẩn.

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Now we are using self, as convention requires:
    def sayHi(self):
        print(f'Hi, my name is {self.name} and I am {self.age} years old!')

d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)

# Notice how we change the function calls into method calls:

d1.sayHi() # Hi, my name is Dot and I am 4 years old!
d2.sayHi() # Hi, my name is Elf and I am 3 years old!
```

Methods

- Chúng ta có thể thêm tham số cho method.

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # This method takes a second parameter -- times
    def bark(self, times):
        print(f'{self.name} says: {"woof!" * times}')

d = Dog('Dot', 4)

d.bark(1) # Dot says: woof!
d.bark(4) # Dot says: woof!woof!woof!woof!
```

Methods

- Các phương thức cũng có thể thiết lập các thuộc tính

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.woofCount = 0 # we initialize the property in the constructor!

    def bark(self, times):
        # Then we can set and get the property in this method
        self.woofCount += times
        print(f'{self.name} says: {"woof!" * times} ({self.woofCount} woofs!)')

d = Dog('Dot', 4)

d.bark(1) # Dot says: woof!
d.bark(4) # Dot says: woof!woof!woof!woof!
```

Deleting Attributes

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

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Now we are using `self`, as convention requires:

```
def sayHi(self):
    print(f'Hi, my name is {self.name} and I am {self.age} years old!')
```

```
d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)
```

Notice how we change the function calls into method calls:

```
d1.sayHi() # Hi, my name is Dot and I am 4 years old!
d2.sayHi() # Hi, my name is Elf and I am 3 years old!
```

del d1.name

```
d1.sayHi()
```

```
Hi, my name is Dot and I am 4 years old!
Hi, my name is Elf and I am 3 years old!
```

```
-----  
AttributeError                               Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_20632/2512387124.py in <module>  
      18  
      19     del d1.name  
----> 20     d1.sayHi()  
  
~\AppData\Local\Temp\ipykernel_20632/2512387124.py in sayHi(self)  
      6     # Now we are using self, as convention requires:  
      7     def sayHi(self):  
----> 8         print(f'Hi, my name is {self.name} and I am {self.age} years old!')  
      9  
     10 d1 = Dog('Dot', 4)
```

AttributeError: 'Dog' object has no attribute 'name'

Deleting Attributes

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Now we are using self, as convention requires:
    def sayHi(self):
        print(f'Hi, my name is {self.name} and I am {self.age} years old!')
```

```
d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)
```

Notice how we change the function calls into method calls:

```
d1.sayHi() # Hi, my name is Dot and I am 4 years old!
d2.sayHi() # Hi, my name is Elf and I am 3 years old!
```

del d1.name
d2.sayHi()

```
Hi, my name is Dot and I am 4 years old!
Hi, my name is Elf and I am 3 years old!
Hi, my name is Elf and I am 3 years old!
```

Deleting Objects

- Có thể xóa chính đối tượng đó bằng cách sử dụng câu lệnh *del*.
- Sau khi bị 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.

```
class Dog(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Now we are using self, as convention requires:
    def sayHi(self):
        print(f'Hi, my name is {self.name} and I am {self.age} years old!')
```

```
d1 = Dog('Dot', 4)
d2 = Dog('Elf', 3)
```

Notice how we change the function calls into method calls:

```
d1.sayHi() # Hi, my name is Dot and I am 4 years old!
d2.sayHi() # Hi, my name is Elf and I am 3 years old!
```

```
del d1
d1
```

```
Hi, my name is Dot and I am 4 years old!
Hi, my name is Elf and I am 3 years old!
```

```
NameError
~\AppData\Local\Temp\ipykernel_20632/356380136.py in <module>
      18
      19 del d1
----> 20 d1

NameError: name 'd1' is not defined
```

Inheritance (Kế thừa)

- Kế thừa (Inheritance) cho phép một lớp (class) có thể kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa.
- Lớp đã gọi là lớp cha (base class hoặc parent class), lớp mới phát sinh 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.

Python Inheritance Syntax

- **Syntax**

```
class BaseClass:  
    Body of base class
```

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

Ví dụ: Polygon (Đa giác)

```
class Polygon:  
    def __init__(self, no_of_sides):  
        self.n = no_of_sides  
        self.sides = [0 for i in range(no_of_sides)]  
  
    def inputSides(self):  
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]  
  
    def dispSides(self):  
        for i in range(self.n):  
            print("Side",i+1,"is",self.sides[i])
```

Class Polygon có thuộc tính n để định nghĩa số cạnh và sides để lưu giá trị mỗi cạnh. Hàm inputSides() nhập vào độ lớn các cạnh và dispSides() sẽ hiện thị danh sách các cạnh của đa giác.

Ví dụ: Polygon (Đa giác)

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

```
In [6]: t = Triangle()
t.inputSides()
t.findArea()
```

```
Enter side 1 : 3
Enter side 2 : 4
Enter side 3 : 5
The area of the triangle is 6.00
```

Hình tam giác là đa giác có ba cạnh, nên ta sẽ tạo một lớp Triangle kế thừa từ Polygon.

Class mới này sẽ thừa kế tất cả các thuộc tính sẵn có trong lớp cha nên sẽ không cần khai báo lại (khả năng sử dụng lại code).

Lớp Triangle không chỉ kế thừa mà còn định nghĩa thêm một hàm mới là hàm findArea.

Overriding (Ghi đè)

- Python cho phép ghi đè lên các phương thức của lớp cha. Bạn có thể thực hiện việc ghi đè phương thức của lớp cha nếu muốn có tính năng khác biệt hoặc đặc biệt trong lớp con.
- Trong ví dụ trên ta thấy class Triangle sử dụng lại hàm `__init__()` từ class Polygon, nếu bạn muốn override (ghi đè) lại định nghĩa của hàm `__init__()` trong class cha, ta dùng **hàm super()**.

super () .`__ init __` (3) tương đương với `Polygon.__ init __ (self, 3)`

```
class Triangle(Polygon):
    def __ init __(self):
        super().__ init __(3)
```

```
class Triangle(Polygon):
    def __ init __(self):
        Polygon.__ init __(self,3)
```

Inheritance

- Ngoài ra Python có hai hàm *isinstance()* và *issubclass()* được dùng để kiểm tra mối quan hệ của hai lớp và instance.
- Hàm *isinstance(obj, Class)* trả về True nếu obj là một instance của lớp Class hoặc là một instance của lớp con của Class.
- Hàm *issubclass(classA, classB)* trả về True nếu class A là lớp con của class B.
- Tất cả các lớp trong Python điều kết thừa từ lớp **object**. (class object)

Ví dụ:

```
t = Triangle()  
isinstance(t,Triangle) # True  
isinstance(t,Polygon) # True  
isinstance(t,int)     # False  
isinstance(t,object)  # True  
issubclass(Polygon,Triangle) # False  
issubclass(Triangle,Polygon) # True  
issubclass(bool,int)       # True
```

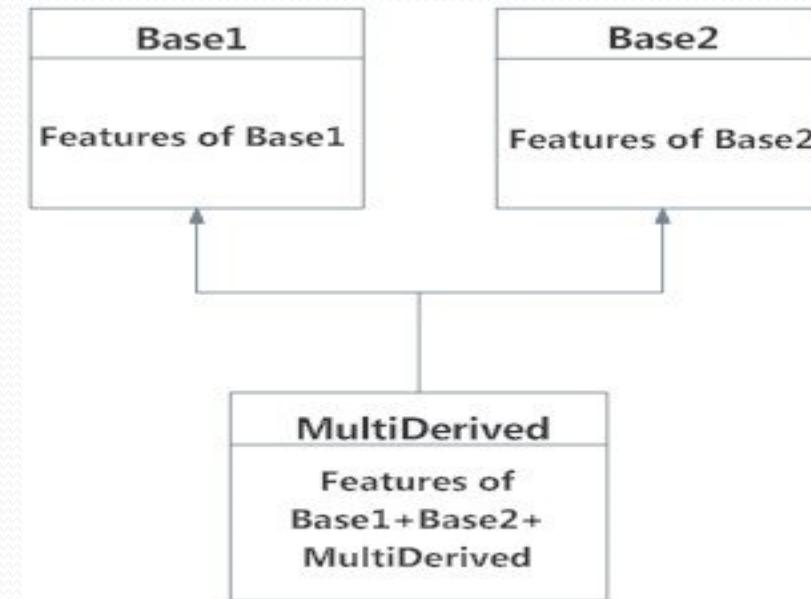
Multiple Inheritance (Đa kế thừa)

- Tương tự như trong C++, trong **Python** một lớp có thể được định nghĩa từ nhiều lớp cha. Điều này được gọi là đa kế thừa.
- Lớp con được định nghĩa từ nhiều lớp cha và kế thừa đặc tính của cả các lớp.
- Các lớp cha có thể có các thuộc tính hoặc các 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.

Multiple Inheritance

- Syntax

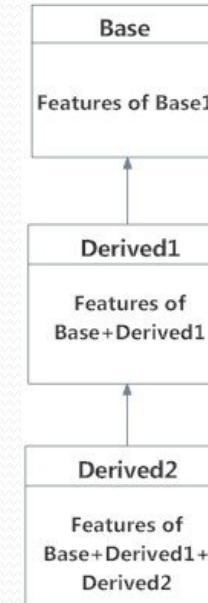
```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class MultiDerived(Base1, Base2):  
    pass
```



Multilevel Inheritance (Kế thừa đa cấp)

- Ngoài việc có thể kế thừa từ các lớp cha, chúng ta còn có thể tạo lớp con mới kế thừa các lớp con trước đó. Đây gọi là kế thừa đa cấp (Multilevel Inheritance).
- Với trường hợp này, các đặc tính của lớp cha và lớp con trước đó sẽ được lớp con mới kế thừa.

```
class Base:  
    pass  
  
class Derived1(Base):  
    pass  
  
class Derived2(Derived1):  
    pas
```



Method Resolution Order

Thứ tự truy xuất phương thức (Method Resolution Order)

- Class được bắt nguồn từ **object**. Trong kịch bản đa thừa kế, bất kỳ thuộc tính cần được truy xuất nào, đầu tiên sẽ được tìm kiếm trong lớp hiện tại. Nếu không tìm thấy, tìm kiếm tiếp tục vào lớp cha đầu tiên và từ trái qua phải.
- Vậy thứ tự truy xuất sẽ là **[MultiDerived, Base1, Base2, object]**.

```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class MultiDerived(Base1, Base2):  
    pass
```

Method Resolution Order

- Ví dụ: Class object

```
print(issubclass(list,object))    # True  
print(isinstance(5.5,object))    # True  
print(isinstance("Hello",object)) # True
```

Method Resolution Order

- Vậy thứ tự truy xuất sẽ là [MultiDerived, Base1, Base2, object].
- Thứ tự này còn được gọi là tuyến tính hóa (**linearization**) của MultiDerived và tập hợp các quy tắc được sử dụng để tìm thứ tự này được gọi là Thứ tự truy xuất phương thức (**MRO**).
- MRO dùng để hiển thị danh sách/tuple các class cha của một class nào đó.

```
class Base1:  
    pass
```

```
class Base2:  
    pass
```

```
class MultiDerived(Base1, Base2):  
    pass
```

MRO được sử dụng theo hai cách:

- `__mro__`: trả về một tuple
- `mro()`: trả về một danh sách.

MRO

- Ví dụ:

```
class Base1:  
    pass  
  
class Base2:  
    pass  
  
class MultiDerived(Base1, Base2):  
    pass
```

```
MultiDerived.__mro__
```

```
In [15]: MultiDerived.__mro__
```

```
Out[15]: (__main__.MultiDerived, __main__.Base1, __main__.Base2, object)
```

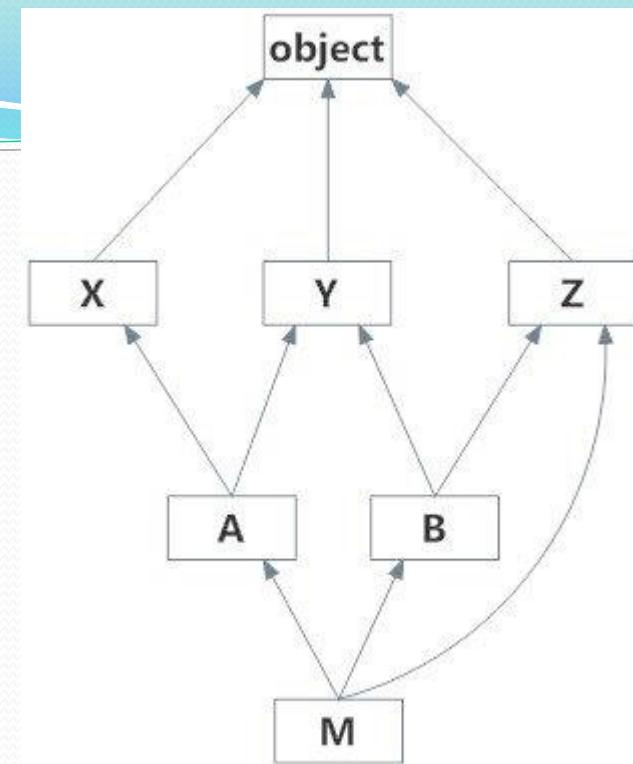
```
MultiDerived.mro()
```

```
In [16]: MultiDerived.mro()
```

```
Out[16]: [__main__.MultiDerived, __main__.Base1, __main__.Base2, object]
```

Ví dụ

```
# Demonstration of 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'>]
print(M.mro())
```



Visualizing Multiple Inheritance in Python

Operator Overloading

- Toán tử Python làm việc bằng các hàm được dựng sẵn (built-in), nhưng một toán tử có thể được sử dụng để thực hiện nhiều hoạt động khác nhau.
- Ví dụ với toán tử '+', bạn có thể cộng số học hai số với nhau, có thể kết hợp hai danh sách, hoặc nối hai chuỗi khác nhau lại...
- Tính năng này trong Python gọi là nạp chồng toán tử, cho phép cùng một toán tử được sử dụng khác nhau tùy từng ngữ cảnh.

Operator Overloading

- Ví dụ:

- Chương trình ngay lập tức báo lỗi *TypeError* vì Python không thể nhận hai đối tượng *Point* cùng lúc.
- Để xử lý vấn đề này, ta sẽ sử dụng nạp chồng toán tử.

```
class Point:  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y  
  
p1 = Point(1, 2)  
p2 = Point(2, 3)  
print(p1+p2) # Error
```

```
-----  
TypeError                                         Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_18076\3332101378.py in <module>  
      7 p1 = Point(1, 2)  
      8 p2 = Point(2, 3)  
----> 9 print(p1+p2)  
  
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Python Special Functions

- Hàm trong Class được bắt đầu với hai dấu gạch dưới liền nhau (`__`) là các hàm đặc biệt, mang các ý nghĩa đặc biệt.
- Có rất nhiều hàm đặc biệt trong Python và một trong đó là hàm `__init__()`. Hàm này được gọi bất cứ khi nào khởi tạo một đối tượng, một biến mới trong class.
- Mục đích khi sử dụng các hàm đặc biệt này là giúp những hàm của chúng ta tương thích với các hàm được dựng sẵn trong Python.

<https://docs.python.org/3/reference/datamodel.html#special-method-names>

Ví dụ

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

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

```
<__main__.Point object at 0x000001F823BE8430>
```

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

Ví dụ

- Sử dụng `__str__()` làm kết quả hiển thị chuẩn hơn. Ngoài ra bạn có thể in ra kết quả tương tự bằng cách sử dụng hàm tích hợp sẵn trong Python là `str()` hoặc `format()`.

```
In [5]: str(p1)
```

```
Out[5]: '(2,3)'
```

```
In [6]: format(p1)
```

```
Out[6]: '(2,3)'
```

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

Ví dụ: Overloading the + Operator

- Để nạp chồng toán tử '+' , ta sẽ sử dụng hàm `__add__()` trong class. Ta có thể cộng hai điểm tọa độ ở ví dụ bên trên.

Ở chương trình này, khi bạn thực hiện `p1 + p2`, Python sẽ gọi ra `p1.__add__(p2)` hay `Point.__add__(p1,p2)`

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

- Tương tự như vậy, bạn có thể nạp chồng nhiều toán tử khác.

Operator	Expression	Internally
Addition	<code>p1 + p2</code>	<code>p1.__add__(p2)</code>
Subtraction	<code>p1 - p2</code>	<code>p1.__sub__(p2)</code>
Multiplication	<code>p1 * p2</code>	<code>p1.__mul__(p2)</code>
Power	<code>p1 ** p2</code>	<code>p1.__pow__(p2)</code>
Division	<code>p1 / p2</code>	<code>p1.__truediv__(p2)</code>
Floor Division	<code>p1 // p2</code>	<code>p1.__floordiv__(p2)</code>
Remainder (modulo)	<code>p1 % p2</code>	<code>p1.__mod__(p2)</code>
Bitwise Left Shift	<code>p1 << p2</code>	<code>p1.__lshift__(p2)</code>
Bitwise Right Shift	<code>p1 >> p2</code>	<code>p1.__rshift__(p2)</code>
Bitwise AND	<code>p1 & p2</code>	<code>p1.__and__(p2)</code>
Bitwise OR	<code>p1 p2</code>	<code>p1.__or__(p2)</code>
Bitwise XOR	<code>p1 ^ p2</code>	<code>p1.__xor__(p2)</code>
Bitwise NOT	<code>~p1</code>	<code>p1.__invert__()</code>

Ví dụ: Overloading Comparison Operators

- Python không chỉ giới hạn được phép nạp chồng các toán tử toán học, mà còn cho phép người dùng nạp chồng toán tử so sánh.
- Có nhiều toán tử so sánh được hỗ trợ bởi Python, ví dụ như: <, >, <=, >=, ==, ...
- Bạn sử dụng nạp chồng toán tử này khi muốn so sánh các đối tượng trong lớp với nhau.

Ví dụ:

- Ví dụ bạn muốn so sánh các điểm trong class *Point*, hãy so sánh độ lớn của các điểm này bắt đầu từ gốc tọa độ,

```
# overloading the less than operator
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

p1 = Point(1,1)
p2 = Point(-2,-3)
p3 = Point(1,-1)

# use less than
print(p1<p2)      # True
print(p2<p3)      # False
print(p1<p3)      # False
```

Overloading Comparison Operators

- Tương tự như vậy, bạn có thể nạp chồng nhiều toán tử khác.

Operator	Expression	Internally
Less than	<code>p1 < p2</code>	<code>p1.__lt__(p2)</code>
Less than or equal to	<code>p1 <= p2</code>	<code>p1.__le__(p2)</code>
Equal to	<code>p1 == p2</code>	<code>p1.__eq__(p2)</code>
Not equal to	<code>p1 != p2</code>	<code>p1.__ne__(p2)</code>
Greater than	<code>p1 > p2</code>	<code>p1.__gt__(p2)</code>
Greater than or equal to	<code>p1 >= p2</code>	<code>p1.__ge__(p2)</code>

Polymorphism

- It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.
- **Example: Polymorphism in addition operator**

In [13]:

```
num1 = 1
num2 = 2
print(num1+num2)
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

3

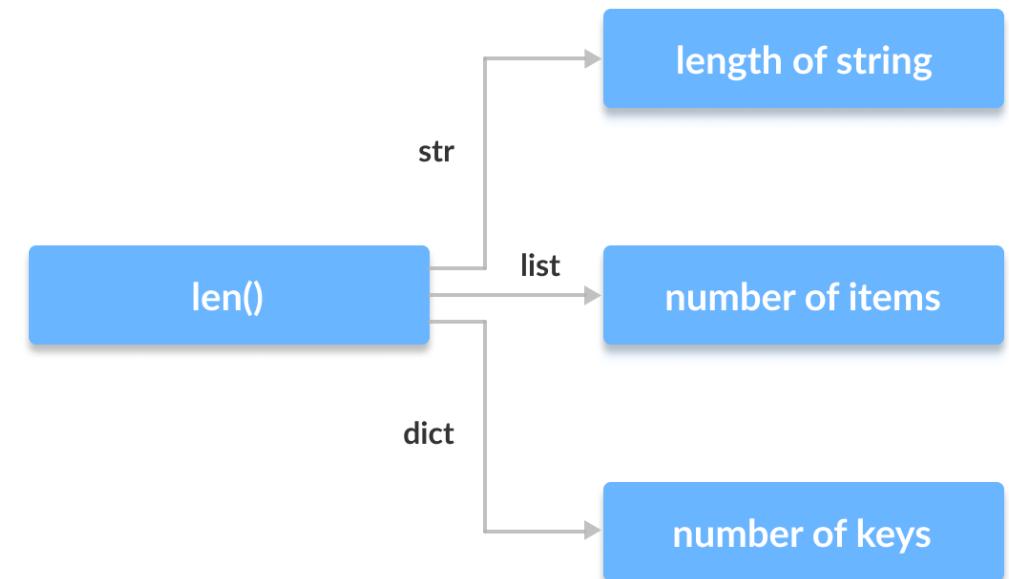
Python Programming

Function Polymorphism in Python

- Có một số hàm trong Python tương thích để chạy với nhiều kiểu dữ liệu.
- **Example: Polymorphic len() function**

```
In [14]: print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

```
9
3
2
```



Polymorphism in `len()` function in Python

Class Polymorphism

- Chúng ta có thể sử dụng khái niệm đa hình trong khi tạo các phương thức vì Python cho phép các lớp khác nhau có các phương thức trùng tên.
- Sau đó, chúng ta có thể tổng quát hóa việc gọi các phương thức này bằng cách bỏ qua đối tượng mà chúng ta đang làm việc.

```
class Dog(object):
    def speak(self):
        print('woof!')

class Cat(object):
    def speak(self):
        print('meow!')

for animal in [ Dog(), Cat() ]:
    animal.speak() # same method name, but one woofs and one meows!
```

Polymorphism

- **Đa hình với hàm**

Có 2 lớp English và French. Cả hai lớp này đều có phương thức greeting().

Tạo ra 2 đối tượng tương ứng từ 2 lớp trên và gọi hành động của 2 đối tượng này trong cùng một hàm

```
class English:  
    def greeting(self):  
        print ("Hello")
```

```
class French:  
    def greeting(self):  
        print ("Bonjour")
```

```
def intro(language):  
    language.greeting()
```

```
e = English()  
f = French()  
intro(e)  
intro(f)
```

Polymorphism and Inheritance

- Tính đa hình cho phép chúng ta truy cập các phương thức và thuộc tính bị ghi đè này có cùng tên với lớp cha.

```

from math import pi
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def fact(self):
        return "I am a two-dimensional shape."
    def __str__(self):
        return self.name

class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length
    def area(self):
        return self.length**2
    def fact(self):
        return "Squares have each angle equal to 90 degrees."

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return pi*self.radius**2

```

```

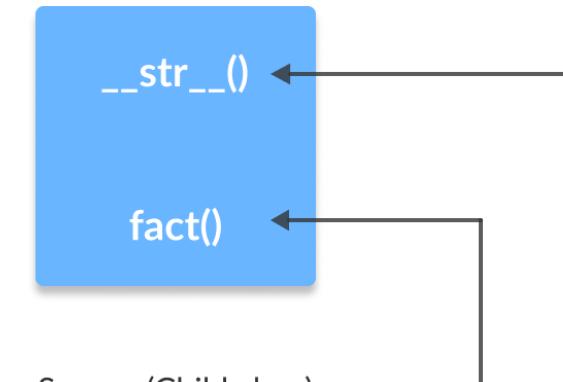
a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())

```

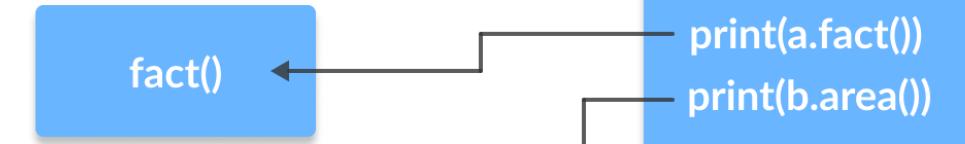
Circle

I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985

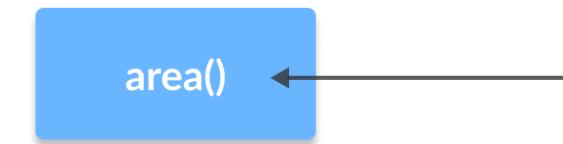
Shape (Parent class)



Square (Child class)



Circle (Child class)



Các ví dụ: Equality Testing (`__eq__`)

```
class A(object):
    def __init__(self, x):
        self.x = x
    a1 = A(5)
    a2 = A(5)
    print(a1 == a2) # False!
```

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return (self.x == other.x)
    a1 = A(5)
    a2 = A(5)
    print(a1 == a2) # True
    print(a1 == 99) # crash
```

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return (isinstance(other, A) and (self.x == other.x))
    a1 = A(5)
    a2 = A(5)
    print(a1 == a2) # True
    print(a1 == 99) # False
```

Ví dụ: Converting to Strings (`__str__` and `__repr__`)

```
class A(object):
    def __init__(self, x):
        self.x = x
a = A(5)
print(a) # prints <__main__.A object at 0x102916128>
```

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return f'A(x={self.x})'
a = A(5)
print(a) # prints A(x=5) (better)
print([a]) # prints [<__main__.A object at 0x102136278>]
```

Note: repr should be a computer-readable form so that
(eval(repr(obj)) == obj), but we are not using it that way.
So this is a simplified use of repr.

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __repr__(self):
        return f'A(x={self.x})'
a = A(5)
print(a) # prints A(x=5) (better)
print([a]) # [A(x=5)]
```

Ví dụ: Static Methods

- Static Methods in a class can be called directly without necessarily making and/or referencing a specific object.

```
class A(object):  
    @staticmethod  
    def f(x):  
        return 10*x
```

```
print(A.f(42)) # 420 (called A.f without creating an instance of A)
```

Additional Reading

- For more on these topics, and many additional OOP-related topics, check the following links:

<https://docs.python.org/3/tutorial/classes.html>

<https://docs.python.org/3/reference/datamodel.html>

