```
# Khối mã này cung cấp cách bạn có thể tải các hình ảnh có trong chương sách này.

from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

```
Downloaded thinkpython.py
Downloaded diagram.py
```

Lớp và phương thức

Python là một **ngôn ngữ hướng đối tượng** - nghĩa là nó cung cấp các tính năng hỗ trợ lập trình hướng đối tượng, có các đặc điểm định nghĩa sau:

- Hầu hết các phép tính được biểu diễn dưới dạng các phép toán trên các đối tượng.
- Các đối tượng thường đại diện cho các vật thể trong thế giới thực, và các phương thức thường tương ứng với cách thức tương tác của các vật thể trong thế giới thực.
- Chương trình bao gồm các định nghĩa lớp và phương thức.

Ví dụ, trong chương trước, chúng ta đã định nghĩa một lớp `Time` tương ứng với cách mọi người ghi lại thời gian trong ngày, và chúng ta đã định nghĩa các hàm tương ứng với các loại hoạt động mà mọi người thực hiện với thời gian.

Nhưng không có mối liên hệ rõ ràng giữa định nghĩa của lớp `Time` và các định nghĩa hàm theo sau. Chúng ta có thể làm cho mối liên hệ rõ ràng hơn bằng cách viết lại một hàm dưới dạng một phương thức, được định nghĩa bên trong định nghĩa lớp.

Định nghĩa phương thức

Trong chương trước, chúng ta đã định nghĩa một lớp tên là `Time` và viết một hàm tên là `print_time` hiển thị thời gian trong ngày.

```
class Time:
    """Biểu diễn thời gian của ngày."""

def print_time(time):
    s = f'{time.hour:02d}:{time.minute:02d}:{time.second:02d}'
    print(s)
```

Để biến `print_time` thành một phương thức, tất cả những gì chúng ta phải làm là di chuyển định nghĩa hàm vào bên trong định nghĩa lớp. Lưu ý sự thay đổi về thụt dòng.

Đồng thời, chúng ta sẽ đổi tên tham số từ `time` thành `self`. Thay đổi này không cần thiết, nhưng thông thường tham số đầu tiên của một phương thức được đặt tên là `self`.

```
class Time:
    """Biểu diễn thời gian của ngày."""

    def print_time(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        print(s)
```

Để gọi phương thức này, bạn phải truyền một đối tượng `Time` làm đối số. Đây là hàm chúng ta sẽ sử dụng để tạo một đối tượng `Time`.

```
def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

Và đây là một trường hợp `Time`.

```
start = make_time(9, 40, 0)
```

Bây giờ có hai cách để gọi `print_time`. Cách đầu tiên (và ít phổ biến hơn) là sử dụng cú pháp hàm.

```
Time.print_time(start)
```

```
09:40:00
```

Trong phiên bản này, `Time` là tên của lớp, `print_time` là tên của phương thức và `start` được truyền làm tham số. Cách thứ hai (và idiomatic hơn) là sử dụng cú pháp phương thức:

```
start.print_time()
```

```
09:40:00
```

Trong phiên bản này, `start` là đối tượng mà phương thức được gọi, được gọi là **người nhận**, dựa trên phép ẩn dụ rằng gọi một phương thức giống như gửi một tin nhắn đến một đối tượng.

Bất kể cú pháp, hành vi của phương thức là giống nhau. Người nhận được gán cho tham số đầu tiên, vì vậy bên trong phương thức, `self` tham chiếu đến cùng một đối tượng với `start`.

Một phương thức khác

Đây là hàm `time_to_int` từ chương trước.

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

Và đây là một phiên bản được viết lại dưới dạng một phương thức.

```
%%add_method_to Time

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
```

Dòng đầu tiên sử dụng lệnh đặc biệt `add_method_to`, thêm một phương thức vào một lớp đã được định nghĩa trước đó. Lệnh này hoạt động trong một sổ ghi chép Jupyter, nhưng nó không phải là một phần của Python, vì vậy nó sẽ không hoạt động trong các môi trường khác. Thông thường, tất cả các phương thức của một lớp đều nằm bên trong định nghĩa lớp, vì vậy chúng được định nghĩa cùng một lúc với lớp. Nhưng đối với cuốn sách này, sẽ hữu ích khi định nghĩa một phương thức tại một thời điểm.

Như trong ví dụ trước, định nghĩa phương thức được thụt vào và tên của tham số là `self`. Ngoại trừ điều đó, phương thức giống hệt với hàm. Đây là cách chúng ta gọi nó.

```
start.time_to_int()
```

```
34800
```

Thường nói rằng chúng ta "gọi" một hàm và "gọi" một phương thức, nhưng chúng có nghĩa giống nhau.

Phương thức tĩnh

Làm một ví dụ khác, hãy xem xét hàm `int_to_time`. Đây là phiên bản từ chương trước.

```
def int_to_time(seconds):
    minute, second = divmod(seconds, 60)
    hour, minute = divmod(minute, 60)
    return make_time(hour, minute, second)
```

Hàm này lấy `seconds` làm tham số và trả về một đối tượng `Time` mới. Nếu chúng ta chuyển đổi nó thành một phương thức của lớp `Time`, chúng ta phải gọi nó trên một đối tượng `Time`. Nhưng nếu chúng ta đang cố gắng tạo một đối tượng `Time` mới, chúng ta sẽ gọi nó trên cái gì?

Chúng ta có thể giải quyết vấn đề con gà và quả trứng này bằng cách sử dụng **phương thức tĩnh**, là một phương thức không yêu cầu một trường hợp của lớp được gọi. Đây là cách chúng ta viết lại hàm này dưới dạng một phương thức tĩnh.

```
%%add_method_to Time

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return make_time(hour, minute, second)
```

Bởi vì nó là một phương thức tĩnh, nó không có `self` làm tham số. Để gọi nó, chúng ta sử dụng `Time`, là đối tượng lớp.

```
start = Time.int_to_time(34800)
```

Kết quả là một đối tượng mới đại diện cho 9:40.

```
start.print_time()
```

```
09:40:00
```

Bây giờ chúng ta đã có `Time.from_seconds`, chúng ta có thể sử dụng nó để viết `add_time` dưới dạng một phương thức. Đây là hàm từ chương trước.

```
def add_time(time, hours, minutes, seconds):
    duration = make_time(hours, minutes, seconds)
    seconds = time_to_int(time) + time_to_int(duration)
    return int_to_time(seconds)
```

Và đây là một phiên bản được viết lại dưới dạng một phương thức.

```
%%add_method_to Time

    def add_time(self, hours, minutes, seconds):
        duration = make_time(hours, minutes, seconds)
        seconds = time_to_int(self) + time_to_int(duration)
        return Time.int_to_time(seconds)
```

`add_time` có `self` làm tham số vì nó không phải là một phương thức tĩnh. Nó là một phương thức thông thường - còn được gọi là **phương thức của đối tượng**. Để gọi nó, chúng ta cần một đối tượng `Time`.

```
end = start.add_time(1, 32, 0)
print_time(end)
```

```
11:12:00
```

So sánh các đối tượng Time

Làm một ví dụ nữa, hãy viết `is_after` dưới dạng một phương thức. Đây là hàm `is_after`, là một giải pháp cho một bài tập trong chương trước.

```
def is_after(t1, t2):
    return time_to_int(t1) > time_to_int(t2)
```

Và đây là nó dưới dạng một phương thức.

```
%%add_method_to Time

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Bởi vì chúng ta đang so sánh hai đối tượng và tham số đầu tiên là `self`, chúng ta sẽ gọi tham số thứ hai là `other`. Để sử dụng phương thức này, chúng ta phải gọi nó trên một đối tượng và truyền đối tượng khác làm đối số.

```
end.is_after(start)
```

```
True
```

Một điều hay về cú pháp này là nó gần như đọc giống như một câu hỏi, "`end` sau `start` không?"

Phương thức `__str__`

Khi bạn viết một phương thức, bạn có thể chọn hầu hết mọi tên bạn muốn. Tuy nhiên, một số tên có ý nghĩa đặc biệt. Ví dụ: nếu một đối tượng có một phương thức tên là `__str__`, Python sử dụng phương thức đó để chuyển đổi đối tượng thành một chuỗi. Ví dụ: đây là một phương thức `__str__` cho một đối tượng thời gian.

```
%%add_method_to Time

    def __str__(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        return s
```

Phương thức này tương tự như `print_time`, từ chương trước, ngoại trừ việc nó trả về chuỗi thay vì in nó.

Bạn có thể gọi phương thức này theo cách thông thường.

```
end.__str__()
```

```
'11:12:00'
```

Nhưng Python cũng có thể tự gọi nó. Nếu bạn sử dụng hàm tích hợp sẵn `str` để chuyển đổi một đối tượng `Time` thành một chuỗi, Python sẽ sử dụng phương thức `__str__` trong lớp `Time`.

```
str(end)
```

```
'11:12:00'
```

Và nó cũng làm tương tự nếu bạn in một đối tượng `Time`.

```
print(end)
```

```
11:12:00
```

Các phương thức như `__str__` được gọi là **phương thức đặc biệt**. Bạn có thể xác định chúng vì tên của chúng bắt đầu và kết thúc bằng hai dấu gạch dưới.

Phương thức init

Đặc biệt nhất trong số các phương thức đặc biệt là `__init__`, được gọi là như vậy vì nó khởi tạo các thuộc tính của một đối tượng mới. Một phương thức `__init__` cho lớp `Time` có thể trông như thế này:

```
%%add_method_to Time

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Bây giờ, khi chúng ta khởi tạo một đối tượng `Time`, Python gọi `__init__` và chuyển các đối số cùng với nó. Vì vậy, chúng ta có thể tạo một đối tượng và khởi tạo các thuộc tính cùng một lúc.

```
time = Time(9, 40, 0)
print(time)
```

```
09:40:00
```

Trong ví dụ này, các tham số là tùy chọn, vì vậy nếu bạn gọi `Time` mà không có đối số, bạn sẽ nhận được các giá trị mặc định.

```
time = Time()
print(time)
```

```
00:00:00
```

Nếu bạn cung cấp một đối số, nó sẽ ghi đè lên `hour`:

```
time = Time(9)
print(time)
```

```
09:00:00
```

Nếu bạn cung cấp hai đối số, chúng sẽ ghi đè lên `hour` và `minute`.

```
time = Time(9, 45)
print(time)
```

```
09:45:00
```

Và nếu bạn cung cấp ba đối số, chúng sẽ ghi đè lên tất cả ba giá trị mặc định.

Khi tôi viết một lớp mới, tôi hầu như luôn bắt đầu bằng cách viết `__init__`, giúp dễ dàng tạo các đối tượng hơn, và `__str__`, hữu ích cho việc gỡ lỗi.

Nạp chồng toán tử

Bằng cách định nghĩa các phương thức đặc biệt khác, bạn có thể chỉ định hành vi của các toán tử trên các kiểu do lập trình viên định nghĩa. Ví dụ: nếu bạn định nghĩa một phương thức có tên `__add__` cho lớp `Time`, bạn có thể sử dụng toán tử `+` trên các đối tượng Time.

Đây là một phương thức `__add__`.

`Giải thích thêm từ dịch giả`

Nạp chồng toán tử là khả năng của các lớp trong Python để định nghĩa lại cách các toán tử (như +, -, *, v.v.) hoạt động đối với các đối tượng của lớp đó.

```
%%add_method_to Time

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return Time.int_to_time(seconds)
```

Chúng ta có thể sử dụng nó như thế này.

```
duration = Time(1, 32)
end = start + duration
print(end)
```

```
11:12:00
```

Có rất nhiều điều xảy ra khi chúng ta chạy ba dòng mã này:

- Khi chúng ta khởi tạo một đối tượng `Time`, phương thức `__init__` được gọi.
- Khi chúng ta sử dụng toán tử `+` với một đối tượng `Time`, phương thức `__add__` của nó được gọi.
- Và khi chúng ta in một đối tượng `Time`, phương thức `__str__` của nó được gọi.

Thay đổi hành vi của một toán tử để nó hoạt động với các kiểu do lập trình viên định nghĩa được gọi là **nạp chồng toán tử**. Đối với mỗi toán tử, như `+`, có một phương thức đặc biệt tương ứng, như `__add__`.

Gỡ lỗi

Một đối tượng `Time` là hợp lệ nếu giá trị của `minute` và `second` nằm trong khoảng từ `0` đến `60` - bao gồm `0` nhưng không bao gồm `60` - và nếu `hour` là dương. Ngoài ra, `hour` và `minute` phải là giá trị nguyên, nhưng chúng ta có thể cho phép `second` có phần thập phân. Các yêu cầu như thế này được gọi là **bất biến** vì chúng luôn phải đúng. Nói cách khác, nếu chúng không đúng, có nghĩa là có điều gì đó đã sai.

Viết mã để kiểm tra các bất biến có thể giúp phát hiện lỗi và tìm nguyên nhân của chúng. Ví dụ: bạn có thể có một phương thức như `is_valid` nhận một đối tượng Time và trả về `False` nếu nó vi phạm một bất biến.

```
%%add_method_to Time

    def is_valid(self):
        if self.hour < 0 or self.minute < 0 or self.second < 0:
            return False
        if self.minute >= 60 or self.second >= 60:
            return False
        if not isinstance(self.hour, int):
            return False
        if not isinstance(self.minute, int):
            return False
        return True
```

Sau đó, ở đầu mỗi phương thức, bạn có thể kiểm tra các đối số để đảm bảo chúng hợp lệ.

```
%%add_method_to Time

    def is_after(self, other):
        assert self.is_valid(), 'self không phải là một Time hợp lệ'
        assert other.is_valid(), 'self không phải là một Time hợp lệ'
        return self.time_to_int() > other.time_to_int()
```

Câu lệnh `assert` đánh giá biểu thức theo sau. Nếu kết quả là `True`, nó không làm gì cả; nếu kết quả là `False`, nó gây ra một `AssertionError`. Đây là một ví dụ.

```
duration = Time(minute=132)
print(duration)
```

```
00:132:00
```

```
%%expect AssertionError

start.is_after(duration)
```

```
AssertionError: self is not a valid Time
```

Các câu lệnh `assert` rất hữu ích vì chúng phân biệt mã xử lý các điều kiện bình thường với mã kiểm tra lỗi.

Thuật ngữ

object-oriented language - ngôn ngữ hướng đối tượng: Một ngôn ngữ cung cấp các tính năng để hỗ trợ lập trình hướng đối tượng, đáng chú ý là các kiểu do người dùng định nghĩa.

method - phương thức: Một hàm được định nghĩa bên trong định nghĩa lớp và được gọi trên các trường hợp của lớp đó.

receiver - đối tượng nhận: Đối tượng mà một phương thức được gọi trên đó.

static method - phương thức tĩnh: Một phương thức có thể được gọi mà không cần đối tượng làm đối tượng nhận.

instance method - phương thức của đối tượng: Một phương thức phải được gọi với một đối tượng làm đối tượng nhận.

special method - phương thức đặc biệt: Một phương thức thay đổi cách các toán tử và một số hàm hoạt động với một đối tượng.

operator overloading - nạp chồng toán tử: Quá trình sử dụng các phương thức đặc biệt để thay đổi cách các toán tử hoạt động với các kiểu do người dùng định nghĩa.

invariant - bất biến: Một điều kiện luôn đúng trong quá trình thực thi chương trình.

Bài tập

```
# Ô này yêu cầu Jupyter cung cấp thông tin gỡ lỗi chi tiết
# khi xảy ra lỗi thời gian chạy. Chạy nó trước khi làm các bài tập.

%xmode Verbose
```

Hỏi một trợ lý ảo

Để biết thêm thông tin về các phương thức tĩnh, hãy hỏi trợ lý ảo:

- "Sự khác biệt giữa phương thức của đối tượng và phương thức tĩnh là gì?"
- "Tại sao các phương thức tĩnh được gọi là tĩnh?"

Nếu bạn yêu cầu trợ lý ảo tạo một phương thức tĩnh, kết quả có thể bắt đầu bằng `@staticmethod`, là một "decorator" cho biết đó là một phương thức tĩnh. Decorators không được đề cập trong cuốn sách này, nhưng nếu bạn tò mò, bạn có thể hỏi trợ lý ảo để biết thêm thông tin.

Trong chương này, chúng tôi đã viết lại một số hàm dưới dạng phương thức. Trợ lý ảo thường giỏi trong loại chuyển đổi mã này. Ví dụ: dán hàm sau vào trợ lý ảo và hỏi nó, "Viết lại hàm này dưới dạng một phương thức của lớp `Time`."

Giải thích của dịch giả

Decorator là một chức năng trong Python cho phép thay đổi hoặc mở rộng hành vi của một hàm, phương thức, hoặc lớp mà không cần phải thay đổi mã nguồn của chúng. Decorator thường được sử dụng để thêm các tính năng hoặc chức năng bổ sung cho hàm hoặc phương thức mà không cần thay đổi trực tiếp chúng.

```
def subtract_time(t1, t2):
    return time_to_int(t1) - time_to_int(t2)
```

Bài tập

Trong chương trước, một loạt bài tập yêu cầu bạn viết một lớp `Date` và một số hàm làm việc với các đối tượng `Date`. Bây giờ hãy thực hành viết lại các hàm đó dưới dạng phương thức.

1. Viết một định nghĩa cho một lớp `Date` đại diện cho một ngày - tức là năm, tháng và ngày trong tháng.
2. Viết một phương thức `__init__` nhận `year`, `month` và `day` làm tham số và gán các tham số vào thuộc tính. Tạo một đối tượng đại diện cho ngày 22 tháng 6 năm 1933.
3. Viết phương thức `__str__` sử dụng f-string để định dạng các thuộc tính và trả về kết quả. Nếu bạn kiểm tra nó với `Date` mà bạn đã tạo, kết quả sẽ là `1933-06-22`.
4. Viết một phương thức có tên `is_after` nhận hai đối tượng `Date` và trả về `True` nếu cái đầu tiên đến sau cái thứ hai. Tạo một đối tượng thứ hai đại diện cho ngày 17 tháng 9 năm 1933 và kiểm tra xem nó có đến sau đối tượng đầu tiên hay không.

Gợi ý: Bạn có thể thấy hữu ích khi viết một phương thức có tên `to_tuple` trả về một tuple chứa các thuộc tính của một đối tượng `Date` theo thứ tự năm-tháng-ngày.

Bạn có thể sử dụng các ví dụ này để kiểm tra giải pháp của mình.

```
birthday1 = Date(1933, 6, 22)
print(birthday1)


birthday2 = Date(1933, 9, 17)
print(birthday2)


birthday1.is_after(birthday2)  # nên là False


birthday2.is_after(birthday1)  # nên là True
```