```
# Ô 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');
download('https://github.com/ramalho/jupyturtle/releases/download/2024-03/jupyturtl

import thinkpython
```

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

Lớp và đối tượng

Tại thời điểm này, chúng ta đã định nghĩa các lớp và tạo các đối tượng đại diện cho thời gian trong ngày và ngày trong năm. Và chúng ta đã định nghĩa các phương thức tạo, sửa đổi và thực hiện các phép tính với các đối tượng này.

Trong chương này, chúng ta sẽ tiếp tục chuyến tham quan lập trình hướng đối tượng (OOP) bằng cách định nghĩa các lớp đại diện cho các đối tượng hình học, bao gồm điểm, đường thẳng, hình chữ nhật và hình tròn. Chúng ta sẽ viết các phương thức tạo và sửa đổi các đối tượng này, và chúng ta sẽ sử dụng mô-đun `jupyturtle` để vẽ chúng.

Tôi sẽ sử dụng các lớp này để minh họa các chủ đề OOP bao gồm nhận dạng và sự tương đương của đối tượng, sao chép nông, sao chép sâu, và tính đa hình.

Tạo một điểm

Trong đồ họa máy tính, một vị trí trên màn hình thường được biểu diễn bằng một cặp tọa độ trong mặt phẳng toạ độ `x-y`. Theo quy ước, điểm `(0, 0)` thường đại diện cho góc trên bên trái của màn hình, và `(x, y)` đại diện cho điểm `x` đơn vị sang phải và `y` đơn vị xuống từ gốc tọa độ. So với hệ tọa độ hai chiều (2D) mà bạn có thể đã thấy trong lớp toán, trục `y` bị đảo ngược.

Có một số cách chúng ta có thể biểu diễn một điểm trong Python:

- Chúng ta có thể lưu trữ các tọa độ riêng biệt trong hai biến, `x` và `y`.
- Chúng ta có thể lưu trữ các tọa độ dưới dạng các phần tử trong một danh sách hoặc tuple.
- Chúng ta có thể tạo một kiểu mới để đại diện cho các điểm dưới dạng đối tượng.

Trong lập trình hướng đối tượng, phong cách lập trình chuẩn nhất là tạo một kiểu mới. Để làm điều đó, chúng ta sẽ bắt đầu với định nghĩa lớp cho `Point`.

```
class Point:
    """Biểu diễn một điểm trong không gian 2 chiều."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'Point({self.x}, {self.y})'
```

Phương thức `__init__` nhận các tọa độ làm tham số và gán chúng cho các thuộc tính `x` và `y`. Phương thức `__str__` trả về một chuỗi biểu diễn của `Point`.

Bây giờ chúng ta có thể khởi tạo và hiển thị một đối tượng `Point` như thế này.

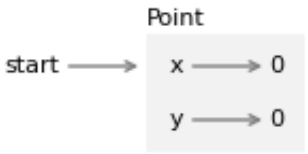
```
start = Point(0, 0)
print(start)
```

```
Point(0, 0)
```

Sơ đồ sau đây cho thấy trạng thái của đối tượng mới.

```
from diagram import make_frame, make_binding

d1 = vars(start)
frame = make_frame(d1, name='Point', dy=-0.25, offsetx=0.18)
binding = make_binding('start', frame)
```

```
from diagram import diagram, adjust

width, height, x, y = [1.41, 0.89, 0.26, 0.5]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```



Như thường lệ, một kiểu do lập trình viên định nghĩa được biểu diễn bằng một hộp có tên của kiểu ở bên ngoài và các thuộc tính bên trong.

Nói chung, các kiểu do lập trình viên định nghĩa là có thể thay đổi, vì vậy chúng ta có thể viết một phương thức như `translate` nhận hai tham số, `dx` và `dy`, và thêm chúng vào các thuộc tính `x` và `y`.

```
%%add_method_to Point

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
```

Hàm này dịch chuyển điểm từ một vị trí trong mặt phẳng sang vị trí khác. Nếu chúng ta không muốn sửa đổi một điểm hiện có, chúng ta có thể sử dụng `copy` để sao chép đối tượng gốc và sau đó sửa đổi bản sao.

```
from copy import copy

end1 = copy(start)
end1.translate(300, 0)
print(end1)
```

```
Point(300, 0)
```

Chúng ta có thể đóng gói các bước đó trong một phương thức khác gọi là `translated`.

```
%%add_method_to Point

    def translated(self, dx=0, dy=0):
        point = copy(self)
        point.translate(dx, dy)
        return point
```

Cũng giống như hàm tích hợp `sort` sửa đổi một danh sách và hàm `sorted` tạo ra một danh sách mới, bây giờ chúng ta có phương thức `translate` sửa đổi một `Point` và phương thức `translated` tạo ra một `Point` mới.

Đây là một ví dụ:

```
end2 = start.translated(0, 150)
print(end2)
```

```
Point(0, 150)
```

Trong phần tiếp theo, chúng ta sẽ sử dụng các điểm này để định nghĩa và vẽ một đường thẳng.

Tạo một đường thẳng

Bây giờ, hãy định nghĩa một lớp đại diện cho đoạn thẳng giữa hai điểm. Như thường lệ, chúng ta sẽ bắt đầu với phương thức `__init__` và phương thức `__str__`.

```
class Line:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __str__(self):
        return f'Line({self.p1}, {self.p2})'
```

Với hai phương thức đó, chúng ta có thể khởi tạo và hiển thị một đối tượng `Line` mà chúng ta sẽ sử dụng để đại diện cho trục x.

```
line1 = Line(start, end1)
print(line1)
```

```
Line(Point(0, 0), Point(300, 0))
```

Khi chúng ta gọi `print` và truyền `line` làm tham số, `print` sẽ gọi phương thức `__str__` trên đối tượng `line`. Phương thức `__str__` sử dụng f-string để tạo ra một chuỗi biểu diễn cho đối tượng `line`.

F-string chứa hai biểu thức trong dấu ngoặc nhọn, `self.p1` và `self.p2`. Khi các biểu thức này được đánh giá, kết quả là các đối tượng `Point`. Sau đó, khi chúng được chuyển đổi thành chuỗi, phương thức `__str__` từ lớp `Point` sẽ được gọi.

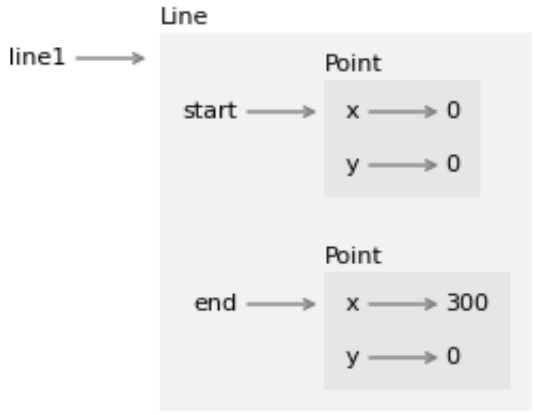
Đó là lý do tại sao, khi chúng ta hiển thị một `Line`, kết quả sẽ chứa các chuỗi biểu diễn của các đối tượng `Point`.

Sơ đồ đối tượng dưới đây thể hiện trạng thái của đối tượng `Line` này.

```
from diagram import Binding, Value, Frame

d1 = vars(line1.p1)
frame1 = make_frame(d1, name='Point', dy=-0.25, offsetx=0.17)

d2 = vars(line1.p2)
frame2 = make_frame(d2, name='Point', dy=-0.25, offsetx=0.17)

binding1 = Binding(Value('start'), frame1, dx=0.4)
binding2 = Binding(Value('end'), frame2, dx=0.4)
frame3 = Frame([binding1, binding2], name='Line', dy=-0.9, offsetx=0.4, offsety=-
```

```
binding = make_binding('line1', frame3)
```

```
width, height, x, y = [2.45, 2.12, 0.27, 1.76]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```



Các chuỗi biểu diễn và sơ đồ đối tượng hữu ích cho việc gỡ lỗi, nhưng mục đích của ví dụ này là để tạo ra đồ họa, không phải văn bản! Vì vậy, chúng ta sẽ sử dụng mô-đun `jupyturtle` để vẽ các đoạn thẳng trên màn hình.

Như chúng ta đã làm trong Chương 4, chúng ta sẽ sử dụng `make_turtle` để tạo một đối tượng `Turtle` và một khung vẽ (canvas) nhỏ nơi nó có thể vẽ. Để vẽ các đoạn thẳng, chúng ta sẽ sử dụng hai hàm mới từ mô-đun `jupyturtle`:

- `jumpto`, hàm nhận hai tọa độ và di chuyển đối tượng `Turtle` đến vị trí đã cho mà không vẽ một đoạn thẳng, và
- `moveto`, hàm di chuyển đối tượng `Turtle` từ vị trí hiện tại đến vị trí đã cho, và vẽ một đoạn thẳng giữa chúng.

Đây là cách chúng ta `import` chúng.

```
from jupyturtle import make_turtle, jumpto, moveto
```

Và đây là một phương pháp để vẽ một `Line`.

```
%%add_method_to Line

    def draw(self):
        jumpto(self.p1.x, self.p1.y)
        moveto(self.p2.x, self.p2.y)
```

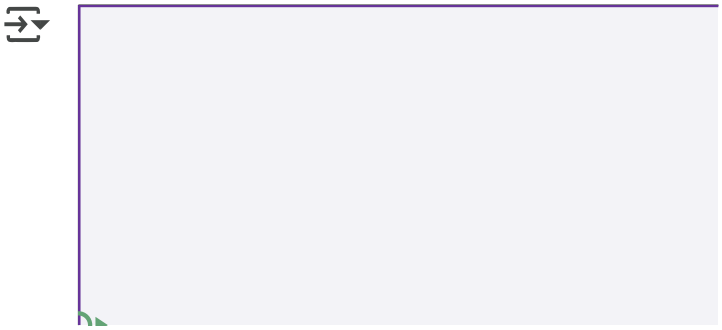
Để minh họa cách sử dụng, tôi sẽ tạo một đoạn thẳng thứ hai đại diện cho trục `y`.

```
line2 = Line(start, end2)
print(line2)
```

```
Line(Point(0, 0), Point(0, 150))
```

Và sau đó vẽ các trục.

```
make_turtle()
line1.draw()
line2.draw()
```



Khi chúng ta định nghĩa và vẽ thêm nhiều đối tượng, chúng ta sẽ sử dụng lại các đoạn thẳng này. Nhưng trước tiên, hãy cùng nói về sự tương đương và nhận dạng của đối tượng.

Sự tương đương và nhận dạng

Giả sử chúng ta tạo ra hai điểm có cùng tọa độ.

```
p1 = Point(200, 100)
p2 = Point(200, 100)
```

Nếu chúng ta sử dụng toán tử `==` để so sánh chúng, chúng ta sẽ nhận được hành vi mặc định cho các kiểu dữ liệu do lập trình viên định nghĩa — kết quả là `True` chỉ khi chúng là cùng một đối tượng, điều mà trong trường hợp này là không đúng.

```
p1 == p2
```

```
False
```

Nếu chúng ta muốn thay đổi hành vi đó, chúng ta có thể cung cấp một phương thức đặc biệt gọi là `__eq__` để định nghĩa hai đối tượng `Point` có nghĩa là bằng nhau khi nào.

```
%%add_method_to Point

def __eq__(self, other):
    return (self.x == other.x) and (self.y == other.y)
```

Định nghĩa này coi hai `Point` là bằng nhau nếu các thuộc tính của chúng bằng nhau. Bây giờ, khi chúng ta sử dụng toán tử `==`, nó sẽ gọi phương thức `__eq__`, điều này chỉ ra rằng `p1` và `p2` được coi là bằng nhau.

```
p1 == p2
```

```
True
```

Tuy nhiên, toán tử `is` vẫn chỉ ra rằng chúng là hai đối tượng khác nhau.

```
p1 is p2
```

```
False
```

Không thể ghi đè toán tử `is` — nó luôn kiểm tra xem các đối tượng có phải là bản sao giống hệt nhau hay không. Tuy nhiên, đối với các kiểu dữ liệu do lập trình viên định nghĩa, bạn có thể ghi đè toán tử `==` để kiểm tra xem các đối tượng có tương đương hay không. Và bạn có thể định nghĩa nghĩa là gì khi hai đối tượng được coi là tương đương.

Tạo một hình chữ nhật

Bây giờ, hãy định nghĩa một lớp đại diện và vẽ các hình chữ nhật. Để đơn giản, chúng ta sẽ giả sử rằng các hình chữ nhật là thẳng đứng hoặc nằm ngang, không nghiêng. Bạn nghĩ chúng ta nên sử dụng những thuộc tính nào để chỉ định vị trí và kích thước của một hình chữ nhật?

Có ít nhất hai khả năng:

- Bạn có thể chỉ định chiều rộng và chiều cao của hình chữ nhật và vị trí của một góc.
- Bạn có thể chỉ định hai góc đối diện.

Ở thời điểm này, khó mà nói cái nào tốt hơn cái nào, vì vậy hãy triển khai phương án đầu tiên.

Dưới đây là định nghĩa lớp.

```
class Rectangle:
    """Biểu diễn một hình chữ nhật.

    các thuộc tính: chiều rộng (width), chiều cao (height), số đo góc (corner).
    """
    def __init__(self, width, height, corner):
        self.width = width
        self.height = height
        self.corner = corner

    def __str__(self):
        return f'Rectangle({self.width}, {self.height}, {self.corner})'
```

Như thường lệ, phương thức `__init__` gán các tham số vào các thuộc tính và phương thức `__str__` trả về một chuỗi biểu diễn của đối tượng. Bây giờ chúng ta có thể khởi tạo một đối tượng `Rectangle`, sử dụng một đối tượng `Point` làm vị trí của góc trên bên trái.

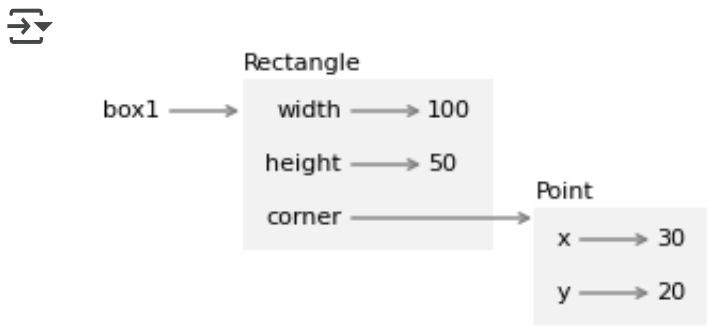
```
corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)
print(box1)
```

```
Rectangle(100, 50, Point(30, 20))
```

Sơ đồ dưới đây thể hiện trạng thái của đối tượng này.

```
from diagram import Binding, Value

def make_rectangle_binding(name, box, **options):
    d1 = vars(box.corner)
    frame_corner = make_frame(d1, name='Point', dy=-0.25, offsetx=0.07)

    d2 = dict(width=box.width, height=box.height)
    frame = make_frame(d2, name='Rectangle', dy=-0.25, offsetx=0.45)
    binding = Binding(Value('corner'), frame1, dx=0.92, draw_value=False, **optio
    frame.bindings.append(binding)

    binding = Binding(Value(name), frame)
    return binding, frame_corner

binding_box1, frame_corner1 = make_rectangle_binding('box1', box1)
```

```
from diagram import Bbox

width, height, x, y = [2.83, 1.49, 0.27, 1.1]
ax = diagram(width, height)
bbox1 = binding_box1.draw(ax, x, y)
bbox2 = frame_corner1.draw(ax, x+1.85, y-0.6)
bbox = Bbox.union([bbox1, bbox2])
#adjust(x, y, bbox)
```



Để vẽ một hình chữ nhật, chúng ta sẽ sử dụng phương thức sau để tạo bốn đối tượng `Point` đại diện cho các góc.

```
%%add_method_to Rectangle

    def make_points(self):
        p1 = self.corner
        p2 = p1.translated(self.width, 0)
        p3 = p2.translated(0, self.height)
        p4 = p3.translated(-self.width, 0)
        return p1, p2, p3, p4
```

Sau đó, chúng ta sẽ tạo bốn đối tượng `Line` để đại diện cho các cạnh.

```
%%add_method_to Rectangle

    def make_lines(self):
        p1, p2, p3, p4 = self.make_points()
        return Line(p1, p2), Line(p2, p3), Line(p3, p4), Line(p4, p1)
```

Sau đó, chúng ta sẽ vẽ các cạnh.

```
%%add_method_to Rectangle

    def draw(self):
        lines = self.make_lines()
        for line in lines:
            line.draw()
```

Dưới đây là một ví dụ.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
```

Hình vẽ bao gồm hai đoạn thẳng để đại diện cho các trục.

Thay đổi hình chữ nhật

Bây giờ, hãy xem xét hai phương thức thay đổi hình chữ nhật, `grow` và `translate`. Chúng ta sẽ thấy rằng `grow` hoạt động như mong đợi, nhưng `translate` lại có một lỗi tinh vi. Hãy thử tìm ra lỗi đó trước khi tôi giải thích.

`grow` nhận hai số, `dwidth` và `dheight`, và cộng chúng vào các thuộc tính `width` và `height` của hình chữ nhật.

```
%%add_method_to Rectangle

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

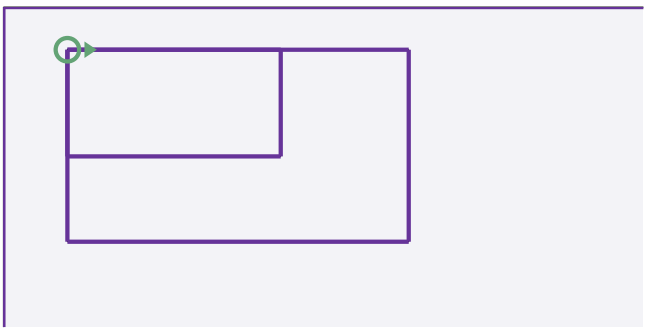
Dưới đây là một ví dụ minh họa tác dụng của phương thức bằng cách tạo một bản sao của `box1` và gọi phương thức `grow` trên bản sao đó.

```
box2 = copy(box1)
box2.grow(60, 40)
print(box2)
```

```
Rectangle(160, 90, Point(30, 20))
```

Nếu chúng ta vẽ `box1` và `box2`, chúng ta có thể xác nhận rằng phương thức `grow` hoạt động như mong đợi.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
box2.draw()
```



Bây giờ, hãy cùng xem phương thức `translate`. Nó nhận hai số, `dx` và `dy`, và di chuyển hình chữ nhật một khoảng cách được chỉ định theo hướng `x` và `y`.

```
%%add_method_to Rectangle

    def translate(self, dx, dy):
        self.corner.translate(dx, dy)
```

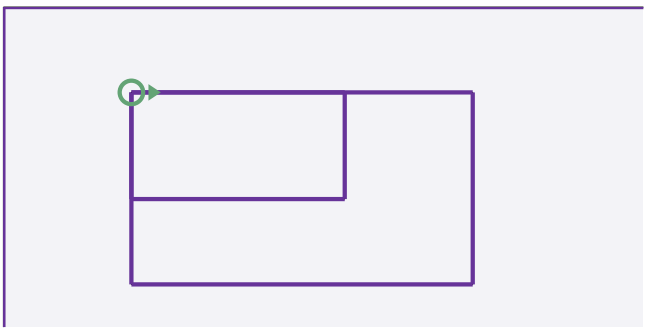
Để minh họa tác dụng, chúng ta sẽ dịch chuyển `box2` sang phải và xuống dưới.

```
box2.translate(30, 20)
print(box2)
```

```
Rectangle(160, 90, Point(60, 40))
```

Bây giờ, hãy xem điều gì xảy ra nếu chúng ta vẽ lại `box1` và `box2`.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
box2.draw()
```



Có vẻ như cả hai hình chữ nhật đều bị dịch chuyển, điều này không phải là điều chúng ta mong muốn! Phần tiếp theo sẽ giải thích điều gì đã xảy ra sai sót.

Sao chép sâu

Khi chúng ta sử dụng `copy` để nhân bản `box1`, nó sao chép đối tượng `Rectangle` nhưng không sao chép đối tượng `Point` mà nó chứa. Do đó, `box1` và `box2` là các đối tượng khác nhau, như mong muốn.

```
box1 is box2
```

```
False
```

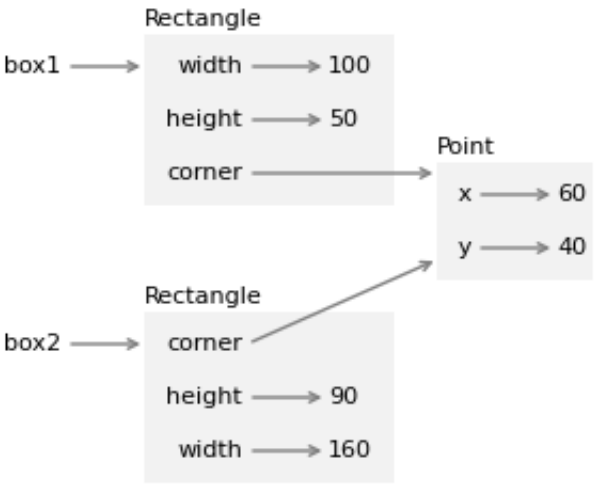
Nhưng các thuộc tính `corner` của chúng lại tham chiếu đến cùng một đối tượng.

```
box1.corner is box2.corner
```

```
True
```

Sơ đồ dưới đây thể hiện trạng thái của các đối tượng này.

```
from diagram import Stack
from copy import deepcopy

binding_box1, frame_corner1 = make_rectangle_binding('box1', box1)
binding_box2, frame_corner2 = make_rectangle_binding('box2', box2, dy=0.4)
binding_box2.value.bindings.reverse()

stack = Stack([binding_box1, binding_box2], dy=-1.3)


from diagram import Bbox

width, height, x, y = [2.76, 2.54, 0.27, 2.16]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = frame_corner1.draw(ax, x+1.85, y-0.6)
bbox = Bbox.union([bbox1, bbox2])
# adjust(x, y, bbox)
```



Những gì thực hiện `copy` được gọi là sao chép nông vì nó sao chép đối tượng nhưng không sao chép các đối tượng mà nó chứa.

Kết quả là, việc thay đổi `width` hoặc `height` của một `Rectangle` sẽ không ảnh hưởng đến đối tượng kia, nhưng thay đổi các thuộc tính của đối tượng `Point` được chia sẻ sẽ ảnh hưởng đến cả hai! Hành vi này dễ gây nhầm lẫn và dễ dẫn đến lỗi.

May mắn thay, mô-đun `copy` cung cấp một hàm khác, gọi là `deepcopy` , hàm này sao chép không chỉ đối tượng mà còn sao chép cả các đối tượng mà nó tham chiếu đến, và các đối tượng mà chúng tham chiếu đến, và cứ tiếp tục như vậy. Phép toán này được gọi là **sao chép sâu**.

Để minh họa, hãy bắt đầu với một `Rectangle` mới chứa một `Point` mới.

```
corner = Point(20, 20)
box3 = Rectangle(100, 50, corner)
print(box3)
```

```
Rectangle(100, 50, Point(20, 20))
```

Và chúng ta sẽ thực hiện một sao chép sâu.

```
from copy import deepcopy

box4 = deepcopy(box3)
```

Chúng ta có thể xác nhận rằng hai đối tượng `Rectangle` tham chiếu đến các đối tượng `Point` khác nhau.

```
box3.corner is box4.corner
```

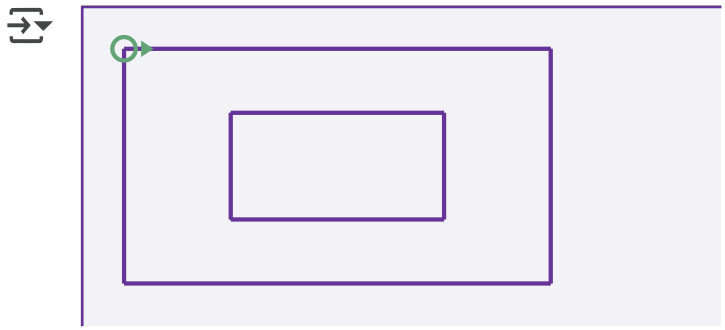
```
False
```

Vì `box3` và `box4` là các đối tượng hoàn toàn tách biệt, chúng ta có thể sửa đổi một cái mà không ảnh hưởng đến cái còn lại. Để minh họa, chúng ta sẽ di chuyển `box3` và thay đổi kích thước của `box4` .

```
box3.translate(50, 30)
box4.grow(100, 60)
```

Và chúng ta có thể xác nhận rằng tác dụng là như mong đợi.

```
make_turtle()
line1.draw()
line2.draw()
box3.draw()
box4.draw()
```



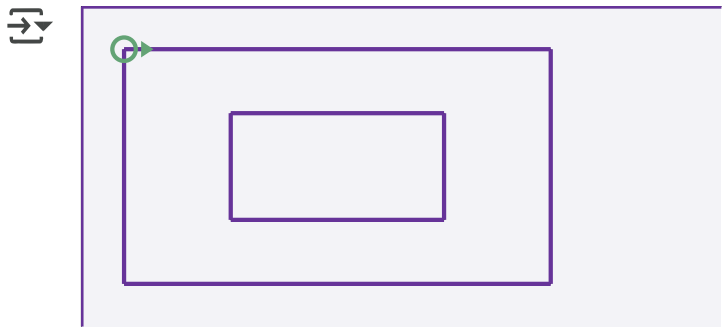
Tính đa hình

Trong ví dụ trước, chúng ta đã gọi phương thức `draw` trên hai đối tượng `Line` và hai đối tượng `Rectangle`. Chúng ta có thể làm điều tương tự ngắn gọn hơn bằng cách tạo một danh sách các đối tượng.

```
shapes = [line1, line2, box3, box4]
```

Các phần tử của danh sách này là các kiểu khác nhau, nhưng tất cả đều cung cấp một phương thức `draw`, vì vậy chúng ta có thể lặp qua danh sách và gọi `draw` trên từng phần tử.

```
make_turtle()

for shape in shapes:
    shape.draw()
```



Lần đầu tiên và lần thứ hai lặp qua vòng lặp, `shape` tham chiếu đến một đối tượng `Line`, vì vậy khi `draw` được gọi, phương thức được chạy là phương thức được định nghĩa trong lớp `Line`.

Lần thứ ba và lần thứ tư lặp qua vòng lặp, `shape` tham chiếu đến một đối tượng `Rectangle`, vì vậy khi `draw` được gọi, phương thức được chạy là phương thức được định nghĩa trong lớp `Rectangle`.

Theo một nghĩa nào đó, mỗi đối tượng biết cách tự vẽ mình. Tính năng này được gọi là **đa hình**. Từ này bắt nguồn từ gốc Hy Lạp có nghĩa là "nhiều hình dạng". Trong lập trình hướng đối tượng, đa hình là khả năng của các kiểu khác nhau cung cấp cùng một phương thức, giúp thực hiện nhiều phép tính - như vẽ hình - bằng cách gọi cùng một phương thức trên các kiểu đối tượng khác nhau.

Như một bài tập cuối chương, bạn sẽ định nghĩa một lớp mới đại diện cho một hình tròn và cung cấp một phương thức `draw`. Sau đó, bạn có thể sử dụng đa hình để vẽ đường thẳng, hình chữ nhật và hình tròn.

Gỡ lỗi

Trong chương này, chúng ta đã gặp một lỗi tinh tế xảy ra vì chúng ta đã tạo một `Point` được chia sẻ bởi hai đối tượng `Rectangle`, và sau đó chúng ta đã sửa đổi `Point`. Nói chung, có hai cách để tránh các vấn đề như thế này: bạn có thể tránh chia sẻ các đối tượng hoặc bạn có thể tránh sửa đổi chúng.

Để tránh chia sẻ các đối tượng, bạn có thể sử dụng sao chép sâu, như chúng ta đã làm trong chương này.

Để tránh sửa đổi các đối tượng, hãy xem xét thay thế các hàm không tinh khiết như `translate` bằng các hàm tinh khiết như `translated`. Ví dụ: đây là một phiên bản của `translated` tạo ra một `Point` mới và không bao giờ sửa đổi các thuộc tính của nó.

```
    def translated(self, dx=0, dy=0):
        x = self.x + dx
        y = self.y + dy
        return Point(x, y)
```

Python cung cấp các tính năng giúp dễ dàng hơn để tránh sửa đổi các đối tượng. Chúng nằm ngoài phạm vi của cuốn sách này, nhưng nếu bạn tò mò, hãy hỏi trợ lý ảo, "Làm thế nào để tôi làm cho một đối tượng Python không thể thay đổi?"

Tạo một đối tượng mới mất nhiều thời gian hơn là sửa đổi một đối tượng hiện có, nhưng sự khác biệt hiếm khi quan trọng trong thực tế. Các chương trình tránh các đối tượng được chia sẻ và các hàm không tinh khiết thường dễ phát triển, kiểm tra và gỡ lỗi hơn - và loại gỡ lỗi tốt nhất là loại bạn không phải thực hiện.

Thuật ngữ

shallow copy - sao chép nông: Một hoạt động sao chép không sao chép các đối tượng lồng nhau.

deep copy - sao chép sâu: Một hoạt động sao chép tất cả các đối tượng, kể cả các đối tượng lồng nhau.

polymorphism - tính đa hình: Khả năng của một phương thức hoặc toán tử hoạt động với nhiều loại đối tượng.

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
```

```
Exception reporting mode: Verbose
```

Hỏi trợ lý ảo

Đối với tất cả các bài tập sau, hãy cân nhắc hỏi trợ lý ảo để được giúp đỡ. Nếu bạn làm vậy, bạn sẽ muốn bao gồm một phần lời nhắc về các định nghĩa lớp cho `Point`, `Line` và `Rectangle` - nếu không, trợ lý ảo sẽ đoán về các thuộc tính và hàm của chúng, và mã nó tạo ra sẽ không hoạt động.

Bài tập 1

Viết một phương thức `__eq__` cho lớp `Line` trả về `True` nếu các đối tượng `Line` tham chiếu đến các đối tượng `Point` tương đương, theo bất kỳ thứ tự nào.

Bạn có thể sử dụng dàn ý sau để bắt đầu.

```
%%add_method_to Line

def __eq__(self, other):
    return None
```

```
%%add_method_to Line

def __eq__(self, other):
    if (self.p1 == other.p1) and (self.p2 == other.p2):
        return True
    if (self.p1 == other.p2) and (self.p2 == other.p1):
        return True
    return False
```

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

```
start1 = Point(0, 0)
start2 = Point(0, 0)
end = Point(200, 100)
```

Ví dụ này nên là `True` vì các đối tượng `Line` tham chiếu đến các đối tượng `Point` tương đương, theo cùng một thứ tự.

```
line_a = Line(start1, end)
line_b = Line(start2, end)
line_a == line_b    # nên là True
```

Ví dụ này nên là `True` vì các đối tượng `Line` tham chiếu đến các đối tượng `Point` tương đương, theo thứ tự ngược lại.

```
line_c = Line(end, start1)
line_a == line_c     # nên là True
```

Sự tương đương luôn phải truyền đạt - nghĩa là, nếu `line_a` và `line_b` tương đương, và `line_a` và `line_c` tương đương, thì `line_b` và `line_c` cũng phải tương đương.

```
line_b == line_c     # nên là True
```

Ví dụ này nên là `False` vì các đối tượng `Line` tham chiếu đến các đối tượng `Point` không tương đương.

```
line_d = Line(start1, start2)
line_a == line_d    # nên là False
```

Bài tập 2

Viết một phương thức `Line` có tên `midpoint` tính toán điểm giữa của một đoạn thẳng và trả về kết quả dưới dạng một đối tượng `Point`.

Bạn có thể sử dụng dàn ý sau để bắt đầu.

```
%%add_method_to Line

    def midpoint(self):
        return Point(0, 0)


%%add_method_to Line

    def midpoint(self):
        mid_x = (self.p1.x + self.p2.x) / 2
        mid_y = (self.p1.y + self.p2.y) / 2
        return Point(mid_x, mid_y)
```

Bạn có thể sử dụng các ví dụ sau để kiểm tra mã của mình và vẽ kết quả.

```
start = Point(0, 0)
end1 = Point(300, 0)
end2 = Point(0, 150)
line1 = Line(start, end1)
line2 = Line(start, end2)


mid1 = line1.midpoint()
print(mid1)


mid2 = line2.midpoint()
print(mid2)


line3 = Line(mid1, mid2)


make_turtle()

for shape in [line1, line2, line3]:
    shape.draw()
```

Bài tập 3

Viết một phương thức `Rectangle` có tên `midpoint` tìm điểm ở giữa của một hình chữ nhật và trả về kết quả dưới dạng một đối tượng `Point`.

Bạn có thể sử dụng dàn ý sau để bắt đầu.

```
%%add_method_to Rectangle

    def midpoint(self):
        return Point(0, 0)


%%add_method_to Rectangle

    def midpoint(self):
        mid_x = self.corner.x + self.width / 2
        mid_y = self.corner.y + self.height / 2
        return Point(mid_x, mid_y)
```

Bạn có thể sử dụng ví dụ sau để kiểm tra mã của mình.

```
corner = Point(30, 20)
rectangle = Rectangle(100, 80, corner)


mid = rectangle.midpoint()
print(mid)


diagonal = Line(corner, mid)


make_turtle()

for shape in [line1, line2, rectangle, diagonal]:
    shape.draw()
```

Bài tập 4

Viết một phương thức `Rectangle` có tên `make_cross` thực hiện:

1. Sử dụng `make_lines` để lấy danh sách các đối tượng `Line` đại diện cho bốn cạnh của hình chữ nhật.
2. Tính toán các điểm giữa của bốn đường thẳng.
3. Tạo và trả về một danh sách hai đối tượng `Line` đại diện cho các đường thẳng nối các điểm giữa đối diện, tạo thành một chữ thập ở giữa hình chữ nhật.

Bạn có thể sử dụng dàn ý sau để bắt đầu.

```
%%add_method_to Rectangle

    def make_diagonals(self):
        return []
```

```
%%add_method_to Rectangle

    def make_cross(self):
        midpoints = []

        for line in self.make_lines():
            midpoints.append(line.midpoint())

        p1, p2, p3, p4 = midpoints
        return Line(p1, p3), Line(p2, p4)
```

Bạn có thể sử dụng ví dụ sau để kiểm tra mã của mình.

```
corner = Point(30, 20)
rectangle = Rectangle(100, 80, corner)
```

```
lines = rectangle.make_cross()
```

```
make_turtle()

rectangle.draw()
for line in lines:
    line.draw()
```

Bài tập 5

Viết một định nghĩa cho một lớp có tên `Circle` với các thuộc tính `center` và `radius`, trong đó `center` là một đối tượng Point và `radius` là một số. Bao gồm các phương thức đặc biệt `__init__` và `__str__`, và một phương thức có tên `draw` sử dụng các hàm `jupyturtle` để vẽ hình tròn.

Bạn có thể sử dụng hàm sau, đây là một phiên bản của hàm `circle` mà chúng ta đã viết trong Chương 4.

```
from jupyturtle import make_turtle, forward, left, right
import math

def draw_circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    angle = 360 / n
    left(angle / 2)
    for i in range(n):
        forward(length)
        left(angle)
```

Bạn có thể sử dụng ví dụ sau để kiểm tra mã của mình. Chúng ta sẽ bắt đầu với một `Rectangle` hình vuông có chiều rộng và chiều cao là `100`.

```
corner = Point(20, 20)
rectangle = Rectangle(100, 100, corner)
```

Mã sau đây sẽ tạo một `Circle` vừa với hình vuông.

```
center = rectangle.midpoint()
radius = rectangle.height / 2

circle = Circle(center, radius)
print(circle)
```

Nếu mọi thứ hoạt động chính xác, mã sau đây sẽ vẽ hình tròn bên trong hình vuông (tiếp xúc với cả bốn cạnh).

```
make_turtle(delay=0.01)

rectangle.draw()
circle.draw()
```