```
# 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 hàm

Tại thời điểm này, bạn đã biết cách sử dụng hàm để tổ chức mã và cách sử dụng các kiểu dữ liệu tích hợp để tổ chức dữ liệu. Bước tiếp theo là **lập trình hướng đối tượng (OOP)**, sử dụng các kiểu do lập trình viên định nghĩa để tổ chức cả mã và dữ liệu.

Lập trình hướng đối tượng là một chủ đề lớn, vì vậy chúng ta sẽ tiến hành từ từ. Trong chương này, chúng ta sẽ bắt đầu với mã không phải là chuẩn Pythonic - nghĩa là nó không phải là kiểu mã mà các lập trình viên có kinh nghiệm viết - nhưng nó là một điểm khởi đầu tốt. Trong hai chương tiếp theo, chúng ta sẽ sử dụng các tính năng bổ sung để viết mã theo phong cách chuẩn Python hơn.

Giải thích thêm của dịch giả

Thuật ngữ "idiomatic" thường mô tả cách viết mã theo phong cách tự nhiên, chuẩn mực, và phù hợp với ngôn ngữ lập trình Python. Thông thường, nó được gọi là chuẩn Pythonic hoặc theo phong cách chuẩn Python.

Các kiểu do lập trình viên định nghĩa

Chúng ta đã sử dụng nhiều kiểu tích hợp của Python - bây giờ chúng ta sẽ định nghĩa một kiểu mới. Làm ví dụ đầu tiên, chúng ta sẽ tạo một kiểu gọi là `Time` đại diện cho một thời điểm trong ngày. Một kiểu do lập trình viên định nghĩa còn được gọi là **lớp**.

Định nghĩa lớp trông như thế này:

```
class Time:
    """Thể hiện một thời điểm trong ngày."""
```

Tiêu đề cho biết lớp mới được gọi là `Time`. Thân là một docstring giải thích mục đích của lớp. Định nghĩa một lớp tạo ra một **đối tượng lớp**.

Đối tượng lớp giống như một nhà máy để tạo ra các đối tượng. Để tạo một đối tượng `Time`, bạn gọi `Time` như thể nó là một hàm.

```
lunch = Time()
```

Kết quả là một đối tượng mới có kiểu là `__main__.Time`, trong đó `__main__` là tên của mô-đun nơi `Time` được định nghĩa.

```
type(lunch)
```

```
__main__.Time
```

Khi bạn in một đối tượng, Python cho bạn biết nó thuộc loại gì và nó được lưu trữ ở đâu trong bộ nhớ (tiền tố `0x` có nghĩa là số tiếp theo ở dạng thập lục phân).

```
print(lunch)
```

```
<__main__.Time object at 0x7a80f5c42e10>
```

Tạo một đối tượng mới được gọi là **khởi tạo**, và đối tượng là một **trường hợp** của lớp.

Thuộc tính

Một đối tượng có thể chứa các biến, được gọi là **thuộc tính** và được phát âm với trọng âm ở âm tiết đầu tiên, giống như "AT-trib-ute", chứ không phải âm tiết thứ hai, giống như "a-TRIB-ute". Chúng ta có thể tạo các thuộc tính bằng cách sử dụng ký hiệu dấu chấm.

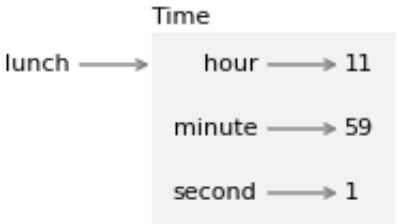
```
lunch.hour = 11
lunch.minute = 59
lunch.second = 1
```

Ví dụ này tạo ra các thuộc tính có tên là `hour`, `minute` và `second`, chứa giờ, phút và giây của thời gian `11:59:01`, đây là giờ ăn trưa theo quan điểm của tôi.

Sơ đồ sau đây cho thấy trạng thái của `lunch` và các thuộc tính của nó sau các phép gán này.

```
from diagram import make_frame, make_binding

d1 = dict(hour=11, minute=59, second=1)
frame = make_frame(d1, name='Time', dy=-0.3, offsetx=0.48)
binding = make_binding('lunch', frame)
```

```
from diagram import diagram, adjust

width, height, x, y = [1.77, 1.24, 0.25, 0.86]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```



Biến `lunch` tham chiếu đến một đối tượng `Time`, chứa ba thuộc tính. Mỗi thuộc tính tham chiếu đến một số nguyên. Một sơ đồ trạng thái như thế này - hiển thị một đối tượng và các thuộc tính của nó - được gọi là **sơ đồ đối tượng**.

Bạn có thể đọc giá trị của một thuộc tính bằng cách sử dụng toán tử dấu chấm.

```
lunch.hour
```

```
11
```

Bạn có thể sử dụng một thuộc tính như một phần của bất kỳ biểu thức nào.

```
total_minutes = lunch.hour * 60 + lunch.minute
total_minutes
```

```
719
```

Và bạn có thể sử dụng toán tử dấu chấm trong một biểu thức trong một f-string.

```
f'{lunch.hour}:{lunch.minute}:{lunch.second}'
```

```
'11:59:1'
```

Nhưng hãy lưu ý rằng ví dụ trước không ở định dạng chuẩn. Để sửa nó, chúng ta phải in các thuộc tính `minute` và `second` với một số 0 hàng đầu. Chúng ta có thể làm điều đó bằng cách mở rộng các biểu thức trong dấu ngoặc nhọn với một **bộ định dạng**. Trong ví dụ sau, các bộ định dạng chỉ ra rằng `minute` và `second` nên được hiển thị với ít nhất hai chữ số và một số 0 hàng đầu nếu cần.

```
f'{lunch.hour}:{lunch.minute:02d}:{lunch.second:02d}'
```

```
'11:59:01'
```

Chúng ta sẽ sử dụng f-string này để viết một hàm hiển thị giá trị của một đối tượng `Time`. Bạn có thể truyền một đối tượng làm đối số theo cách thông thường. Ví dụ: hàm sau nhận một đối tượng `Time` làm đối số.

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

Khi chúng ta gọi nó, chúng ta có thể truyền `lunch` làm đối số.

```
print_time(lunch)
```

```
11:59:01
```

Đối tượng là giá trị trả về

Các hàm có thể trả về các đối tượng. Ví dụ: `make_time` nhận các tham số được gọi là `hour`, `minute` và `second`, lưu trữ chúng dưới dạng thuộc tính trong một đối tượng `Time` và trả về đối tượng mới.

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

Có thể ngạc nhiên là các tham số có cùng tên với các thuộc tính, nhưng đó là một cách phổ biến để viết một hàm như thế này. Đây là cách chúng ta sử dụng `make_time` để tạo một đối tượng `Time`.

```
time = make_time(11, 59, 1)
print_time(time)
```

```
11:59:01
```

Đối tượng là biến đổi được

Giả sử bạn sẽ xem một bộ phim, như *Monty Python and the Holy Grail*, bắt đầu lúc `9:20 PM` và kéo dài `92` phút, tức là 01 giờ `32` phút. Phim sẽ kết thúc lúc mấy giờ?

Đầu tiên, chúng ta sẽ tạo một đối tượng `Time` đại diện cho thời gian bắt đầu.

```
start = make_time(9, 20, 0)
print_time(start)
```

```
09:20:00
```

Để tìm thời gian kết thúc, chúng ta có thể sửa đổi các thuộc tính của đối tượng `Time`, thêm thời lượng của phim.

```
start.hour += 1
start.minute += 32
print_time(start)
```

```
10:52:00
```

Phim sẽ kết thúc lúc 10:52 PM.

Hãy đóng gói phép tính này trong một hàm và tổng quát hóa nó để lấy thời lượng của phim trong ba tham số: `hours`, `minutes` và `seconds`.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds
```

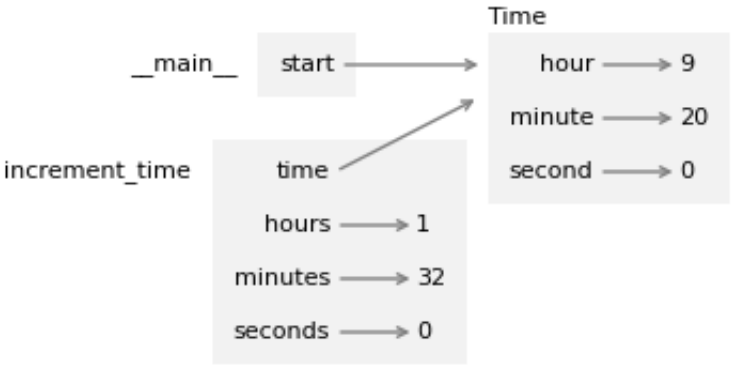
Đây là một ví dụ minh họa hiệu ứng.

```
start = make_time(9, 20, 0)
increment_time(start, 1, 32, 0)
print_time(start)
```

```
10:52:00
```

Sơ đồ ngăn xếp sau đây cho thấy trạng thái của chương trình ngay trước khi `increment_time` sửa đổi đối tượng.

```
from diagram import Frame, Binding, Value, Stack

d1 = dict(hour=9, minute=20, second=0)
obj1 = make_frame(d1, name='Time', dy=-0.25, offsetx=0.78)

binding1 = make_binding('start', frame, draw_value=False, dx=0.7)
frame1 = Frame([binding1], name='__main__', loc='left', offsetx=-0.2)

binding2 = Binding(Value('time'), draw_value=False, dx=0.7, dy=0.35)
binding3 = make_binding('hours', 1)
binding4 = make_binding('minutes',32)
binding5 = make_binding('seconds', 0)
frame2 = Frame([binding2, binding3, binding4, binding5], name='increment_time',
               loc='left', dy=-0.25, offsetx=0.08)

stack = Stack([frame1, frame2], dx=-0.3, dy=-0.5)


from diagram import Bbox

width, height, x, y = [3.4, 1.89, 1.75, 1.5]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = obj1.draw(ax, x+0.23, y)
bbox = Bbox.union([bbox1, bbox2])
# adjust(x, y, bbox)
```



Bên trong hàm, `time` là một biệt danh của `start`, vì vậy khi `time` bị sửa đổi, `start` thay đổi.

Hàm này hoạt động, nhưng sau khi chạy, chúng ta còn lại một biến có tên `start` tham chiếu đến một đối tượng đại diện cho thời gian *kết thúc*, và chúng ta không còn có một đối tượng đại diện cho thời gian bắt đầu. Sẽ tốt hơn nếu để `start` không thay đổi và tạo một đối tượng mới để đại diện cho thời gian kết thúc. Chúng ta có thể làm điều đó bằng cách sao chép `start` và sửa đổi bản sao.

Sao chép

Mô-đun `copy` cung cấp một hàm có tên `copy` có thể nhân đôi bất kỳ đối tượng nào. Chúng ta có thể nhập nó như thế này.

```
from copy import copy
```

Để xem cách nó hoạt động, hãy bắt đầu với một đối tượng `Time` mới đại diện cho thời gian bắt đầu của bộ phim.

```
start = make_time(9, 20, 0)
```

Và tạo một bản sao.

```
end = copy(start)
```

Bây giờ `start` và `end` chứa cùng một dữ liệu.

```
print_time(start)
print_time(end)
```

```
09:20:00
09:20:00
```

Nhưng toán tử `is` xác nhận rằng chúng không phải là cùng một đối tượng.

```
start is end
```

```
False
```

Hãy xem toán tử `==` hoạt động như thế nào.

```
start == end
```

```
False
```

Bạn có thể mong đợi toán tử `==` sẽ cho kết quả là `True` vì các đối tượng chứa cùng một dữ liệu. Nhưng đối với các lớp do lập trình viên định nghĩa, hành vi mặc định của toán tử `==` giống như toán tử `is` - nó kiểm tra danh tính chứ không phải sự tương đương.

Hàm tinh khiết

Chúng ta có thể sử dụng `copy` để viết các hàm tinh khiết không sửa đổi các tham số của chúng. Ví dụ: đây là một hàm nhận một đối tượng `Time` và một khoảng thời gian tính bằng giờ, phút và giây. Nó tạo một bản sao của đối tượng gốc, sử dụng `increment_time` để sửa đổi bản sao và trả về nó.

```
def add_time(time, hours, minutes, seconds):
    total = copy(time)
    increment_time(total, hours, minutes, seconds)
    return total
```

Đây là cách chúng ta sử dụng nó.

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

```
10:52:00
```

Giá trị trả về là một đối tượng mới đại diện cho thời gian kết thúc của bộ phim. Và chúng ta có thể xác nhận rằng `start` không thay đổi.

```
print_time(start)
```

```
09:20:00
```

`add_time` là **một hàm tinh khiết** vì nó không sửa đổi bất kỳ đối tượng nào được truyền cho nó làm đối số và hiệu ứng duy nhất của nó là trả về một giá trị.

Bất cứ điều gì có thể được thực hiện với các hàm không tinh khiết cũng có thể được thực hiện với các hàm tinh khiết. Trên thực tế, một số ngôn ngữ lập trình chỉ cho phép các hàm tinh khiết. Các chương trình sử dụng hàm tinh khiết có thể ít lỗi hơn, nhưng các hàm không tinh khiết đôi khi tiện lợi hơn và có thể hiệu quả hơn.

Nói chung, tôi khuyên bạn nên viết các hàm tinh khiết bất cứ khi nào có lý và chỉ sử dụng các hàm không tinh khiết nếu có lợi thế hấp dẫn. Cách tiếp cận này có thể được gọi là **phong cách lập trình hàm**.

Nguyên mẫu và chỉnh sửa

Trong ví dụ trước, `increment_time` và `add_time` dường như hoạt động, nhưng nếu chúng ta thử một ví dụ khác, chúng ta sẽ thấy rằng chúng không hoàn toàn chính xác.

Giả sử bạn đến rạp chiếu phim và phát hiện ra rằng bộ phim bắt đầu lúc `9:40`, không phải `9:20`. Đây là những gì xảy ra khi chúng ta tính thời gian kết thúc được cập nhật.

```
start = make_time(9, 40, 0)
end = add_time(start, 1, 32, 0)
print_time(end)
```

```
10:72:00
```

Kết quả không phải là thời gian hợp lệ. Vấn đề là `increment_time` không xử lý các trường hợp số giây hoặc phút cộng lại lớn hơn `60`.

Đây là một phiên bản cải tiến kiểm tra xem `second` có vượt quá hoặc bằng `60` không - nếu có, nó tăng `minute` - sau đó kiểm tra xem `minute` có vượt quá hoặc bằng `60` không - nếu có, nó tăng `hour`.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Sửa `increment_time` cũng sửa `add_time`, sử dụng nó. Vì vậy, bây giờ ví dụ trước hoạt động chính xác.

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

```
11:12:00
```

Nhưng hàm này vẫn chưa chính xác, vì các đối số có thể lớn hơn `60`. Ví dụ: giả sử chúng ta được cung cấp thời gian chạy là `92` phút, thay vì `1` giờ và `32` phút. Chúng ta có thể gọi `add_time` như thế này.

```
end = add_time(start, 0, 92, 0)
print_time(end)
```

```
10:72:00
```

Kết quả không phải là thời gian hợp lệ. Vì vậy, hãy thử một cách tiếp cận khác, sử dụng hàm `divmod`. Chúng ta sẽ tạo một bản sao của `start` và sửa đổi nó bằng cách tăng thuộc tính `minute`.

```
end = copy(start)
end.minute = start.minute + 92
end.minute
```

```
132
```

Bây giờ `minute` là `132`, tức là `2` giờ và `12` phút. Chúng ta có thể sử dụng `divmod` để chia cho `60` và trả về số giờ nguyên và số phút còn lại.

```
carry, end.minute = divmod(end.minute, 60)
carry, end.minute
```

```
(2, 12)
```

Bây giờ `minute` là chính xác và chúng ta có thể thêm các giờ vào `hour`.

```
end.hour += carry
print_time(end)
```

```
11:12:00
```

Kết quả là một thời gian hợp lệ. Chúng ta có thể làm điều tương tự với `hour` và `second`, và đóng gói toàn bộ quá trình trong một hàm.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    carry, time.second = divmod(time.second, 60)
    carry, time.minute = divmod(time.minute + carry, 60)
    carry, time.hour = divmod(time.hour + carry, 60)
```

Với phiên bản `increment_time` này, `add_time` hoạt động chính xác, ngay cả khi các đối số vượt quá `60`.

```
end = add_time(start, 0, 90, 120)
print_time(end)
```

```
11:12:00
```

Phần này chứng minh một kế hoạch phát triển chương trình mà tôi gọi là **nguyên mẫu và chỉnh sửa**. Chúng ta bắt đầu với một nguyên mẫu đơn giản hoạt động chính xác cho ví dụ đầu tiên. Sau đó, chúng ta đã thử nghiệm nó với các ví dụ khó hơn - khi chúng ta tìm thấy lỗi, chúng ta đã sửa đổi chương trình để sửa lỗi, giống như vá lốp xe bị thủng.

Cách tiếp cận này có thể hiệu quả, đặc biệt nếu bạn chưa hiểu sâu về vấn đề. Nhưng các sửa chữa tăng dần có thể tạo ra mã không cần thiết phức tạp - vì nó xử lý nhiều trường hợp đặc biệt - và không đáng tin cậy - vì khó biết liệu bạn đã tìm thấy tất cả các lỗi hay chưa.

Phát triển thiết kế đầu tiên

Một kế hoạch thay thế là **phát triển thiết kế đầu tiên**, bao gồm nhiều kế hoạch hơn trước khi tạo nguyên mẫu. Trong quá trình thiết kế đầu tiên, đôi khi một cái nhìn sâu sắc cấp cao về vấn đề sẽ giúp lập trình dễ dàng hơn nhiều.

Trong trường hợp này, cái nhìn sâu sắc là chúng ta có thể nghĩ về một đối tượng `Time` là một số ba chữ số ở cơ số 60 - còn được gọi là hệ thập lục phân. Thuộc tính `second` là "cột đơn vị", thuộc tính `minute` là "cột sáu mươi", và thuộc tính `hour` là "cột ba nghìn sáu trăm". Khi chúng ta viết `increment_time`, chúng ta thực sự đang thực hiện phép cộng cơ số 60, đó là lý do tại sao chúng ta phải chuyển từ cột này sang cột tiếp theo.

Quan sát này gợi ý một cách tiếp cận khác đối với toàn bộ vấn đề - chúng ta có thể chuyển đổi các đối tượng `Time` thành số nguyên và tận dụng thực tế là Python biết cách thực hiện phép toán số nguyên.

Dưới đây là một hàm chuyển đổi từ `Time` thành số nguyên.

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

Kết quả là số giây kể từ đầu thời điểm bắt đầu một ngày. Ví dụ, `01:01:01` tương ứng với 1 giờ, 1 phút và 1 giây kể từ thời điểm bắt đầu một đầu ngày, với tổng số giây là 3600 giây, 60 giây và 1 giây.

```
time = make_time(1, 1, 1)
print_time(time)
time_to_int(time)
```

```
01:01:01
3661
```

Và đây là một hàm đi theo hướng ngược lại - chuyển đổi một số nguyên thành một đối tượng `Time` - sử dụng hàm `divmod`.

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

Chúng ta có thể kiểm tra nó bằng cách chuyển đổi ví dụ trước trở lại `Time`.

```
time = int_to_time(3661)
print_time(time)
```

```
01:01:01
```

Sử dụng các hàm này, chúng ta có thể viết một phiên bản ngắn gọn hơn của `add_time`.

```
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)
```

Dòng đầu tiên chuyển đổi các đối số thành một đối tượng `Time` được gọi là `duration`. Dòng thứ hai chuyển đổi `time` và `duration` thành giây và cộng chúng lại. Dòng thứ ba chuyển đổi tổng thành một đối tượng `Time` và trả về nó.

Đây là cách nó hoạt động.

```
start = make_time(9, 40, 0)
end = add_time(start, 1, 32, 0)
print_time(end)
```

```
11:12:00
```

Theo một số cách, việc chuyển đổi từ cơ số 60 sang cơ số 10 và ngược lại khó hơn là chỉ xử lý thời gian. Chuyển đổi cơ số trừu tượng hơn; trực giác của chúng ta để xử lý các giá trị thời gian tốt hơn.

Nhưng nếu chúng ta có cái nhìn sâu sắc để coi thời gian là số cơ số 60 - và đầu tư công sức để viết các hàm chuyển đổi `time_to_int` và `int_to_time` - chúng ta sẽ có được một chương trình ngắn gọn hơn, dễ đọc, dễ gỡ lỗi, và đáng tin cậy hơn.

Ngoài ra, việc thêm các tính năng sau này cũng dễ dàng hơn. Ví dụ: hãy tưởng tượng trừ hai đối tượng `Time` để tìm khoảng thời gian giữa chúng. Cách tiếp cận ngây thơ là thực hiện phép trừ với việc vay mượn. Sử dụng các hàm chuyển đổi dễ dàng hơn và có nhiều khả năng chính xác hơn.

Ngược đời là đôi khi làm cho một vấn đề khó hơn - hoặc tổng quát hơn - lại dễ dàng hơn, vì có ít trường hợp đặc biệt hơn và ít cơ hội hơn để mắc lỗi.

Gỡ lỗi

Python cung cấp một số hàm tích hợp sẵn hữu ích để kiểm tra và gỡ lỗi các chương trình làm việc với các đối tượng. Ví dụ: nếu bạn không chắc chắn về kiểu của một đối tượng, bạn có thể hỏi.

```
type(start)
```

Bạn cũng có thể sử dụng `isinstance` để kiểm tra xem một đối tượng có phải là một trường hợp của một lớp cụ thể hay không.

```
isinstance(end, Time)
```

Nếu bạn không chắc chắn liệu một đối tượng có một thuộc tính cụ thể hay không, bạn có thể sử dụng hàm tích hợp sẵn `hasattr`.

```
hasattr(start, 'hour')
```

Để lấy tất cả các thuộc tính và giá trị của chúng trong một từ điển, bạn có thể sử dụng `vars`.

```
vars(start)
```

Mô-đun `structshape`, mà chúng ta đã thấy trong Chương 11, cũng hoạt động với các kiểu do lập trình viên định nghĩa.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshap
```

```
from structshape import structshape

t = start, end
structshape(t)
```

Thuật ngữ

object-oriented programming - lập trình hướng đối tượng: Một phong cách lập trình sử dụng các đối tượng để tổ chức mã và dữ liệu.

class - lớp: Một kiểu do lập trình viên định nghĩa. Định nghĩa lớp tạo ra một đối tượng lớp mới.

class object - đối tượng lớp: Một đối tượng đại diện cho một lớp - nó là kết quả của định nghĩa lớp.

instantiation - khởi tạo: Quá trình tạo một đối tượng thuộc về một lớp.

instance - trường hợp: Một đối tượng thuộc về một lớp.

attribute - thuộc tính: Một biến liên kết với một đối tượng, còn được gọi là biến trường hợp.

object diagram - sơ đồ đối tượng: Một biểu diễn đồ họa của một đối tượng, các thuộc tính của nó và giá trị của chúng.

format specifier - bộ định dạng: Trong một f-string, bộ định dạng xác định cách một giá trị được chuyển đổi thành một chuỗi.

pure function - hàm tinh khiết: Một hàm không sửa đổi các tham số của nó hoặc không có tác dụng nào khác ngoài trả về một giá trị.

functional programming style - phong cách lập trình hàm: Một cách lập trình sử dụng các hàm tinh khiết bất cứ khi nào có thể.

prototype and patch - nguyên mẫu và chỉnh sửa: Một cách phát triển chương trình bằng cách bắt đầu với một bản nháp thô và từ từ thêm các tính năng mới, và sửa lỗi.

design-first development - phát triển thiết kế đầu tiên: Một cách phát triển chương trình với kế hoạch cẩn thận hơn so với nguyên mẫu và vá lỗi.

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

Có rất nhiều thuật ngữ mới trong chương này. Một cuộc trò chuyện với trợ lý ảo có thể giúp củng cố sự hiểu biết của bạn. Hãy xem xét việc hỏi:

- "Sự khác biệt giữa lớp và kiểu là gì?"
- "Sự khác biệt giữa đối tượng và trường hợp là gì?"
- "Sự khác biệt giữa biến và thuộc tính là gì?"
- "Ưu điểm và nhược điểm của các hàm tinh khiết so với các hàm không tinh khiết là gì?"

Bởi vì chúng ta mới chỉ bắt đầu với lập trình hướng đối tượng, nên mã trong chương này không phải là chuẩn Pythonic - nó không phải là loại mã mà các lập trình viên có kinh nghiệm viết. Nếu bạn hỏi trợ lý ảo để được giúp đỡ với các bài tập, bạn có thể sẽ thấy các tính năng mà chúng tôi chưa đề cập đến. Đặc biệt, bạn có thể thấy một phương thức được gọi là `__init__` được sử dụng để khởi tạo các thuộc tính của một trường hợp.

Nếu những tính năng này có ý nghĩa với bạn, hãy tiếp tục và sử dụng chúng. Nhưng nếu không, hãy kiên nhẫn - chúng ta sẽ sớm đến đó. Trong thời gian chờ đợi, hãy xem bạn có thể giải quyết các bài tập sau chỉ bằng cách sử dụng các tính năng mà chúng ta đã đề cập cho đến nay hay không.

Ngoài ra, trong chương này, chúng ta đã thấy một ví dụ về một bộ định dạng. Để biết thêm thông tin, hãy hỏi "Các bộ định dạng nào có thể được sử dụng trong một f-string của Python?"

Bài tập 1

Viết một hàm có tên `subtract_time` nhận hai đối tượng `Time` và trả về khoảng thời gian giữa chúng tính bằng giây - giả sử rằng chúng là hai thời điểm trong cùng một ngày.

Đây là một phác thảo của hàm để giúp bạn bắt đầu.

```
def subtract_time(t1, t2):
    """Tính toán độ chênh lệch giữa hai mốc thời gian tính bằng giây.

    >>> subtract_time(make_time(3, 2, 1), make_time(3, 2, 0))
    1
    >>> subtract_time(make_time(3, 2, 1), make_time(3, 0, 0))
    121
    >>> subtract_time(make_time(11, 12, 0), make_time(9, 40, 0))
    5520
    """
    return None
```

Bạn có thể sử dụng `doctest` để kiểm tra hàm của mình.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(subtract_time)
```

Bài tập 2

Viết một hàm có tên `is_after` nhận hai đối tượng `Time` và trả về `True` nếu thời gian đầu tiên xảy ra sau thời gian thứ hai trong cùng một ngày, và `False` trong trường hợp khác.

Đây là một phác thảo của hàm để giúp bạn bắt đầu.

```
def is_after(t1, t2):
    """Kiểm tra xem `t1` có sau `t2` hay không.

    >>> is_after(make_time(3, 2, 1), make_time(3, 2, 0))
    True
    >>> is_after(make_time(3, 2, 1), make_time(3, 2, 1))
    False
    >>> is_after(make_time(11, 12, 0), make_time(9, 40, 0))
    True
    """
    return None
```

Bạn có thể sử dụng `doctest` để kiểm tra hàm của mình.

```
run_doctests(is_after)
```

Bài tập 3

Đây là định nghĩa cho một lớp `Date` đại diện cho một ngày - tức là năm, tháng và ngày trong tháng.

```
class Date:
    """Biểu diễn năm, tháng và ngày."""
```

1. Viết một hàm có tên `make_date` nhận `year`, `month`, và `day` làm tham số, tạo một đối tượng `Date`, gán các tham số vào thuộc tính và trả về kết quả là đối tượng mới. Tạo một đối tượng đại diện cho ngày 22 tháng 6 năm 1933.
2. Viết một hàm có tên `print_date` nhận một đối tượng `Date`, sử dụng f-string để định dạng các thuộc tính và in kết quả. Nếu bạn kiểm tra nó với `Date` mà bạn đã tạo, kết quả sẽ là `1933-06-22`.
3. Viết một hàm có tên `is_after` nhận hai đối tượng `Date` làm tham số 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 hàm có tên `date_to_tuple` nhận một đối tượng `Date` và trả về một tuple chứa các thuộc tính của nó theo thứ tự năm, tháng, ngày.

Bạn có thể sử dụng phác thảo hàm này để bắt đầu.

```
def make_date(year, month, day):
    return None
```

Bạn có thể sử dụng các ví dụ này để kiểm tra `make_date`.

```
birthday1 = make_date(1933, 6, 22)
```

```
birthday2 = make_date(1933, 9, 17)
```

Bạn có thể sử dụng phác thảo hàm này để bắt đầu.

```
def print_date(date):
    print('')
```

Bạn có thể sử dụng ví dụ này để kiểm tra `print_date`.

```
print_date(birthday1)
```

Bạn có thể sử dụng phác thảo hàm này để bắt đầu.

```
def is_after(date1, date2):
    return None
```

Bạn có thể sử dụng các ví dụ này để kiểm tra `is_after`.

```
is_after(birthday1, birthday2)  # nên là False
```

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