Code Reuse

Inheritance (Optional)

Documenting Functions, Classes, and Methods (Tài liệu về Hàm, Lớp và Phương thức)

Trong quá trình viết mã, việc tạo tài liệu cho các hàm, lớp và phương thức là vô cùng quan trọng. Tài liệu giúp người đọc mã hiểu được mục đích, cách sử dụng và hàm ý của từng phần mã. Trong Python, tài liệu thường được thêm vào mã thông qua các chuỗi tài liệu (**docstring**).

1. Tài liệu cho Hàm (Function)

Khi tạo tài liệu cho một hàm, bạn cần mô tả rõ ràng về mục đích của hàm, đầu vào (input), đầu ra (output) và các ngoại lệ có thể xảy ra. Điều này giúp người sử dụng hàm hiểu cách hoạt động của nó.

```
def add(a, b):
    """This function adds two numbers and returns the result.
    Args:
        a (int): The first number to add.
        b (int): The second number to add.
    Returns:
        int: The sum of a and b.
    """
    return a + b
```

Đây là một hàm có tên add, nhận vào hai đối số a và b. Mục đích của hàm này là để cộng hai số lại với nhau và trả về kết quả. Hàm add này có docstring mô tả đầu vào, đầu ra và mục đích của nó.

2. Tài liệu cho Lớp (Class)

Tương tự, khi tạo tài liệu cho một lớp, bạn cần mô tả rõ về mục đích và chức năng của lớp, cũng như các thuộc tính và phương thức của lớp.

```
class Person:
    """A class to represent a person.
    Attributes:
        name (str): The name of the person.
        age (int): The age of the person.
    Methods:
        say_hello: Prints a greeting message.
    """
        def __init__(self, name, age):
        """The constructor for Person class.
        Args:
            name (str): The name of the person.
            age (int): The age of the person.
        """
        self.name = name
        self.age = age
    def say_hello(self):
        """Prints a greeting message."""
```

```
        print(f"Hello, my name is {self.name}.")
```

Đây là một lớp **Person** với hai thuộc tính là **name** và **age**, và một phương thức **say_hello**. Phương thức **__init__** là phương thức khởi tạo, được gọi khi một đối tượng của lớp được tạo. Phương thức **say_hello** dùng để in ra một thông điệp chào mừng.

3. Tài liệu cho Phương thức (Method)

Cách tạo tài liệu cho phương thức tương tự như hàm, vì phương thức cũng là một loại hàm nhưng nằm trong lớp.

```
class Person:
    # ...

    def say_hello(self):
        """Prints a greeting message."""
        print(f"Hello, my name is {self.name}.")
```

Đây là phương thức **say_hello** nằm trong lớp **Person**, dùng để in ra một thông điệp chào mừng.

Bên cạnh việc tạo tài liệu, việc hiểu và áp dụng **kế thừa (inheritance)** trong lập trình hướng đối tượng cũng rất quan trọng. Kế thừa cho phép bạn tạo ra các lớp mới dựa trên những lớp đã có sẵn, giúp tiết kiệm thời gian, công sức và giảm lỗi khi viết mã. Trong Python, chúng ta sử dụng dấu ngoặc đơn trong khai báo lớp để thể hiện mối quan hệ kế thừa.

```
class Fruit:
    """A class to represent a fruit.
    Attributes:
        color (str): The color of the fruit.
        flavor (str): The flavor of the fruit.
    """
    # ...
class Apple(Fruit):
    """A class to represent an apple, inherits from Fruit class."""
    # ...
```

Với kiến thức về việc tạo tài liệu và kế thừa, bạn có thể viết mã Python một cách hiệu quả và rõ ràng hơn.

Đây là một lớp **Fruit** với hai thuộc tính là **color** và **flavor**. Lớp **Apple** kế thừa từ lớp **Fruit**, điều này có nghĩa là **Apple** sẽ có tất cả các thuộc tính và phương thức của **Fruit** (trừ khi **Apple** ghi đè lên chúng).

Object Inheritance

Kế thừa trong Lập trình Hướng đối tượng

Trong lập trình hướng đối tượng, khái niệm kế thừa cho phép bạn xây dựng mối quan hệ giữa các đối tượng, nhóm các khái niệm tương tự lại với nhau và giảm bớt việc lặp lại code. Hãy tạo một lớp Fruit tự định nghĩa với các thuộc tính màu sắc và hương vị:

```
class Fruit:
    def __init__(self, color, flavor):
        self.color = color
        self.flavor = flavor
```

Chúng ta đã định nghĩa một lớp **Fruit** với hàm khởi tạo cho các thuộc tính màu sắc và hương vị. Tiếp theo, chúng ta sẽ định nghĩa một lớp **Apple** cùng với một lớp **Grape** mới, cả hai chúng ta đều muốn kế thừa các thuộc tính và hành vi từ lớp **Fruit**:

```
class Apple(Fruit):
    pass
class Grape(Fruit):
    pass
```

Trong Python, chúng ta sử dụng dấu ngoặc đơn trong khai báo lớp để cho phép lớp kế thừa từ lớp **Fruit**. Vì vậy, trong ví dụ này, chúng ta đang hướng dẫn máy tính của chúng ta rằng cả lớp **Apple** và lớp **Grape** đều kế thừa từ lớp **Fruit**. Điều này có nghĩa là cả hai đều có cùng phương thức khởi tạo, đặt các thuộc tính màu sắc và hương vị. Bây giờ, chúng ta có thể tạo các thể hiện của lớp **Apple** và **Grape**:

```
granny_smith = Apple("green", "tart")
carnelian = Grape("purple", "sweet")
print(granny_smith.flavor)  # Output: tart
print(carnelian.color)  # Output: purple
```

Kế thừa cho phép chúng ta định nghĩa các thuộc tính hoặc phương thức được chia sẻ bởi tất cả các loại trái cây mà không cần phải định nghĩa chúng trong mỗi lớp trái cây riêng lẻ. Chúng ta cũng có thể định nghĩa các thuộc tính hoặc phương thức cụ thể chỉ có liên quan đến một loại trái cây cụ thể.

Composition (Optional)

Tài liệu về Các hàm, lớp và phương thức

1. **Giới thiệu** Trong việc phát triển phần mềm, việc tạo tài liệu cho mã nguồn là một phần rất quan trọng. Điều này giúp cho những người khác dễ dàng hiểu được mã nguồn mà bạn đã viết. Trong Python, việc này đặc biệt quan trọng đối với các hàm, lớp và phương thức. Bạn có thể viết tài liệu cho một hàm, lớp hoặc phương thức bằng cách sử dụng chuỗi tài liệu (docstring).
2. **Chuỗi tài liệu (docstring)** Một docstring là một chuỗi văn bản nằm ở đầu một hàm, lớp, phương thức hoặc mô-đun Python để giải thích mục đích và cách sử dụng của nó. Điều này rất hữu ích cho những người khác khi họ muốn sử dụng hoặc mở rộng mã của bạn. Chuỗi tài liệu được đặt ngay sau dòng khai báo của hàm, lớp hoặc phương thức và trước bất kỳ mã nào khác.
3. **Ví dụ về chuỗi tài liệu**

Ví dụ về việc sử dụng chuỗi tài liệu trong một hàm:

```
def add(a, b):
    """This function adds two numbers and returns the result.
    Args:
        a (int): The first number to add.
        b (int): The second number to add.
    Returns:
        int: The sum of a and b.
    """
    return a + b
```

Đoạn mã này định nghĩa một hàm có tên là **add** nhận vào hai đối số **a** và **b**. Hàm này sẽ trả về tổng của hai số này. Docstring của hàm giải thích rằng hàm này sẽ cộng hai số và trả về kết quả. Đồng thời, nó cũng mô tả về các đối số đầu vào và kiểu dữ liệu trả về.

4. **Sử dụng chuỗi tài liệu** Bạn có thể truy cập vào chuỗi tài liệu của một hàm, lớp hoặc phương thức bằng cách sử dụng thuộc tính __**doc**__. Ví dụ:

```
print(add.__doc__)
```

Đoạn mã này in ra chuỗi tài liệu (docstring) của hàm **add** thông qua việc sử dụng thuộc tính __**doc**__. Kết quả in ra sẽ là docstring của hàm **add** mà ta đã định nghĩa ở trên.

5. **Thành phần kết hợp (Composition)** Trong lập trình hướng đối tượng, kế thừa là một phần quan trọng. Tuy nhiên, không phải lúc nào mối quan hệ giữa các lớp cũng là kế thừa. Có thể một lớp A không phải là lớp con của lớp B nhưng lại chứa các đối tượng của lớp B. Đây chính là thành phần kết hợp (Composition).
6. **Ví dụ về Composition** Ví dụ, chúng ta có lớp **Package** biểu diễn một gói phần mềm có thể được cài đặt trên mỗi máy trong mạng lưới của chúng ta. Lớp này chứa nhiều thông tin về phần mềm, như tên, phiên bản, kích thước, và nhiều hơn nữa. Chúng ta cũng có một lớp **Repository** biểu diễn tất cả các gói mà chúng ta có sẵn để cài đặt nội bộ. Trong trường hợp này, **Repository** không phải là một **Package** và **Package** cũng không phải là một **Repository**. Thay vào đó, **Repository** chứa các **Package**. Để mô hình hóa điều này trong mã của chúng ta, lớp **Repository** sẽ có một thuộc tính có thể là một danh sách hoặc một từ điển, chứa các thể hiện của lớp **Package**.

```
class Repository:
    def __init__(self):
        self.packages = {}

    def add_package(self, package):
        self.packages[package.name] = package

    def total_size(self):
        result = 0
        for package in self.packages.values():
            result += package.size
        return result
```

Đoạn mã này định nghĩa một lớp **Repository** với một từ điển **packages** để lưu trữ các gói phần mềm. Các hàm trong lớp **Repository** bao gồm:

- __**init**__: Hàm khởi tạo của lớp, được gọi khi một đối tượng của lớp này được tạo ra. Hàm này khởi tạo **packages** là một từ điển rỗng.
- **add_package**: Hàm này nhận vào một đối tượng **package** và thêm đối tượng này vào từ điển **packages** với key là tên của package.

- **total_size**: Hàm này tính toán và trả về tổng kích thước của tất cả các gói trong từ điển **packages**.

Đoạn mã này mô phỏng quan hệ giữa lớp **Repository** và **Package** thông qua mô hình kết hợp (Composition). Trong quan hệ này, **Repository** không phải là **Package** và ngược lại, nhưng **Repository** chứa nhiều **Package**.

7. **Kết luận** Composition là một công cụ mạnh mẽ cho phép chúng ta tận dụng mã từ các lớp khác nhau. Thông qua việc sử dụng các thuộc tính và phương thức của các đối tượng khác như thuộc tính của lớp của chúng ta, chúng ta có thể tạo ra các lớp có chức năng phức tạp hơn mà không cần phải viết lại toàn bộ mã của các lớp khác.

Object Composition

Đối tượng (Object) Composition là một khái niệm trong lập trình hướng đối tượng, nó cho phép một lớp sử dụng mã nguồn từ một lớp khác. Tuy không có mối quan hệ thừa kế giữa hai lớp, nhưng chúng lại có mối quan hệ với nhau.
Hãy xem xét ví dụ sau: Giả sử chúng ta có một lớp **Package** đại diện cho một gói phần mềm. Lớp này chứa các thuộc tính của gói phần mềm như **name** (tên), **version** (phiên bản) và **size** (kích thước). Chúng ta cũng có một lớp khác là **Repository** đại diện cho tất cả các gói phần mềm có thể cài đặt. Dù không có mối quan hệ thừa kế giữa hai lớp này, nhưng chúng lại có mối liên hệ với nhau: lớp **Repository** sẽ chứa một danh sách hoặc từ điển các **Package** mà nó quản lý.
Dưới đây là định nghĩa của lớp **Repository**:

```
class Repository:
    def __init__(self):
        self.packages = {}

    def add_package(self, package):
        self.packages[package.name] = package

    def total_size(self):
        result = 0
        for package in self.packages.values():
            result += package.size
        return result
```

Trong hàm khởi tạo __**init**__, chúng ta khởi tạo từ điển **packages**, nó sẽ chứa các đối tượng **Package** có sẵn trong phiên bản **Repository** này. Chúng ta khởi tạo từ điển trong hàm khởi tạo để đảm bảo mỗi thể hiện của lớp **Repository** đều có từ điển riêng của nó.
Tiếp theo, chúng ta định nghĩa hàm **add_package**, nó nhận một đối tượng **Package** làm tham số, sau đó thêm nó vào từ điển của chúng ta, sử dụng thuộc tính **name** của package làm key.
Cuối cùng, chúng ta định nghĩa một hàm **total_size** để tính tổng kích thước của tất cả các gói chứa trong kho lưu trữ của chúng ta. Hàm này duyệt qua tất cả các giá trị trong từ điển **Repository** của chúng ta và cộng các thuộc tính **size** từ mỗi đối tượng **Package** chứa trong từ điển, cuối cùng trả về tổng kích thước.
Trong ví dụ này, chúng ta đang sử dụng các thuộc tính của **Package** trong lớp **Repository** của chúng ta. Chúng ta cũng đang gọi phương thức **values()** trên từ điển **packages**. Composition cho phép chúng ta sử dụng các đối tượng như là thuộc tính, cũng như truy cập tất cả thuộc tính và phương thức của chúng.

Python Modules (Optional)

Modules trong Python (Tùy chọn)

Đến giờ, chúng ta đã sử dụng các tính năng cốt lõi của ngôn ngữ Python. Như các lệnh cơ bản như if, for, while, hoặc việc định nghĩa các hàm hoặc lớp, là một phần của ngôn ngữ và sẵn sàng cho chúng ta sử dụng bất cứ khi nào chúng ta cần. Tương tự như vậy, các kiểu dữ liệu như integers, floats, strings, lists, và dictionaries cũng là một phần của ngôn ngữ Python cơ bản vì chúng được sử dụng thường xuyên. Tất nhiên, điều này không đủ để thực hiện các tác vụ phức tạp. Chúng ta sẽ cần nhiều công cụ bổ sung như khả năng gửi gói tin qua mạng, đọc tệp tin từ máy của chúng ta, xử lý hình ảnh, hoặc bất cứ điều gì bạn muốn làm để công việc của bạn hiệu quả hơn. Để tổ chức mã nguồn cho các tác vụ như vậy, Python cung cấp một trừu tượng gọi là module. Module có thể được sử dụng để tổ chức các hàm, lớp và dữ liệu khác lại với nhau theo cách có cấu trúc. Nội tại, các module được thiết lập thông qua các tệp riêng biệt chứa các lớp và hàm cần thiết. Python đã đi kèm với một loạt các module sẵn sàng để sử dụng. Tất cả các module này đều nằm trong một nhóm gọi là thư viện chuẩn của Python.

Hãy xem cách chúng ta có thể sử dụng một số trong số đó. Đầu tiên, chúng ta sẽ sử dụng từ khóa **import** để nhập module **random**. Module này rất hữu ích để tạo ra các số ngẫu nhiên hoặc thực hiện lựa chọn ngẫu nhiên.

Sau khi đã nhập module, hãy sử dụng một hàm do module này cung cấp gọi là **randint**.

Hàm này nhận hai tham số và tạo ra một số ngẫu nhiên nằm giữa hai tham số mà chúng ta truyền vào. Trong trường hợp này, chúng ta đang tạo ra một số ngẫu nhiên từ 1 đến 10. Như bạn thấy, hàm này trả về các số khác nhau mỗi khi nó được gọi. Khá thú vị, phải không?

Cú pháp được sử dụng để gọi một hàm được cung cấp bởi một module tương tự như cách gọi một phương thức được cung cấp bởi một lớp. Nó sử dụng dấu chấm để phân tách tên của module và hàm được cung cấp bởi module đó.

Hãy thử sử dụng một module khác, module **datetime**. Chúng ta sử dụng module này để xử lý ngày và giờ.

Bây giờ, hãy lấy ngày hiện tại.

Nếu bạn tự hỏi tại sao chúng ta có một **datetime** kép, đó là vì module **datetime** cung cấp một lớp **datetime**, và lớp **datetime** này cung cấp cho chúng ta một phương thức gọi là **now**. Phương thức **now** này tạo ra một thể hiện của lớp **datetime** cho thời gian hiện tại. Chúng ta có thể thao tác với thể hiện **datetime** này theo nhiều cách khác nhau. Hãy xem qua một vài ví dụ.

Khi chúng ta gọi **print** với một thể hiện của lớp **datetime**, chúng ta thấy ngày được in ra theo một định dạng cụ thể. Ở phía sau hậu trường, hàm **print** đang gọi phương thức **str** của lớp **datetime** làm định dạng nó theo cách mà chúng ta thấy ở đây. Chúng ta cũng có thể truy cập thể hiện thông qua các thuộc tính và phương thức của nó. Ví dụ, chúng ta có thể xem các phần riêng lẻ của ngày như năm, như thế này.

Module **datetime** cung cấp nhiều lớp hơn lớp **datetime**. Ví dụ, chúng ta có thể sử dụng lớp **timedelta** để tính toán một ngày trong tương lai hoặc trong quá khứ. Hãy thử nghiệm điều này. Trong trường hợp này, chúng ta đang tạo một thể hiện của lớp **timedelta** với giá trị là 28 ngày, sau đó chúng ta đang cộng nó với thể hiện của lớp **datetime** mà chúng ta đã có và in ra kết quả. Có rất nhiều thứ khác có sẵn trong các module **datetime** và **random**. Nếu bạn quan tâm và muốn tìm hiểu thêm, bạn có thể đọc toàn bộ tài liệu tham khảo, nó có sẵn trực tuyến và chúng tôi sẽ bao gồm một liên kết trong phần đọc tiếp theo. Đây chỉ là một cái nhìn qua loa về những gì bạn có thể làm với các module. Bạn cũng có thể phát triển các module của riêng bạn. Chúng tôi sẽ nói thêm về điều này trong một khóa học sau. Hiện tại, hãy chỉ tập trung vào việc sử dụng các module Python hiện có.

Augmenting Python with Modules

Mô-đun Python là những tệp riêng biệt chứa các lớp, hàm và dữ liệu khác cho phép chúng ta nhập và sử dụng các phương thức và lớp này trong mã của chính mình. Python đi kèm với rất nhiều mô-đun ngay từ khi cài đặt. Những mô-đun này được gọi là Thư viện Tiêu chuẩn Python (Python

Standard Library). Bạn có thể sử dụng những mô-đun này bằng cách sử dụng từ khóa import, theo sau là tên mô-đun. Ví dụ, chúng ta sẽ nhập mô-đun random, sau đó gọi hàm randint trong mô-đun này:

```
>>> import random
>>> random.randint(1,10)
8
>>> random.randint(1,10)
7
>>> random.randint(1,10)
1
```

Hàm này nhận hai tham số kiểu số nguyên và trả về một số nguyên ngẫu nhiên giữa các giá trị chúng ta truyền vào; trong trường hợp này, là 1 và 10. Bạn có thể nhận thấy rằng việc gọi các hàm trong một module rất giống với việc gọi các phương thức trong một lớp. Chúng ta cũng sử dụng ký hiệu dấu chấm ở đây, với một dấu chấm giữa tên module và tên hàm.
Hãy xem xét một module khác: datetime. Module này rất hữu ích khi làm việc với ngày và giờ.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> type(now)
<class 'datetime.datetime'>
>>> print(now)
2019-04-24 16:54:55.155199
```

Đầu tiên, chúng ta nhập module. Tiếp theo, chúng ta gọi phương thức now() thuộc lớp datetime nằm trong module datetime. Phương thức này tạo ra một thể hiện của lớp datetime cho ngày và giờ hiện tại. Thể hiện này có một số phương thức mà chúng ta có thể gọi:

```
>>> print(now)
2019-04-24 16:54:55.155199
>>> now.year
2019
>>> print(now + datetime.timedelta(days=28))
2019-05-22 16:54:55.155199
```

Khi chúng ta gọi hàm print với một thể hiện của lớp datetime, chúng ta nhận được ngày và giờ được in ra theo một định dạng cụ thể. Điều này là bởi vì lớp datetime đã định nghĩa phương thức **str** tạo ra chuỗi đã được định dạng mà chúng ta thấy ở đây. Chúng ta cũng có thể gọi trực tiếp các thuộc tính và phương thức của lớp, như now.year trả về thuộc tính year của thể hiện.
Cuối cùng, chúng ta có thể truy cập vào các lớp khác chứa trong module datetime, như lớp timedelta. Trong ví dụ này, chúng ta đang tạo ra một thể hiện của lớp timedelta với tham số là 28 ngày. Sau đó, chúng ta thêm đối tượng này vào thể hiện của lớp datetime từ trước đó và in kết quả. Điều này có tác dụng thêm 28 ngày vào đối tượng datetime ban đầu của chúng ta.

Supplemental Reading for Code Reuse (Optional)

Supplemental Reading for Code Reuse

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

Pypi is the Python repository and index of an impressive number of modules developed by Python programmers around the world.

C1M5L3_Code_Reuse_V2

Bài giảng này giảng về việc tái sử dụng mã thông qua sự kế thừa và tổ chức trong Python.
Đầu tiên, nó nhắc lại về kế thừa bằng cách định nghĩa một lớp **Animal** đơn giản:

```
class Animal:
    name = ""
    category = ""

    def __init__(self, name):
        self.name = name

    def set_category(self, category):
        self.category = category
```

Lớp **Animal** này chưa đủ để làm nhiều thứ. Do đó, bạn được yêu cầu định nghĩa một lớp **Turtle** kế thừa từ lớp **Animal** và đặt danh mục của nó.
Tiếp theo, bạn được yêu cầu tạo một lớp **Snake** khác cũng kế thừa từ lớp **Animal** và đặt danh mục của nó là "reptile".
Bây giờ, giả sử chúng ta có nhiều loại Động vật (như rùa và rắn) trong một Sở thú. Chúng ta có một lớp **Zoo** để tổ chức các loại Động vật khác nhau này.

```
class Zoo:
    def __init__(self):
        self.current_animals = {}

    def add_animal(self, animal):
        self.current_animals[animal.name] = animal.category

    def total_of_category(self, category):
        result = 0
        for animal in self.current_animals.values():
            if animal == category:
                result += 1
        return result
```

Lớp **Zoo** có hai phương thức chính, **add_animal()** để thêm các thể hiện của lớp **Animal** vào sở thú, và **total_of_category()** để đếm tổng số lượng của mỗi loại động vật trong một danh mục nhất định.
Sau đó, bạn tạo các thể hiện của lớp **Turtle** và **Snake**, thêm chúng vào sở thú và kiểm tra số lượng động vật thuộc danh mục "reptile".

```
turtle = Turtle("Turtle") #create an instance of the Turtle class
snake = Snake("Snake") #create an instance of the Snake class

zoo.add_animal(turtle)
zoo.add_animal(snake)

print(zoo.total_of_category("reptile")) #how many zoo animal types in the reptile category
```

Nếu bạn đã điền đúng, kết quả sẽ là 2, vì chúng ta đã thêm một rùa và một con rắn, cả hai đều thuộc danh mục "reptile".
Tóm lại, bài giảng này hướng dẫn cách tái sử dụng mã bằng cách sử dụng kế thừa để tạo các lớp con và cách tổ chức các thể hiện của các lớp con này trong một lớp khác.

TỔNG KẾT
Nội dung trên tóm tắt lại các khái niệm quan trọng trong lập trình hướng đối tượng mà bạn đã học. Cụ thể:

1. Trong một ngôn ngữ hướng đối tượng như Python, các khái niệm thực tế được đại diện bởi các lớp (classes).
2. Các thể hiện của các lớp thường được gọi là đối tượng (objects).
3. Đối tượng có các thuộc tính (attributes) dùng để lưu trữ thông tin về chúng, và chúng có thể thực hiện công việc bằng cách gọi phương thức (methods) của chúng.
4. Chúng ta có thể truy cập vào thuộc tính và phương thức sử dụng ký hiệu dấu chấm (dot notation).
5. Đối tượng có thể được tổ chức thông qua kế thừa (inheritance), và chúng có thể được chứa bên trong nhau thông qua tổ hợp (composition).

Trong kinh nghiệm của Marga, lập trình hướng đối tượng rất hữu ích khi mô hình hóa các khái niệm thực tế. Trong công việc của cô, Marga thường làm việc với các đối tượng đại diện cho người dùng và tài khoản của họ. Cô sử dụng chúng để nhóm nhiều thuộc tính khác nhau giúp cô chuyển mã trừu tượng thành tương tác cụ thể. Cô cũng sử dụng các đối tượng trong mã của mình để nhóm các chức năng dựa trên dữ liệu mà chúng tác động.
Phần "In Marga's Words: My Favorite Course" chia sẻ về kinh nghiệm của Marga khi viết khóa học. Cô thực sự thích viết tất cả các khóa học, đặc biệt là khóa học 1 vì nó là cơ bản của Python. Việc viết khóa học này khiến cô nhớ lại thời gian làm trợ giảng ở đại học, dạy lập trình Python cho sinh viên năm đầu. Cô rất thích nhìn thấy sự phát triển của sinh viên từ không biết gì về lập trình sau một học kỳ trở thành lập trình viên Python khá giỏi. Khi viết khóa học, cô đã lấy lại cảm giác này và nhớ lại tất cả những sinh viên đã chia sẻ hành trình này với cô.

Bài thực hành này nhằm mô phỏng việc tạo ra một số lớp để giả lập máy chủ nhận các kết nối từ bên ngoài và một bộ cân bằng tải đảm bảo có đủ máy chủ phục vụ những kết nối này.
1. Lớp Server: Bạn cần tạo một lớp Server để đại diện cho các máy chủ phục vụ các kết nối. Mỗi kết nối được đại diện bởi một id (có thể là địa chỉ IP của máy tính kết nối với máy chủ). Trong mô phỏng này, mỗi kết nối tạo ra một lượng tải ngẫu nhiên trên máy chủ, từ 1 đến 10. Bạn sẽ cần điền vào các phần thiếu trong phương thức add_connection và load của lớp Server.
2. Lớp LoadBalancing: Lớp này bắt đầu chỉ với một máy chủ sẵn sàng. Khi một kết nối được thêm vào, nó sẽ ngẫu nhiên chọn một máy chủ để phục vụ kết nối đó, và sau đó chuyển kết nối đến máy chủ. Lớp LoadBalancing cũng cần theo dõi các kết nối đang diễn ra để có thể đóng chúng. Bạn cần điền vào các phần thiếu để hoàn thành lớp này.
Bài tập yêu cầu bạn tạo các phương thức để thêm và xóa kết nối từ mỗi Server, tính toán tải trên mỗi Server, và quản lý tập hợp các Server trong một LoadBalancer. Ngoài ra, bạn cũng cần để cài đặt một cơ chế tự động thêm máy chủ mới vào LoadBalancer khi tải trung bình vượt quá 50%.
Dưới đây là một ví dụ về cách bạn có thể điền vào các phần còn thiếu trong lớp Server:

```
class Server:
    def __init__(self):
        self.connections = {}

    def add_connection(self, connection_id):
        connection_load = random.random()*10+1
        self.connections[connection_id] = connection_load

    def close_connection(self, connection_id):
        if connection_id in self.connections:
            del self.connections[connection_id]

    def load(self):
        total = sum(self.connections.values())
        return total
```

1. Lớp Server

- **__init__**: Phương thức khởi tạo này tạo một đối tượng Server mới với không có kết nối nào đang hoạt động. Kết nối sẽ được lưu trữ trong một từ điển, với ID kết nối làm khóa và tải của kết nối làm giá trị.
- **add_connection**: Phương thức này nhận một ID kết nối làm đối số và thêm kết nối đó vào từ điển **connections** của máy chủ. Tải của kết nối mới này được tính toán ngẫu nhiên từ 1 đến 10.
- **close_connection**: Phương thức này nhận một ID kết nối làm đối số và loại bỏ kết nối có ID đó khỏi từ điển **connections** của máy chủ, nếu nó tồn tại.
- **load**: Phương thức này tính toán tải hiện tại cho tất cả các kết nối trên máy chủ bằng cách cộng tải của mỗi kết nối.

Và sau đây là cách bạn có thể hoàn thành lớp LoadBalancing:

```
class LoadBalancing:
    def __init__(self):
        self.connections = {}
        self.servers = [Server()]

    def add_connection(self, connection_id):
```

```
        server = random.choice(self.servers)
        server.add_connection(connection_id)
        self.connections[connection_id] = server

    def close_connection(self, connection_id):
        if connection_id in self.connections:
            server = self.connections[connection_id]
            server.close_connection(connection_id)
            del self.connections[connection_id]

    def avg_load(self):
        total_load = sum([server.load() for server in self.servers])
        avg_load = total_load / len(self.servers)
        return avg_load

    def ensure_availability(self):
        if self.avg_load() > 50:
            self.servers.append(Server())
```

2. Lớp LoadBalancing

- **__init__**: Phương thức khởi tạo này tạo một đối tượng LoadBalancing mới với một máy chủ sẵn sàng. Kết nối và các máy chủ được lưu trữ trong từ điển và danh sách tương ứng.
- **add_connection**: Phương thức này nhận một ID kết nối làm đối số, ngẫu nhiên chọn một máy chủ và thêm kết nối vào máy chủ đó. Nó cũng thêm kết nối vào từ điển **connections** của LoadBalancing, với ID kết nối làm khóa và máy chủ được chọn làm giá trị.
- **close_connection**: Phương thức này nhận một ID kết nối làm đối số và đóng kết nối trên máy chủ tương ứng. Nó cũng loại bỏ kết nối khỏi từ điển **connections** của LoadBalancing.
- **avg_load**: Phương thức này tính toán tải trung bình của tất cả các máy chủ bằng cách cộng tải của mỗi máy chủ và chia cho số máy chủ.
- **ensure_availability**: Phương thức này kiểm tra nếu tải trung bình cao hơn 50%, nếu vậy, nó sẽ thêm một máy chủ mới vào danh sách **servers**.

Nhớ rằng đây chỉ là một cách tiếp cận cơ bản và có thể có nhiều cách để giải quyết bài tập này.