Chương 8. Lớp (Class)

Nội dung

Giới thiệu

Tạo và sử dụng lớp

Làm việc với lớp và thể hiện của lớp

Kế thừa

Sử dụng các lớp

Thư viện chuẩn của Python

Giới thiệu

Lập trình hướng đối tượng là một trong những cách hiệu quả nhất khi viết phần mềm. Trong lập trình hướng đối tượng, ta viết ra những lớp (classes) trong lập trình hướng đối tượng, ta viết các lớp đại diện cho những thứ trong thế giới thực và các tình huống, đồng thời ta tạo các đối tượng dựa trên những các lớp đó.

Khi viết một lớp, ta xác định hành vi chung mà tất cả các đối tượng có thể có.

Khi ta tạo các đối tượng riêng lẻ từ lớp, mỗi đối tượng sẽ tự động được trang bị hành vi chung; sau đó ta có thể cung cấp cho từng đối tượng bất cứ đặc điểm riêng biệt nào mà ta mong muốn.

Tạo một đối tượng từ một lớp được gọi là khởi tạo và ta làm việc với các thể hiện (instances) của một lớp.

Giới thiệu

Nội dung trong chương tập trung vào:

viết các lớp và tạo các thể hiện của các lớp đó

chỉ định loại thông tin có thể được lưu trữ trong các thể hiện và sẽ xác định các hành động có thể được thực hiện với các thể hiện này

viết các lớp kế thừa chức năng của các lớp hiện có, vì vậy các lớp tương tự có thể chia sẻ mã một cách hiệu quả

lưu trữ các lớp trong các mô-đun và import các lớp do các lập trình viên khác viết vào các tệp chương trình.

8.1. Tạo và sử dụng lớp

Chúng ta có thể mô hình hóa hầu hết mọi thứ bằng cách sử dụng các lớp. Hãy bắt đầu bằng cách viết một lớp, Dog, đại diện cho một con chó — không phải một con chó cụ thể, mà là bất kỳ con chó nào.

Tất cả có tên tuổi.

Chúng ta cũng biết rằng hầu hết các con chó đều ngồi và cuộn lại. Hai phần thông tin (tên và tuổi) và hai hành vi đó (ngồi và cuộn lại) sẽ có trong lớp Dog của vì chúng phổ biến với hầu hết các loài chó. Lớp này sẽ nói Python cách để tạo một đối tượng đại diện cho một con chó.

Tạo ra lớp Dog

Mỗi cá thể được tạo từ lớp Dog sẽ lưu trữ tên và tuổi, và chúng ta sẽ cung cấp cho mỗi chú chó khả năng sit() và roll_over():

```
①  class Dog:
②      """A simple attempt to model a dog."""
③      def __init__(self, name, age):
           """Initialize name and age attributes."""
④          self.name = name
           self.age = age
⑤      def sit(self):
           """Simulate a dog sitting in response to a command."""
           print(f"{self.name} is now sitting.")
       def roll_over(self):
           """Simulate rolling over in response to a command."""
           print(f"{self.name} rolled over!")
```

Phương thức __init__()

Phương thức __init __() tại ③ là một phương thức đặc biệt và Python chạy tự động bất cứ khi nào chúng ta tạo một thể hiện mới dựa trên trên lớp Dog. Phương thức này có hai dấu gạch dưới ở đầu và hai dấu gạch dưới ở cuối, một quy ước giúp ngăn chặn phương thức mặc định của Python trùng tên với phương thức khác của lập trình viên.

Đảm bảo sử dụng **hai gạch dưới** ở mỗi bên của __init __(). Nếu ta chỉ sử dụng **một** ở mỗi bên, phương thức sẽ không được gọi tự động khi ta sử dụng lớp, điều này có thể dẫn đến lỗi khó xác định

Chúng ta định nghĩa phương thức __init __() để có ba tham số: self, name, và age. Tham số self là ***bắt buộc*** trong quá trình xác định phương thức và nó phải đến đầu tiên trước các tham số khác. Nó phải được bao gồm trong định nghĩa vì khi Python gọi phương thức này sau đó (để tạo một phiên bản của Dog), lệnh gọi phương thức sẽ tự động chuyển đối số tự động.

Phương thức __init__()

Hai biến được định nghĩa tại ④ có tiền tố self. Bất kỳ biến tiền tố là self khả dụng cho mọi phương thức trong lớp và chúng ta cũng có thể truy cập các biến này thông qua bất kỳ thể hiện nào được tạo từ lớp.

Dòng self.name = name nhận giá trị được liên kết với tên tham số và gán nó cho tên biến, sau đó được gắn vào thể hiện đang được tạo. Quá trình tương tự cũng xảy ra với self.age = age. Các biến mà có thể truy cập thông qua các thể hiện như thế này được gọi là thuộc tính.

Lớp Dog có hai phương thức khác được định nghĩa: sit() và roll_over() ⑤. Vì các phương thức này không cần thêm thông tin để chạy, chúng ta chỉ xác định chúng có một tham số là self.

Các thể hiện chúng ta tạo sau này sẽ có quyền truy cập vào các phương thức này. Nói cách khác, các chú chó cụ thể có thể ngồi và lăn lộn.

Tạo một thể hiện của lớp (instance)

Tạo ra thể hiện cho một chú chó cụ thể

```
    class Dog:
    --snip--
①   my_dog = Dog('Willie', 6)
②   print(f"My dog's name is {my_dog.name}.")
③   print(f"My dog is {my_dog.age} years old.")
```

Lớp Dog mà chúng ta đang sử dụng ở đây là lớp ta vừa viết trong phần trước.
Tại ①, chúng ta yêu cầu Python tạo ra một con chó có tên là 'Willie' và có tuổi là 6. Khi Python đọc dòng này, nó gọi phương thức __init __() trong Dog với các đối số 'Willie' và 6.

Phương thức __init __() tạo ra một thể hiện cho con chó cụ thể này và đặt các thuộc tính tên và tuổi sử dụng các giá trị mà chúng ta đã cung cấp. Python sau đó trả về một thể hiện đại diện cho con chó này.

Truy cập các thuộc tính

Để truy cập các thuộc tính của một thể hiện ta sử dụng ký hiệu dấu chấm. Chúng ta truy cập giá trị của tên thuộc tính của my_dog bằng cách viết:

```
my_dog.name
```

Ký hiệu dấu chấm thường được sử dụng trong Python. Cú pháp này trình bày cách Python tìm giá trị của thuộc tính. Ở đây Python xem xét thể hiện my_dog và sau đó tìm tên thuộc tính được liên kết với my_dog.

Kết quả:

```
My dog's name is Willie.
My dog is 6 years old.
```

Gọi phương thức

```
class Dog:
--snip--
my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

Để gọi một phương thức, ta cung cấp tên của thể hiện (trong trường hợp này là my_dog) và phương thức ta muốn gọi, được phân tách bằng dấu chấm.

Khi Python đọc my_dog.sit(), nó tìm kiếm phương thức sit() trong lớp Dog và chạy mã. Python diễn giải dòng my_dog.roll_over() theo cách tương tự.

```
Willie is now sitting.
Willie rolled over!
```

Tạo ra nhiều thể hiện

```
class Dog:
--snip--
my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)
print(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()
print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

8.2. Làm việc với lớp và thể hiện của lớp

Chúng ta có thể sử dụng các lớp để biểu diễn nhiều tình huống trong thế giới thực. Một khi ta viết một lớp, ta sẽ dành phần lớn thời gian của mình để làm việc với các thể hiện được tạo từ lớp đó.

Một trong những nhiệm vụ đầu tiên ta sẽ muốn làm là sửa đổi các thuộc tính được liên kết với một thể hiện cụ thể. Ta có thể sửa đổi các thuộc tính của một thể hiện một cách trực tiếp hoặc viết các phương thức cập nhật các thuộc tính theo những cách riêng.

Lớp Car

```
class Car:
"""A simple attempt to represent a car."""
①       def __init__(self, make, model, year):
            """Initialize attributes to describe a car."""
            self.make = make
            self.model = model
            self.year = year
②       def get_descriptive_name(self):
            """Return a neatly formatted descriptive name."""
            long_name = f"{self.year} {self. make} {self.model}"
            return long_name.title()
③   my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
```

Lớp Car

Tại ②, chúng ta xác định một phương thức có tên get_descriptive_name() để gộp năm sản xuất, chế tạo và mô hình thành một chuỗi mô tả chiếc xe một cách gọn gàng. Điều này sẽ giúp ta không phải in giá trị của từng thuộc tính riêng lẻ. Để làm việc với các giá trị thuộc tính trong phương thức này, chúng ta sử dụng self.make, self.model và self.year.

Tại ③, chúng ta tạo một thể hiện từ lớp Car và gán nó cho biến my_new_car. Sau đó, chúng ta gọi phương thức get_descriptive_name() để hiển thị loại ô tô chúng ta có:

```
2019 Audi A4
```

Thiết lập giá trị mặc định cho thuộc tính

Thêm một thuộc tính gọi là odometer_reading luôn bắt đầu bằng giá trị bằng 0. Chúng ta cũng sẽ thêm phương thức read_odometer() để giúp chúng ta đọc đồng hồ đo đường của ô tô

Python tạo một thuộc tính mới được gọi là odometer_reading và đặt giá trị ban đầu của nó là 0 ① . Chúng ta cũng có một phương thức mới có tên read_odometer() tại ② giúp dễ dàng đọc số dặm mà ô tô đã đi được.

```
2019 Audi A4
This car has 0 miles on it.
```

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
①       self.odometer_reading = 0
    def get_descriptive_name(self):
        --snip--
②   def read_odometer(self):
        """Print a statement showing the car's mileage
        print(f"This car has {self.odometer_reading} m
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Thay đổi giá trị thuộc tính

Thay đổi giá trị của thuộc tính theo ba cách:

- thay đổi giá trị trực tiếp thông qua một thể hiện,
- đặt giá trị thông qua một phương thức hoặc
- gia số giá trị (thêm một số lượng nhất định vào nó) thông qua một phương thức

Thay đổi giá trị thuộc tính trực tiếp

Đặt số đọc đồng hồ đo đường thành 23 một cách trực tiếp:

```
class Car:
    --snip--
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Thay đổi giá trị thuộc tính thông qua phương thức

Thay vì truy cập trực tiếp thuộc tính, ta chuyển giá trị mới cho phương thức xử lý cập nhật giá trị nội bộ

```
class Car:
    --snip--
①   def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Thay đổi giá trị thuộc tính thông qua phương thức (t)

Chúng ta thể mở rộng phương thức update_odometer() để thực hiện công việc bổ sung mỗi khi số đọc đồng hồ đo đường được thay đổi. Ta thêm một chút logic vào đảm bảo rằng không ai cố gắng quay ngược lại số đọc của đồng hồ đo đường đi:

```
class Car:
    --snip--
    def update_odometer(self, mileage):
        """      Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.       """
①       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
②           print("You can't roll back an odometer!")
```

Tăng giá trị của thuộc tính thông qua phương thức

Giả sử chúng ta mua một chiếc ô tô đã qua sử dụng và tăng giá trị đi được lên 100 dặm giữa khoảng thời gian mua nó với thời gian đăng ký nó.

Đây là một phương thức cho phép chúng ta chuyển số gia tăng này và thêm giá trị đó cho số đọc đồng hồ đo đường đi:

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

```
class Car:
    --snip--
    def update_odometer(self, mileage):
        --snip--
①   def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles
my_used_car = Car('subaru', 'outback', 2015)
print(my_used_car.get_descriptive_name())
my_used_car.update_odometer(23_500)
my_used_car.read_odometer()
my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

8.3. Kế thừa

Không phải lúc nào chúng ta cũng phải bắt đầu lại từ đầu khi viết một lớp. Nếu lớp ta đang viết là một phiên bản chuyên biệt của một lớp khác mà đã được viết, ta có thể sử dụng kế thừa.

Khi một lớp kế thừa từ lớp khác, nó sẽ sử dụng các thuộc tính và phương thức của lớp thứ nhất. Lớp ban đầu được gọi là lớp cha , và lớp mới là lớp con.

Lớp con có thể kế thừa bất kỳ hoặc tất cả các thuộc tính và phương thức của lớp cha của nó, nhưng cũng có thể tự do định nghĩa các thuộc tính và phương thức mới của riêng nó.

Phương thức __init __() cho lớp con

Khi ta đang viết một lớp mới dựa trên một lớp hiện có, ta sẽ thường muốn gọi phương thức __init __() từ lớp cha. Điều này sẽ khởi tạo bất kỳ thuộc tính nào đã được định nghĩa trong phương thức __init __() của lớp cha và làm các thuộc tính có sẵn trong lớp con.

Ví dụ, hãy mô hình hóa một chiếc ô tô điện. Ô tô điện chỉ là một loại ô tô cụ thể, vì vậy chúng ta có thể dựa xây dựng lớp ElectricCar dựa trên lớp Car đã được xây dựng trước đó. Sau đó, chúng ta sẽ chỉ phải viết mã cho các thuộc tính và hành vi cụ thể đối với ô tô điện.

Hãy bắt đầu bằng cách tạo một phiên bản đơn giản của lớp ElectricCar,thực hiện mọi thứ mà lớp Car làm:

Phương thức __init __() cho lớp con

```
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
```

```
①   class ElectricCar(Car):
        """Represent aspects of a car, specific to electric vehicles."""
②       def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
③           super().__init__(make, model, year)
④   my_tesla = ElectricCar('tesla', 'model s', 2019)
    print(my_tesla.get_descriptive_name())
```

Tại ①, chúng ta xác định lớp con, ElectricCar. Tên của lớp cha phải được bao gồm trong dấu ngoặc đơn trong định nghĩa của một lớp con. Phương thức __init __() tại ② lấy thông tin cần thiết để tạo một thể hiện của lớp Car.

Hàm super tại ③ là một chức năng đặc biệt cho phép gọi một phương thức từ lớp cha. Dòng này yêu cầu Python gọi __init __() từ Car, cung cấp cho một thể hiện ElectricCar tất cả các thuộc tính xác định trong phương thức đó. Tên super xuất phát từ quy ước gọi lớp cha là superclass và lớp con là subclass.

Phương thức __init __() cho lớp con

Khi phương thức __init __() được định nghĩa trong ElectricCar được gọi, đến lượt nó ra lệnh cho Python gọi phương thức __init __() được định nghĩa trong lớp cha Car. Chúng ta cung cấp các đối số 'tesla', 'model s' và 2019. Ngoài __init __(), không có thuộc tính hoặc phương thức nào đặc biệt của một chiếc ô tô điện. Tại thời điểm này, chúng ta chỉ đảm bảo ô tô điện có các hành vi tương ứng của ô tô:

```
2019 Tesla Model S
```

Thực thể ElectricCar hoạt động giống như một thực thể Car, vì vậy bây giờ chúng ta có thể bắt đầu xác định các thuộc tính và phương thức cụ thể cho ô tô điện.

Định nghĩa các thuộc tính và phương thức cho lớp con

Thêm một thuộc tính đặc biệt cho ô tô điện (ví dụ là pin) và một phương thức để thông báo về thuộc tính này

```
class Car:
    --snip--
class ElectricCar(Car):
    "    ""Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        super().__init__(make, model, year)
①       self.battery_size = 75
②   def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

Định nghĩa các thuộc tính và phương thức cho lớp con

```
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Không có giới hạn về số lượng ta có thể chuyên biệt hóa lớp ElectricCar. Ta có thể thêm nhiều thuộc tính và phương thức tùy ý để tạo mô hình một chiếc ô tô điện ở bất kỳ mức độ chính xác nào ta cần

Ghi đè phương thức từ lớp cha

Để ghi đè phương thức của lớp cha, ta định nghĩa một phương thức trong lớp con có cùng tên với phương thức muốn ghi đè của lớp cha. Python sẽ bỏ qua phương thức của lớp cha và chỉ chú ý đến phương thức ta định nghĩa trong lớp con

Giả sử lớp Car có một phương thức gọi là fill_gas_tank(). Phương thức này là vô nghĩa đối với một chiếc xe chạy hoàn toàn bằng điện, vì vậy ta có thể muốn ghi đè phương thức này.

```
class ElectricCar(Car):
    --snip--
    def fill_gas_tank(self):
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

Thuộc tính là một thể hiện lớp

Một phần của một lớp có thể được viết như một lớp riêng biệt. Ta có thể chia lớp lớn của mình thành các lớp nhỏ hơn làm việc cùng nhau.

```
class Battery:
    """A simple attempt to model a battery for an electric car."""
②   def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size
③       def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")
```

Thuộc tính là một thể hiện lớp

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
④  self.battery = Battery()

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Thuộc tính là một thể hiện lớp

Thêm một phương thức khác vào Battery báo cáo phạm vi hoạt động của ô tô dựa trên kích thước pin

```
class Car:
    --snip--
class Battery:
    --snip--
①   def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315
        print(f"This car can go about {range} miles on a full charge.")
class ElectricCar(Car):
--snip--
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

8.4. Sử dụng các lớp

Khi thêm nhiều chức năng hơn vào các lớp, các tệp lưu trữ mã nguồn có thể dài ra, ngay cả khi ta đã sử dụng kế thừa đúng cách.

Để phù hợp với triết lý tổng thể của Python, ta sẽ muốn giữ các tệp mã nguồn gọn gàng nhất có thể. Để trợ giúp, Python cho phép lưu trữ các lớp trong các mô-đun và sau đó nhập các lớp cần thiết vào chương trình chính.

Nhập vào một lớp đơn

Kể từ thời điểm này, bất kỳ chương trình sử dụng mô-đun này sẽ cần một tên tệp cụ thể hơn, chẳng hạn như my_car.py. Dưới đây là car.py chỉ với mã từ Car class:

car.py

```
"""A class that can be used to represent a car."""
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 100
```

my_car.py

```
from car import Car
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

```
2019 Audi A4
This car has 23 miles on it.
```

Lưu trữ nhiều lớp trong một module

Có thể lưu trữ bao nhiêu lớp tùy thích trong một mô-đun duy nhất, mỗi lớp trong một mô-đun nên có liên quan với nhau bằng cách nào đó.

car.py

```
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
class Battery:
    """A model a battery for an electric car."""
    def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size
class ElectricCar(Car):
    """Models aspects of a car, specific to
electric vehicles."""
    def __init__(self, make, model, year):
```

my_electric_car.py

```
from car import ElectricCar
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

Nhập vào nhiều lớp trong một module

Ta có thể nhập bao nhiêu lớp tùy thích vào một chương trình. Nếu chúng ta muốn làm một chiếc ô tô thông thường và một chiếc ô tô điện trong cùng một tệp, chúng ta nhập cả hai lớp, Car và ElectricCar

Nhập hai hay nhiều lớp từ một module bằng cách phân tách các lớp với dấu phẩy.

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)

print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)

print(my_tesla.get_descriptive_name())
```

```
2019 Beetle Beetle
2019 Roadster Roadster
```

Nhập vào toàn bộ module

Có thể nhập toàn bộ mô-đun và sau đó truy cập các lớp ta cần sử dụng ký hiệu dấu chấm. Cách tiếp cận này đơn giản và tạo ra mã dễ đọc.

```
import car
my_beetle = car.Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

đầu tiên chúng ta nhập vào module car. Sau đó, để truy cập vào các lớp, ta sử dụng qua cú pháp: module_name.ClassName.

Nhập vào toàn bộ các lớp trong một module

Nhập mọi lớp từ một module bằng cú pháp sau:

```
from module_name import *
```

Phương pháp này không được khuyến khích vì hai lý do:

- Không rõ lớp nào ta đang sử dụng từ mô-đun.
- Nhầm lẫn với tên trong tệp.

Nhập vào một module từ một module

Chia các lớp của mình qua một số mô-đun để giữ cho bất kỳ cái nào không phát triển quá lớn và tránh lưu trữ các lớp không liên quan trong cùng một mô-đun. Khi lưu trữ các lớp của mình trong một số mô-đun, ta có thể thấy rằng một lớp trong một mô-đun phụ thuộc vào một lớp trong một mô-đun khác.

car.py

```
class Car:
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
```

electric_car.py

```
from car import Car
class Battery:
    --snip--
class ElectricCar(Car):
    --snip—
```

my_car.py

```
from car import Car
from electric_car import ElectricCar
my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())
my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

Sử dụng bí danh

Bí danh có thể khá hữu ích khi sử dụng mô-đun để sắp xếp mã dự án. Ta có thể sử dụng bí danh khi nhập lớp.

```
from electric_car import ElectricCar as EC

my_tesla = EC('tesla', 'roadster', 2019)
```

8.5. Thư viện chuẩn của Python

Thư viện chuẩn Python là một tập hợp các mô-đun đi kèm với mọi cài đặt Python.

Có thể sử dụng bất kỳ hàm hoặc lớp nào trong thư viện chuẩn bằng cách bao gồm một câu lệnh nhập đơn giản ở đầu tệp.

Hàm **randin()** nhận hai đối số là số nguyên và trả về một số nguyên được chọn ngẫu nhiên giữa

```
>>> from random import randint
>>> randint(1, 6)
3
```

Hàm **choise()** nhận vào một danh sách hoặc tuple và trả về một phần tử được chọn ngẫu nhiên:

```
>>> from random import choice
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
>>> first_up = choice(players)
>>> first_up
'florence'
```

Kết chương

Trong chương này, ta đã học cách viết các lớp theo mong muốn. Ta đã học cách lưu trữ thông tin trong một lớp bằng cách sử dụng các thuộc tính và cách viết các phương thức cung cấp cho các lớp và hành vi mà chúng cần. Ta đã học cách viết các phương thức __init __() để tạo các thể hiện từ các lớp với chính xác các thuộc tính mong muốn.

Chúng ta đã thấy cách sửa đổi các thuộc tính của một thể hiện trực tiếp và thông qua các phương thức. Ta đã học được rằng kế thừa có thể đơn giản hóa việc tạo các lớp có liên quan với nhau và đã học cách sử dụng các thể hiện của một lớp làm thuộc tính trong một lớp khác để giữ mỗi lớp đơn giản.

Ta đã thấy cách lưu trữ các lớp trong mô-đun và nhập các lớp nếu cần vào tệp nơi chúng sẽ được sử dụng có thể giúp các dự án được tổ chức quy củ. Ta bắt đầu tìm hiểu về thư viện chuẩn Python và đã thấy một ví dụ dựa trên mô-đun random.

Bài tập chương 8