LẬP TRÌNH PYTHON CƠ BẢN

BÀI 5

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG TRONG PYTHON

NỘI DUNG

1. Giới thiệu
2. Cài đặt lớp
3. Quan hệ kế thừa
4. Quan hệ hợp thành
5. Lớp cơ sở trừu tượng
6. Tính đa hình

5. LỚP CƠ SỞ TRỪU TƯỢNG

5. LỚP CƠ SỞ TRỪU TƯỢNG

Lớp trừu tượng (Abstract class)

- Một ***Abstract class*** có thể được coi là một bản thiết kế cho các class khác, cho phép tạo một tập hợp các phương thức mà phải được tạo trong bất kỳ class con nào được xây dựng từ Abstract class của bạn.
- Một class có chứa một hoặc các ***phương thức abstract*** được gọi là một ***Abstract class***.
- Một ***phương thức abstract*** là một phương thức có khai báo nhưng không có bất kỳ triển khai nào.
- Các ***Abstract class*** không thể khởi tạo và nó cần các class con để triển khai cho các ***phương thức abstract*** được định nghĩa trong các ***Abstract class***.

5. LỚP CƠ SỞ TRỪU TƯỢNG

Lớp cơ sở trừu tượng (Abstract Base classes - ABC)

- Mặc định trong Python sẽ không cung cấp Abstract class cho chúng ta sử dụng.
- Python có một mô-đun gọi là Abstract Base Classes (ABC), cần import trước khi sử dụng.
- Để khai báo được phương thức abstract, cần import thêm mô-đun ***abstractmethod*** cùng với từ khóa ***@abstractmethod***

```
from abc import ABC, abstractmethod
```

5. LỚP CƠ SỞ TRỪU TƯỢNG

Lớp cơ sở trừu tượng (Abstract Base classes - ABC)

- Giả sử, ta tạo một lớp cơ sở trừu tượng dùng làm mẫu cho các lớp hình dạng khác nhau.
- Tất cả các lớp kế thừa lớp Shape sẽ luôn có:
  - Phương thức tính chu vi (get_perimeter)
  - Phương thức tính diện tích (get_area)

```
# shapes_abc.py

from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass

    @abstractmethod
    def get_perimeter(self):
        pass
```

Lớp cơ sở trừu tượng

- Tiếp tục tạo lớp Circle kế thừa lớp Shape.
- Để có thể khởi tạo đối tượng circle, cần cung cấp các triển khai phù hợp cho tất cả các phương thức trừu tượng của nó.
- Khi đã xác định cách triển khai tùy chỉnh phù hợp cho tất cả các phương thức trừu tượng, ta có thể tiến hành khởi tạo circle.

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

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

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

circle = Circle(100)
```

```
Traceback (most recent call last):
    ...
TypeError: Can't instantiate abstract class Circle
    with abstract method get_perimeter
```

Lớp cơ sở trừu tượng

- Tương tự, ta có thể tạo ra lớp Square với đầy đủ các phương thức trừu tượng của lớp cơ sở trừu tượng.

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

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

    def get_perimeter(self):
        return 2 * pi * self.radius
```

```
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def get_area(self):
        return self.side ** 2

    def get_perimeter(self):
        return 4 * self.side
```

- Trong Python, có thể sử dụng ***static duck typing*** để thay thế cho lớp cơ sở trừu tượng.

BÀI TẬP 5.5

Viết chương trình Python thực hiện các yêu cầu sau:

- Tạo một lớp cơ sở trừu tượng **Vehicle**.
  - Các phương thức trừu tượng: start_engine(), stop_engine(), và drive().
- Tạo các lớp con **Car**, **Motobike** kế thừa từ **Vehicle**.
  - **Car** gồm thuộc tính số ghế ngồi (seats) và các phương thức cần thiết.
  - **Motobike** gồm thuộc tính dung tích bình xăng (fuel) và các phương thức cần thiết.
- Khởi tạo đối tượng thuộc các lớp Car, Motobike và sử dụng các phương thức để thao tác lên đối tượng.

BÀI TẬP 5.5

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass
```

```python
class Car(Vehicle):
    def __init__(self, seats):
        self.seats = seats
        self.engine_running = False

    def start_engine(self):
        self.engine_running = True
        print("Car engine started.")

    def stop_engine(self):
        self.engine_running = False
        print("Car engine stopped.")
```

```python
class Motobike(Vehicle):
    def __init__(self, fuel):
        self.fuel = fuel
        self.engine_running = False

    def start_engine(self):
        self.engine_running = True
        print("Motobike engine started.")

    def stop_engine(self):
        self.engine_running = False
        print("Motobike engine stopped.")
```

```python
vehicles = [
    Car(seats=4),
    Motobike(fuel=6)
]
```

```python
for vehicle in vehicles:
    print(f"\n{vehicle.__class__.__name__}:")
    vehicle.start_engine()
    vehicle.drive()
    vehicle.stop_engine()
```

5. LỚP CƠ SỞ TRỪU TƯỢNG

Duck typing

- Duck typing là một khái niệm trong lập trình hướng đối tượng, phổ biến trong các ngôn ngữ lập trình động như Python, Ruby, và JavaScript.
- Thuật ngữ này xuất phát từ câu nói "Nếu nó đi như vịt, kêu như vịt, thì nó là vịt".
- Ý tưởng cơ bản của duck typing là việc xác định khả năng của một đối tượng dựa trên các phương thức và thuộc tính mà nó cung cấp, hơn là dựa trên loại (type) của đối tượng đó.

Duck typing

Ví dụ:

```
class Duck:
    def quack(self):
        print("Quack!")

class Robot:
    def quack(self):
        print("I'm not a duck but I can quack!")
```

```
def make_it_quack(entity):
    entity.quack()

donald = Duck()
doraemon = Robot()

make_it_quack(donald)
make_it_quack(doraemon)
```

```
Quack!
I'm not a duck but I can quack!
```

Duck typing

- Đặc điểm của Duck Typing
  - Không quan trọng loại đối tượng: Trong duck typing, quan trọng là đối tượng có các phương thức hoặc thuộc tính mà bạn muốn sử dụng, không quan trọng đối tượng đó là thể hiện của lớp nào.
  - Linh hoạt và dễ mở rộng: Duck typing cho phép viết mã linh hoạt và dễ dàng mở rộng, có thể sử dụng bất kỳ đối tượng nào miễn là nó có các phương thức hoặc thuộc tính cần thiết.
  - Kiểm tra lúc chạy (runtime): Việc kiểm tra các thuộc tính và phương thức được thực hiện lúc chạy, giúp chương trình có thể hoạt động linh hoạt hơn.

Static duck typing

- Static duck typing trong Python có thể được thực hiện bằng cách sử dụng các công cụ kiểm tra kiểu tĩnh (static type checking) như MyPy và sử dụng các giao diện kiểu như Protocol từ module typing.
- Điều này cho phép kết hợp sự linh hoạt của duck typing với sự an toàn của kiểm tra kiểu tĩnh.
- Cần cài đặt mô-đun MyPy để thực hiện.

```
pip install mypy
```

5. LỚP CƠ SỞ TRỪU TƯỢNG

Static duck typing

- ***Quackable*** được định nghĩa bằng cách sử dụng Protocol.
- ***Quackable*** yêu cầu bất kỳ đối tượng nào tuân theo giao diện này phải có phương thức ***quack*** mà không cần chỉ rõ đối tượng đó thuộc lớp nào.

```
from typing import Protocol

class Quackable(Protocol):
    def quack(self) -> None:
        ...
```

```
class Duck:
    def quack(self) -> None:
        print("Quack!")

class Robot:
    def quack(self) -> None:
        print("I'm not a duck but I can quack!")
```

```
def make_it_quack(duck: Quackable) -> None:
    duck.quack()

donald = Duck()
doraemon = Robot()

make_it_quack(donald)
make_it_quack(doraemon)
```

```
Quack!
I'm not a duck but I can quack!
```

BÀI TẬP 5.6

Viết chương trình Python thực hiện các yêu cầu sau:

- Tạo một lớp **Engine** với các phương thức start() và stop().
- Tạo các lớp Car và Motobike với các thuộc tính và phương thức cần thiết.
- Sử dụng static duck typing để đảm bảo các đối tượng **Car** và **Motobike** có thể được sử dụng thay thế cho nhau trong các hàm sử dụng giao diện Engine.
- Khởi tạo đối tượng thuộc các lớp **Car**, **Motobike** và sử dụng các phương thức để thao tác lên đối tượng.

BÀI TẬP 5.6

```python
from typing import Protocol

class Engine(Protocol):
    def start(self) -> None:
        ...

    def stop(self) -> None:
        ...
```

```python
class Car:
    def __init__(self, seats: int):
        self.seats = seats
        self.engine_running = False

    def start(self) -> None:
        self.engine_running = True
        print("Car engine started.")

    def stop(self) -> None:
        self.engine_running = False
        print("Car engine stopped.")
```

```python
class Motobike:
    def __init__(self, fuel: float):
        self.fuel = fuel
        self.engine_running = False

    def start(self) -> None:
        self.engine_running = True
        print("Motobike engine started.")

    def stop(self) -> None:
        self.engine_running = False
        print("Motobike engine stopped.")
```

```python
def operate_vehicle(vehicle: Engine) -> None:
    vehicle.start()
    vehicle.stop()
```

```python
my_car = Car(seats=4)
my_bike = Motobike(fuel=5)
operate_vehicle(my_car)
operate_vehicle(my_bike)
```

6. TÍNH ĐA HÌNH

6. TÍNH ĐA HÌNH

Tính đa hình (Polymorphism)

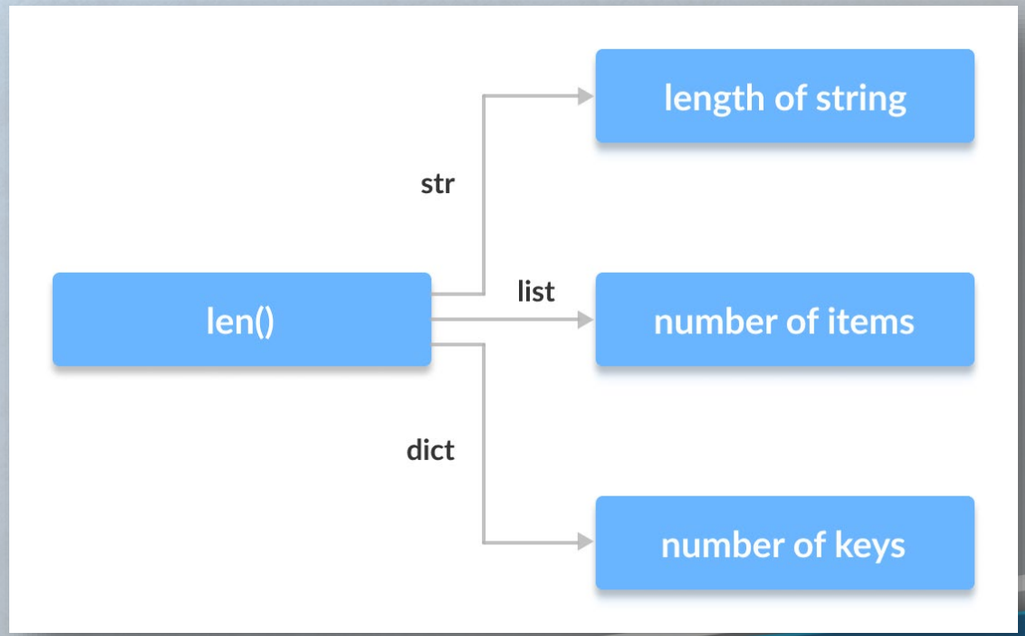
- Theo nghĩa đen, đa hình tức là nhiều hình dạng, có nghĩa là cùng một đối tượng nhưng lại thể hiện khác nhau trong các thời điểm khác nhau.
- Đa hình là một tính năng trong OOP, nó sử dụng một giao diện chung cho nhiều kiểu dữ liệu khác nhau.
- Giả sử, chúng ta cần tô màu cho một hình dạng, có nhiều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

Tính đa hình (Polymorphism)

- Có một số hàm trong Python có thể tương thích để được thực thi với nhiều kiểu dữ liệu. Một trong những hàm như vậy là hàm len(). Nó có thể thực thi với nhiều kiểu dữ liệu trong Python.

```
print(len("Lập trình Python cơ bản"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "Nguyễn Văn A", "Address": "Hà Nội"}))
```

```
22
3
2
```



Tính đa hình (Polymorphism)

- Chúng ta biết rằng toán tử + được sử dụng rộng rãi trong các chương trình Python. Nhưng, nó không chỉ có một cách sử dụng duy nhất. Đối với kiểu dữ liệu số nguyên, toán tử + được sử dụng để thực hiện phép toán cộng số học.
- Tương tự, đối với kiểu dữ liệu chuỗi, toán tử + được sử dụng để thực hiện nối các chuỗi ký tự lại với nhau.

```
num1 = 1
num2 = 2
print(num1+num2)

str1 = 'Xin '
str2 = 'chào!'
print(str1+str2)
```

```
3
Xin chào!
```

Tính đa hình (Polymorphism)

- Tính đa hình đối với lớp
  - Để sử dụng tính đa hình, chúng ta đã tạo một giao diện chung, là hàm **vi_du()** nhận bất kỳ đối tượng nào và gọi phương thức **in_thong_tin()** tương ứng của đối tượng đó.

```
class SinhVienY:
    def in_thong_tin(self):
        print("Sinh viên trường Y")

class SinhVienLuat:
    def in_thong_tin(self):
        print("Sinh viên trường Luật")
```

```
def vi_du(sv):
    sv.in_thong_tin()

sv1 = SinhVienLuat()
sv2 = SinhVienY()
vi_du(sv1)
vi_du(sv2)
```

```
Sinh viên trường Luật
Sinh viên trường Y
```

6. TÍNH ĐA HÌNH

Tính đa hình (Polymorphism)

- Tính đa hình đối với lớp
  - Trong ví dụ trên, ta chưa liên kết cả hai lớp cũng như chưa sử dụng tính năng kế thừa.
  - Do tính đa hình nên khi chúng ta thêm cùng một phương thức vào cả hai lớp thì Python sẽ kiểm tra kiểu lớp của đối tượng và thực thi phương thức có trong lớp tương ứng của nó.
  - Khi đó, phương thức với đối tượng nào thì nó sẽ hoạt động theo đúng đối tượng đó.

Tính đa hình (Polymorphism)

- Tóm lại:
  - Đa hình là tính chất mà một phương thức cùng tên thể hiện khác nhau với từng đối tượng dữ liệu
  - Các lớp khác nhau có thể có các phương thức trùng tên, các đối tượng tương ứng với lớp sẽ gọi đến các phương thức tương ứng của lớp đó và không gọi đến phương thức cùng tên của lớp khác.
  - Khi lớp con kế thừa lớp cha có cùng các phương thức trùng tên, thì khi đối tượng của lớp con gọi đến phương thức đó, nó sẽ chọn ưu tiên đối tượng được định nghĩa trong lớp con, được gọi là ghi đè (overloading).

6. TÍNH ĐA HÌNH

Nạp chồng toán tử

- Trong Python, chúng ta có thể thay đổi cách các toán tử hoạt động đối với các kiểu do người dùng xác định.
- Ví dụ: toán tử + sẽ thực hiện phép cộng số học trên hai số, hợp nhất hai danh sách hoặc nối hai chuỗi.
- Tính năng này trong Python cho phép cùng một toán tử có ý nghĩa khác nhau tùy theo ngữ cảnh được gọi là nạp chồng toán tử.

6. TÍNH ĐA HÌNH

Nạp chồng toán tử

- Để thực hiện việc nạp chồng toán tử, chúng ta cần phải định nghĩa các phương thức có tên đặc biệt cho phép thay thế các toán tử tương ứng.
- Các phương thức này được bắt đầu bằng các dấu gạch dưới, giống như phương thức __init__() đã được học.

Nạp chồng toán tử

- Ví dụ để nạp chồng toán tử dấu cộng +, chúng ta sẽ cần triển khai hàm __add__() trong lớp.

```
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)

# Output: (3,5)
```

Nạp chồng toán tử

- Một số các hàm đặc biệt:

Toán tử	Biểu thức	Hàm để định nghĩa nạp chồng
Phép cộng	a + b	a.__add__(b)
Phép trừ	a – b	a.__sub__(b)
Phép nhân	a * b	a.__mul__(b)
Phép số mũ	a ** b	a.__pow__(b)
Phép chia	a / b	a.__truediv__(b)
Phép chia làm tròn	a // b	a.__floordiv__(b)
Phép chia lấy dư	a % b	a.__mod__(b)

Nạp chồng toán tử

- Một số các hàm đặc biệt:

Toán tử	Biểu thức	Hàm để định nghĩa nạp chồng
Dịch bit sang trái	a << b	a.__lshift__(b)
Dịch bit sang phải	a >> b	a.__rshift__(b)
Phép AND	a & b	a.__and__(b)
Phép OR	a | b	a.__or__(b)
Phép XOR	a ^ b	a.__xor__(b)
Phép NOT	~a	a.__invert__()

Nạp chồng toán tử

- Một số các hàm đặc biệt:

Toán tử	Biểu thức	Hàm bên trong
Nhỏ hơn	a < b	a.__lt__(b)
Nhỏ hơn hoặc bằng	a <= b	a.__le__(b)
Bằng	a == b	a.__eq__(b)
Không bằng/ Khác	a != b	a.__ne__(b)
Lớn hơn	a > b	a.__gt__(b)
Lớn hơn hoặc bằng	a >= b	a.__ge__(b)

Nạp chồng toán tử

- Tóm lại:
  - Chúng ta có thể định nghĩa bất cứ một toán tử nào trong một lớp để thay thế cho các phép toán thông thường mặc định của Python để nó có thể sử dụng để làm việc như các phép toán này
  - Việc nạp chồng toán tử có mục tiêu là làm cho chương trình viết dễ hiểu và rõ ràng hơn do sử dụng các toán tử quen thuộc hàng ngày
  - Chúng ta buộc phải nhớ các tên hàm đặc biệt cần sử dụng để thực hiện việc nạp chồng một toán tử tương ứng ví dụ __add__(), __mul__(), __sub__(),...các tên này buộc phải đặt đúng thì việc nạp chồng toán tử mới diễn ra.

BÀI TẬP 5.7

Viết chương trình Python thực hiện các yêu cầu sau:

- Viết một lớp Phân số để biểu diễn các phân số. Lớp này cần có các thuộc tính Tử số và Mẫu số. Nạp chồng các toán tử sau:
  - __add__: Toán tử cộng để thực hiện phép cộng hai phân số.
  - __mul__: Toán tử nhân để thực hiện phép nhân hai phân số.
  - __eq__: Toán tử so sánh bằng để so sánh hai phân số.
  - __str__: Nạp chồng toán tử chuỗi để hiển thị phân số dưới dạng chuỗi.
- Tạo 2 đối tượng phân số, sử dụng các toán tử đã được định nghĩa.

BÀI TẬP 5.7

```
class PhanSo:
    def __init__(self, tu_so, mau_so):
        self.tu_so = tu_so
        self.mau_so = mau_so

    def __add__(self, ps):
        tu_so_moi = self.tu_so * ps.mau_so + ps.tu_so * self.mau_so
        mau_so_moi = self.mau_so * ps.mau_so
        return PhanSo(tu_so_moi, mau_so_moi)

    def __mul__(self, ps):
        tu_so_moi = self.tu_so * ps.tu_so
        mau_so_moi = self.mau_so * ps.mau_so
        return PhanSo(tu_so_moi, mau_so_moi)

    def __eq__(self, ps):
        return self.tu_so * ps.mau_so == ps.tu_so * self.mau_so
```

```
ps1 = PhanSo(1, 2)
ps2 = PhanSo(3, 4)

print("ps1 + ps2 =", ps1 + ps2)
print("ps1 * ps2 =", ps1 * ps2)
print("ps1 == ps2:", ps1 == ps2)
```

BÀI TẬP 5.8

Viết chương trình Python thực hiện các yêu cầu sau:

Cho hai số phức dạng:

$$SP1 = a1 + i * b1;\ SP2 = a2 + i * b2;$$

Phép cộng, trừ hai số phức được định nghĩa như sau:

$$SP3 = SP1 + SP2 = (a1 + a2) + i * (b1 + b2);$$

$$SP3 = SP1 - SP2 = (a1 - a2) + i * (b1 - b2);$$

Hãy xây dựng lớp số phức với các thuộc tính Thực, ảo và các phương thức:

- Phương thức __str__: in số phức
- Phương thức toán tử ** để thực hiện cộng hai số phức
- Phương thức toán tử // để thực hiện trừ hai số phức.

Xây dựng chương trình chính để sử dụng lớp Số phức nói trên.

BÀI TẬP 5.7

```
class SoPhuc:
    def __init__(self, thuc, ao):
        self.thuc = thuc
        self.ao = ao

    def __str__(self):
        return f"{self.thuc} + {self.ao}i" if self.ao >= 0 else f"{self.thuc} - {-self.ao}i"

    def __pow__(self, other):
        return SoPhuc(self.thuc + other.thuc, self.ao + other.ao)

    def __floordiv__(self, other):
        return SoPhuc(self.thuc - other.thuc, self.ao - other.ao)
```

```
if __name__ == "__main__":
    sp1 = SoPhuc(1, 2)  # Số phức SP1 = 1 + 2i
    sp2 = SoPhuc(3, 4)  # Số phức SP2 = 3 + 4i

    # Phép cộng
    sp3 = sp1 ** sp2
    print("SP1 ** SP2 =", sp3)

    # Phép trừ
    sp4 = sp1 // sp2
    print("SP1 // SP2 =", sp4)
```

KẾT THÚC