```
# 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');

import thinkpython
```

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

Tính kế thừa

Tính năng ngôn ngữ thường liên quan nhất đến lập trình hướng đối tượng là **tính kế thừa**. Tính kế thừa là khả năng định nghĩa một lớp mới là phiên bản đã được chỉnh sửa của một lớp hiện có. Trong chương này, tôi sẽ minh họa tính kế thừa bằng cách sử dụng các lớp đại diện cho các lá bài, bộ bài và các tay chơi bài poker. Nếu bạn không chơi poker, đừng lo — tôi sẽ giải thích những gì bạn cần biết.

Biểu diễn các lá bài

Có 52 lá bài trong một bộ bài tiêu chuẩn — mỗi lá thuộc về một trong bốn chất và một trong mười ba hạng. Các chất bao gồm: Bích (Spades), Cơ (Hearts), Rô (Diamonds) và Chuồn (Clubs). Các hạng bao gồm: Át (Ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, J (Jack), Q (Queen) và K (King). Tùy vào trò chơi, Át có thể lớn hơn K hoặc nhỏ hơn 2.

Nếu muốn định nghĩa một đối tượng mới để đại diện cho một lá bài, việc chọn các thuộc tính hiển nhiên là: hạng và chất. Tuy nhiên, loại dữ liệu của các thuộc tính này không dễ xác định. Một cách là sử dụng chuỗi, ví dụ 'Spade' cho chất và 'Queen' cho hạng. Tuy nhiên, nhược điểm của cách này là khó so sánh các lá bài để xác định lá nào lớn hơn về hạng hoặc chất.

Một cách thay thế là sử dụng số nguyên để mã hóa các hạng và chất. Trong ngữ cảnh này, “mã hóa” nghĩa là định nghĩa một ánh xạ giữa số và chất, hoặc giữa số và hạng. Dạng mã hóa này không nhằm mục đích giữ bí mật (đó là “mã hóa dữ liệu”).

Ví dụ, bảng sau đây thể hiện các chất và mã số nguyên tương ứng:

Chất	Mã số
Spades	3
Hearts	2
Diamonds	1
Clubs	0

Với cách mã hóa này, chúng ta có thể so sánh các chất bài bằng cách so sánh các mã số của chúng.

Để mã hóa các hạng bài, chúng ta sẽ sử dụng số nguyên `2` để biểu thị hạng bài `2`, `3` để biểu thị `3`, và cứ tiếp tục như vậy đến `10`. Bảng sau đây cho thấy các mã dành cho các lá bài hình.

Hạng	Mã số
Jack	11
Queen	12
King	13

Và chúng ta có thể sử dụng `1` hoặc `14` để biểu thị lá Át, tùy thuộc vào việc chúng ta muốn nó được coi là thấp hơn hay cao hơn so với các hạng bài khác.

Để biểu diễn các mã hóa này, chúng ta sẽ sử dụng hai danh sách chuỗi, một danh sách chứa tên các chất bài và danh sách còn lại chứa tên các hạng bài.

Dưới đây là một định nghĩa cho một lớp đại diện cho lá bài, với các danh sách chuỗi này được sử dụng làm **biến lớp**, là các biến được định nghĩa bên trong định nghĩa lớp nhưng không nằm trong một phương thức.

```
class Card:
    """Biểu diễn bộ bài tiêu chuẩn."""

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']
```

Phần tử đầu tiên của `rank_names` là `None` vì không có lá bài nào có hạng là 0. Bằng cách đưa `None` vào làm chỗ giữ chỗ, chúng ta có được một danh sách với tính chất hữu ích là chỉ số `2` ánh xạ đến chuỗi `'2'`, và tương tự như vậy.

Biến lớp được liên kết với lớp thay vì một thực thể của lớp, vì vậy chúng ta có thể truy cập chúng như sau.

```
Card.suit_names
```

```
['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

Chúng ta có thể sử dụng `suit_names` để tra cứu một chất bài và nhận được chuỗi tương ứng.

```
Card.suit_names[0]
```

```
'Clubs'
```

Và `rank_names` để tra cứu một giá trị hạng.

```
Card.rank_names[11]
```

```
'Jack'
```

Các thuộc tính của lá bài

Đây là một phương thức `__init__` cho lớp `Card` — nó nhận `suit` và `rank` làm tham số và gán chúng cho các thuộc tính có tên giống nhau.

```
%%add_method_to Card

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

Bây giờ chúng ta có thể tạo một đối tượng `Card` như thế này.

```
queen = Card(1, 12)
```

Chúng ta có thể sử dụng thực thể mới để truy cập các thuộc tính.

```
queen.suit, queen.rank
```

```
(1, 12)
```

Việc sử dụng thực thể để truy cập các biến lớp cũng là hợp lệ

```
queen.suit_names
```

```
['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

Nhưng nếu bạn sử dụng lớp, sẽ rõ ràng hơn rằng chúng là biến lớp, chứ không phải thuộc tính.

In các lá bài

Dưới đây là phương thức `__str__` cho các đối tượng `Card`.

```
%%add_method_to Card

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'
```

Khi chúng ta in một đối tượng `Card`, Python gọi phương thức `__str__` để lấy một đại diện dễ đọc của lá bài.

```
print(queen)
```

```
Queen of Diamonds
```

Dưới đây là một sơ đồ của đối tượng lớp `Card` và thể hiện đối tượng `Card`. `Card` là một đối tượng lớp, vì vậy kiểu của nó là `type`. `queen` là một thể hiện của lớp `Card`, vì vậy kiểu của nó là `Card`. Để tiết kiệm không gian, tôi không vẽ nội dung của `suit_names` và `rank_names`.

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

bindings = [Binding(Value(name), draw_value=False)
            for name in ['suit_names', 'rank_names']]

frame1 = Frame(bindings, name='type', dy=-0.5, offsetx=0.77)
binding1 = Binding(Value('Card'), frame1)

bindings = [Binding(Value(name), Value(value))
            for name, value in zip(['suit', 'rank'], [1, 11])]

frame2 = Frame(bindings, name='Card', dy=-0.3, offsetx=0.33)
binding2 = Binding(Value('queen'), frame2)

stack = Stack([binding1, binding2], dy=-1.2)
```

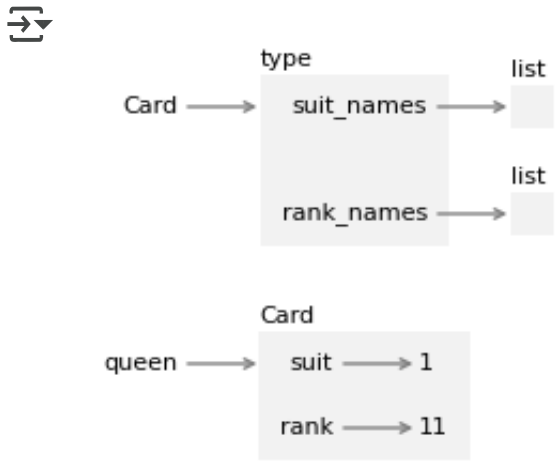
```
from diagram import diagram, Bbox, make_list, adjust

width, height, x, y = [2.11, 2.14, 0.35, 1.76]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)

value = make_list([])
bbox2 = value.draw(ax, x+1.66, y)

value = make_list([])
bbox3 = value.draw(ax, x+1.66, y-0.5)

bbox = Bbox.union([bbox, bbox2, bbox3])
#adjust(x, y, bbox)
```



Mỗi thực thể của `Card` đều có các thuộc tính `suit` và `rank` riêng, nhưng chỉ có một đối tượng lớp `Card` và chỉ có một bản sao duy nhất của các biến lớp `suit_names` và `rank_names`.

So sánh các lá bài

Giả sử chúng ta tạo một đối tượng `Card` thứ hai với cùng chất và hạng.

```
queen2 = Card(1, 12)
print(queen2)
```

```
Queen of Diamonds
```

Nếu chúng ta sử dụng toán tử `==` để so sánh chúng, nó sẽ kiểm tra xem `queen` và `queen2` có tham chiếu đến cùng một đối tượng hay không.

```
queen == queen2
```

```
False
```

Chúng không tham chiếu đến cùng một đối tượng, vì vậy nó trả về `False`. Chúng ta có thể thay đổi hành vi này bằng cách định nghĩa phương thức đặc biệt `__eq__`.

```
%%add_method_to Card

    def __eq__(self, other):
        return self.suit == other.suit and self.rank == other.rank
```

`__eq__` nhận hai đối tượng `Card` làm tham số và trả về `True` nếu chúng có cùng chất và hạng, ngay cả khi chúng không phải là cùng một đối tượng. Nói cách khác, nó kiểm tra xem chúng có tương đương hay không, mặc dù chúng không phải là bản sao giống hệt nhau.

Khi chúng ta sử dụng toán tử `==` với các đối tượng `Card`, Python sẽ gọi phương thức `__eq__`.

```
queen == queen2
```

```
True
```

Như một bài kiểm tra thứ hai, chúng ta hãy tạo một lá bài với cùng một chất nhưng có hạng khác.

```
six = Card(1, 6)
print(six)
```

```
6 of Diamonds
```

Chúng ta có thể xác nhận rằng `queen` và `six` không phải là tương đương.

```
queen == six
```

```
False
```

Nếu chúng ta sử dụng toán tử `!=`, Python sẽ gọi một phương thức đặc biệt có tên là `__ne__`, nếu phương thức này tồn tại. Nếu không, Python sẽ gọi phương thức `__eq__` và đảo ngược kết quả — vì vậy, nếu `__eq__` trả về `True`, kết quả của toán tử `!=` sẽ là `False`.

```
queen != queen2
```

```
False
```

```
queen != six
```

```
True
```

Bây giờ giả sử chúng ta muốn so sánh hai lá bài để xem lá nào lớn hơn. Nếu chúng ta sử dụng một trong các toán tử quan hệ, chúng ta sẽ gặp lỗi `TypeError`.

```
%%expect TypeError
```

```
queen < queen2
```

```
TypeError: '<' not supported between instances of 'Card' and 'Card'
```

Để thay đổi hành vi của toán tử `<`, chúng ta có thể định nghĩa một phương thức đặc biệt gọi là `__lt__`, viết tắt của "less than" (nhỏ hơn). Vì mục đích ví dụ này, giả sử rằng cùng chất (suit) quan trọng hơn hạng (rank) — vì vậy tất cả các lá bài Bích (Spades) có giá trị cao hơn tất cả các lá bài Cơ (Hearts), Cơ cao hơn các lá bài Rô (Diamonds), và cứ tiếp tục như vậy. Nếu hai lá bài có cùng chất, lá bài có hạng cao hơn sẽ thắng.

Để triển khai logic này, chúng ta sẽ sử dụng phương thức sau, trả về một bộ chứa chất liệu và hạng của lá bài theo thứ tự đó.

```
%%add_method_to Card
```

```
    def to_tuple(self):
        return (self.suit, self.rank)
```

Chúng ta có thể sử dụng phương thức này để viết phương thức `__lt__`.

```
%%add_method_to Card

    def __lt__(self, other):
        return self.to_tuple() < other.to_tuple()
```

So sánh tuple sẽ so sánh các phần tử đầu tiên của mỗi tuple, đại diện cho bộ bài. Nếu chúng giống nhau, nó sẽ so sánh các phần tử thứ hai, đại diện cho giá trị của lá bài.

Bây giờ, nếu chúng ta sử dụng toán tử `<`, Python sẽ gọi phương thức `__lt__`.

```
six < queen
```

```
True
```

Nếu chúng ta sử dụng toán tử `>`, Python sẽ gọi một phương thức đặc biệt có tên là `__gt__`, nếu phương thức này tồn tại. Nếu không, Python sẽ gọi phương thức `__lt__` với các đối số theo thứ tự ngược lại.

```
queen < queen2
```

```
False
```

```
queen > queen2
```

```
False
```

Cuối cùng, nếu chúng ta sử dụng toán tử `<=`, Python sẽ gọi một phương thức đặc biệt có tên là `__le__`.

```
%%add_method_to Card

    def __le__(self, other):
        return self.to_tuple() <= other.to_tuple()
```

Vậy là chúng ta có thể kiểm tra xem một lá bài có nhỏ hơn hoặc bằng một lá bài khác hay không.

```
queen <= queen2
```

```
True
```

```
queen <= six
```

```
False
```

Nếu chúng ta sử dụng toán tử `>=`, Python sẽ gọi phương thức `__ge__` nếu nó tồn tại. Nếu không, nó sẽ gọi phương thức `__le__` với các đối số đảo ngược lại.

```
queen >= six
```

Như chúng ta đã định nghĩa, các phương thức này là đầy đủ ở chỗ chúng ta có thể so sánh bất kỳ hai đối tượng `Card` nào, và nhất quán ở chỗ kết quả từ các toán tử khác nhau không mâu thuẫn với nhau. Với hai tính chất này, chúng ta có thể nói rằng các đối tượng `Card` là **được sắp xếp hoàn toàn**. Và điều này có nghĩa là, như chúng ta sẽ thấy sớm, chúng có thể được sắp xếp.

Bộ bài

Bây giờ chúng ta đã có các đối tượng đại diện cho các lá bài, hãy định nghĩa các đối tượng đại diện cho bộ bài. Dưới đây là định nghĩa lớp `Deck` với phương thức `__init__` nhận một danh sách các đối tượng Card làm tham số và gán nó vào một thuộc tính có tên là `cards`.

```
class Deck:

    def __init__(self, cards):
        self.cards = cards
```

Để tạo một danh sách chứa 52 lá bài trong một bộ bài chuẩn, chúng ta sẽ sử dụng phương thức tĩnh sau đây.

```
%%add_method_to Deck

    def make_cards():
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards
```

Trong phương thức `make_cards`, vòng lặp ngoài lặp qua các chất bài từ `0` đến `3`. Vòng lặp trong lặp qua các hạng bài từ `2` đến `14` — trong đó `14` đại diện cho lá Ace có giá trị cao hơn K. Mỗi lần lặp sẽ tạo ra một đối tượng `Card` với chất và hạng bài hiện tại, và thêm nó vào danh sách `cards`.

Dưới đây là cách chúng ta tạo một danh sách các lá bài và một đối tượng `Deck` chứa danh sách đó.

```
cards = Deck.make_cards()
deck = Deck(cards)
len(deck.cards)
```

```
52
```

Nó chứa 52 lá bài, như mong muốn.

In bộ bài

Dưới đây là phương thức `__str__` cho lớp `Deck`.

```
%%add_method_to Deck

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Phương thức này minh họa một cách hiệu quả để kết hợp một chuỗi lớn — xây dựng một danh sách các chuỗi và sau đó sử dụng phương thức chuỗi `join`.

Chúng ta sẽ kiểm tra phương thức này với một bộ bài chỉ chứa hai lá bài.

```
small_deck = Deck([queen, six])
```

Nếu chúng ta gọi `str`, nó sẽ gọi phương thức `__str__`.

```
str(small_deck)
```

```
'Queen of Diamonds\n6 of Diamonds'
```

Khi Jupyter hiển thị một chuỗi, nó hiển thị dạng 'đại diện của chuỗi, trong đó ký tự xuống dòng được biểu diễn bằng chuỗi `\n`.

Tuy nhiên, nếu chúng ta in kết quả, Jupyter sẽ hiển thị dạng 'có thể in' của chuỗi, trong đó ký tự xuống dòng được in ra dưới dạng khoảng trắng.

```
print(small_deck)
```

```
Queen of Diamonds
6 of Diamonds
```

Vì vậy, các lá bài xuất hiện trên các dòng riêng biệt.

Thêm, xoá, xào bài, và sắp xếp

Để chia bài, chúng ta muốn một phương thức loại bỏ một lá bài khỏi bộ bài và trả lại nó. Phương thức `pop` của danh sách cung cấp một cách tiện lợi để làm điều đó.

```
%%add_method_to Deck

    def take_card(self):
        return self.cards.pop()
```

Đây là cách chúng ta sử dụng nó.

```
card = deck.take_card()
print(card)
```

```
Ace of Spades
```

Chúng ta có thể xác nhận rằng còn lại 51 lá bài trong bộ bài.

```
len(deck.cards)
```

```
51
```

Để thêm một lá bài, chúng ta có thể sử dụng phương thức `append` của danh sách.

```
%%add_method_to Deck

    def put_card(self, card):
        self.cards.append(card)
```

Ví dụ, chúng ta có thể đặt lại lá bài mà chúng ta vừa rút ra.

```
deck.put_card(card)
len(deck.cards)
```

```
52
```

Để xáo trộn bộ bài, chúng ta có thể sử dụng hàm `shuffle` từ mô-đun `random`:

```
import random


# Ô này khởi tạo bộ sinh số ngẫu nhiên để chúng ta
# luôn nhận được kết quả giống nhau.

random.seed(3)
```

```
%%add_method_to Deck

    def shuffle(self):
        random.shuffle(self.cards)
```

Nếu chúng ta xào bài và in ra một vài lá bài đầu tiên, ta có thể thấy chúng không theo bất kỳ thứ tự nào rõ ràng.

```
deck.shuffle()
for card in deck.cards[:4]:
    print(card)
```

```
10 of Clubs
10 of Hearts
5 of Diamonds
9 of Hearts
```

Để sắp xếp các lá bài, chúng ta có thể sử dụng phương thức `sort` của danh sách, phương thức này sắp xếp các phần tử 'tại chỗ' – tức là nó sửa đổi danh sách thay vì tạo ra một danh sách mới.

```
%%add_method_to Deck

    def sort(self):
        self.cards.sort()
```

Khi chúng ta gọi `sort`, nó sẽ sử dụng phương thức `__lt__` để so sánh các lá bài.

```
deck.sort()
```

Nếu chúng ta in ra vài lá bài đầu tiên, chúng ta có thể xác nhận rằng chúng đang ở trong thứ tự tăng dần.

```
for card in deck.cards[:4]:
    print(card)
```

```
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
```

Trong ví dụ này, `Deck.sort` không làm gì ngoài việc gọi `list.sort`. Việc chuyển giao trách nhiệm như thế này được gọi là **ủy thác**.

Lớp cha và lớp con

Tính kế thừa là khả năng định nghĩa một lớp mới là phiên bản sửa đổi của một lớp đã tồn tại. Lấy ví dụ, giả sử chúng ta muốn một lớp để đại diện cho "bài", tức là các lá bài mà một người chơi nắm giữ.

- Một bộ bài giống như một bộ sưu tập bài – cả hai đều được tạo thành từ một tập hợp các lá bài, và cả hai đều yêu cầu các thao tác như thêm và xóa bài.
- Một bộ bài cũng khác với một bộ bài – có những thao tác mà chúng ta muốn sử dụng cho bộ bài mà không hợp lý với bộ bài. Ví dụ, trong poker, chúng ta có thể so sánh hai bộ bài để xem bộ nào thắng. Trong bridge, chúng ta có thể tính điểm cho một bộ bài để đưa ra mức đặt cược.

Mối quan hệ giữa các lớp này – nơi mà một lớp là phiên bản chuyên biệt của một lớp khác – rất phù hợp với kế thừa.

Để định nghĩa một lớp mới dựa trên một lớp đã tồn tại, chúng ta đặt tên của lớp đã tồn tại vào dấu ngoặc đơn.

```
class Hand(Deck):
    """Biểu diễn một bộ bài."""
```

Định nghĩa này chỉ ra rằng `Hand` kế thừa từ `Deck`, điều này có nghĩa là các đối tượng `Hand` có thể truy cập các phương thức được định nghĩa trong `Deck`, như `take_card` và `put_card`.

`Hand` cũng kế thừa phương thức `__init__` từ `Deck`, nhưng nếu chúng ta định nghĩa `__init__` trong lớp `Hand`, nó sẽ ghi đè phương thức `__init__` trong lớp `Deck`.

```
%%add_method_to Hand

    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

Phiên bản này của `__init__` nhận một chuỗi tùy chọn làm tham số và luôn bắt đầu với một danh sách thẻ rỗng. Khi chúng ta tạo một `Hand`, Python sẽ gọi phương thức này, chứ không phải phương thức trong `Deck` — điều này có thể xác nhận bằng cách kiểm tra xem kết quả có thuộc tính `label` hay không.

```
hand = Hand('player 1')
hand.label
```

```
'player 1'
```

Để chia bài, chúng ta có thể sử dụng `take_card` để loại bỏ một lá bài từ một `Deck`, và `put_card` để thêm lá bài vào một `Hand`.

```
deck = Deck(cards)
card = deck.take_card()
hand.put_card(card)
print(hand)
```

```
Ace of Spades
```

Hãy đóng gói mã này trong một phương thức `Deck` gọi là `move_cards`.

```
%%add_method_to Deck

    def move_cards(self, other, num):
        for i in range(num):
            card = self.take_card()
            other.put_card(card)
```

Phương thức này là đa hình — tức là, nó hoạt động với nhiều loại đối tượng: `self` và `other` có thể là một `Hand` hoặc một `Deck`. Vì vậy, chúng ta có thể sử dụng phương thức này để chia bài từ `Deck` sang `Hand`, từ một `Hand` này sang một `Hand` khác, hoặc từ một `Hand` quay lại `Deck`.

Khi một lớp mới kế thừa từ một lớp đã có, lớp đã có được gọi là **lớp cha** và lớp mới được gọi là **lớp con**. Nói chung:

- Các đối tượng của lớp con nên có tất cả các thuộc tính của lớp cha, nhưng chúng có thể có các thuộc tính bổ sung.
- Lớp con nên có tất cả các phương thức của lớp cha, nhưng nó có thể có các phương thức bổ sung.
- Nếu lớp con ghi đè một phương thức từ lớp cha, phương thức mới nên nhận các tham số giống nhau và trả về kết quả tương thích.

Tập hợp các quy tắc này được gọi là "nguyên lý thay thế Liskov" theo tên nhà khoa học máy tính Barbara Liskov.

Nếu bạn tuân thủ những quy tắc này, bất kỳ hàm hoặc phương thức nào được thiết kế để làm việc với một thể hiện của lớp cha, như `Deck`, cũng sẽ hoạt động với các thể hiện của lớp con, như `Hand`. Nếu bạn vi phạm những quy tắc này, mã của bạn sẽ sụp đổ như một ngôi nhà bài (xin lỗi).

Chuyên biệt hóa

Hãy tạo một lớp gọi là `BridgeHand`, đại diện cho một bộ bài trong trò chơi bridge — một trò chơi bài được chơi phổ biến. Chúng ta sẽ kế thừa từ `Hand` và thêm một phương thức mới gọi là `high_card_point_count`, phương thức này đánh giá một bộ bài bằng cách sử dụng phương pháp "điểm cao", cộng điểm cho các lá bài cao trong bộ bài.

Dưới đây là định nghĩa lớp chứa một biến lớp là một từ điển ánh xạ từ tên các lá bài tới giá trị điểm của chúng.

```
class BridgeHand(Hand):
    """Biểu diễn một bộ bài bridge."""

    hcp_dict = {
        'Ace': 4,
        'King': 3,
        'Queen': 2,
        'Jack': 1,
    }
```

Giải thích thêm từ dịch giả

Bộ bài Bridge là một bộ bài gồm 52 lá, được sử dụng trong trò chơi bài Bridge. Bridge là một trò chơi bài đối kháng, thường chơi với 4 người, chia thành 2 đội, mỗi đội gồm 2 người ngồi đối diện nhau.

Với hạng của một lá bài, như `12`, chúng ta có thể sử dụng `Card.rank_names` để lấy dạng chuỗi đại diện cho cấp bậc đó, và sau đó sử dụng `hcp_dict` để lấy điểm số tương ứng.

```
rank = 12
rank_name = Card.rank_names[rank]
score = BridgeHand.hcp_dict.get(rank_name, 0)
rank_name, score
```

```
('Queen', 2)
```

Phương thức sau đây duyệt qua các lá bài trong một `BridgeHand` và cộng dồn điểm số của chúng.

```
%%add_method_to BridgeHand

    def high_card_point_count(self):
        count = 0
        for card in self.cards:
            rank_name = Card.rank_names[card.rank]
            count += BridgeHand.hcp_dict.get(rank_name, 0)
        return count
```

```
# Ô này tạo một bộ bài mới và
# khởi tạo bộ tạo số ngẫu nhiên

cards = Deck.make_cards()
deck = Deck(cards)
random.seed(3)
```

Để kiểm tra, chúng ta sẽ chia một bộ bài gồm năm lá bài — một bộ bài bridge thường có mười ba lá, nhưng việc kiểm tra mã với các ví dụ nhỏ sẽ dễ dàng hơn.

```
hand = BridgeHand('player 2')

deck.shuffle()
deck.move_cards(hand, 5)
print(hand)
```

```
4 of Diamonds
King of Hearts
10 of Hearts
10 of Clubs
Queen of Diamonds
```

Và đây là tổng điểm cho lá K và lá Q.

```
hand.high_card_point_count()
```

```
5
```

`BridgeHand` kế thừa các biến và phương thức của `Hand` và thêm một biến lớp cùng một phương thức cụ thể dành riêng cho trò chơi bridge. Cách sử dụng kế thừa này được gọi là chuyên biệt hóa vì nó định nghĩa một lớp mới chuyên biệt cho một mục đích cụ thể, như chơi bridge.

Gỡ lỗi

Tính kế thừa là một tính năng hữu ích. Một số chương trình vốn sẽ lặp đi lặp lại nếu không có kế thừa có thể được viết ngắn gọn hơn nhờ nó. Ngoài ra, kế thừa có thể giúp tái sử dụng mã nguồn, vì bạn có thể tùy chỉnh hành vi của một lớp cha mà không cần phải sửa đổi nó. Trong một số trường hợp, cấu trúc kế thừa phản ánh cấu trúc tự nhiên của vấn đề, giúp thiết kế dễ hiểu hơn.

Mặt khác, kế thừa có thể làm cho chương trình trở nên khó đọc. Khi một phương thức được gọi, đôi khi không rõ cần tìm định nghĩa của nó ở đâu — đoạn mã liên quan có thể được trải dài qua nhiều mô-đun khác nhau.

Bất cứ khi nào bạn không chắc chắn về luồng thực thi trong chương trình của mình, giải pháp đơn giản nhất là thêm các câu lệnh `print` vào đầu các phương thức liên quan. Nếu `Deck.shuffle` in ra một thông báo như Đang chạy `Deck.shuffle`, thì khi chương trình chạy, nó sẽ vạch ra luồng thực thi.

Ngoài ra, bạn có thể sử dụng hàm sau đây, hàm này nhận một đối tượng và tên một phương thức (dưới dạng chuỗi) rồi trả về lớp cung cấp định nghĩa của phương thức đó.

```
def find_defining_class(obj, method_name):
    """Tìm lớp mà phương thức đã cho được định nghĩa."""
    for typ in type(obj).mro():
        if method_name in vars(typ):
            return typ
    return f'Method {method_name} not found.'
```

Hàm `find_defining_class` sử dụng phương thức `mro` để lấy danh sách các đối tượng lớp (loại) sẽ được tìm kiếm để tìm phương thức. "MRO" là viết tắt của "method resolution order" (thứ tự giải quyết phương thức), là trình tự các lớp mà Python tìm kiếm để "giải quyết" tên phương thức — tức là để tìm đối tượng hàm mà tên đó tham chiếu đến.

Ví dụ, hãy tạo một đối tượng `BridgeHand` và sau đó tìm lớp định nghĩa của phương thức `shuffle`.

```
hand = BridgeHand('player 3')
find_defining_class(hand, 'shuffle')
```

```
Deck
def __init__(cards)
```

```
<no docstring>
```

Phương thức `shuffle` cho đối tượng `BridgeHand` là phương thức trong lớp `Deck`.

Thuật ngữ

- **inheritance - tính kế thừa**: Khả năng định nghĩa một lớp mới là phiên bản đã được chỉnh sửa của một lớp đã được định nghĩa trước đó.
- **encode - mã hóa**: Biểu diễn một tập hợp giá trị bằng một tập hợp giá trị khác bằng cách xây dựng một ánh xạ giữa chúng.
- **class variable - biến lớp**: Một biến được định nghĩa trong phần định nghĩa lớp, nhưng không nằm trong bất kỳ phương thức nào.
- **totally ordered - được sắp xếp hoàn toàn**: Một tập hợp các đối tượng được sắp xếp hoàn toàn nếu chúng ta có thể so sánh bất kỳ hai phần tử nào và kết quả là nhất quán.
- **delegation - ủy thác**: Khi một phương thức chuyển trách nhiệm cho một phương thức khác để thực hiện phần lớn hoặc tất cả công việc.
- **parent class - lớp cha**: Một lớp mà từ đó lớp khác kế thừa.
- **child class - lớp con**: Một lớp kế thừa từ một lớp khác.
- **specialization - chuyên biệt hóa**: Một cách sử dụng kế thừa để tạo ra một lớp mới là phiên bản chuyên biệt của một lớp đã có.

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 trợ lý ảo

Khi thực hiện tốt, lập trình hướng đối tượng có thể làm cho chương trình dễ đọc, dễ kiểm tra và tái sử dụng hơn. Tuy nhiên, nó cũng có thể khiến chương trình trở nên phức tạp và khó bảo trì. Do đó, OOP là một chủ đề gây tranh cãi — một số người rất yêu thích nó, trong khi những người khác thì không.

Để tìm hiểu thêm về chủ đề này, bạn có thể hỏi trợ lý ảo:

- Một số ưu và nhược điểm của lập trình hướng đối tượng là gì?
- Câu nói “ưu tiên sử dụng tính kết hợp thay vì tính kế thừa” có nghĩa là gì?
- Nguyên tắc thay thế Liskov là gì?
- Python có phải là một ngôn ngữ lập trình hướng đối tượng không?
- Yêu cầu để một tập hợp được coi là được sắp xếp hoàn toàn là gì?

Và như mọi khi, hãy cân nhắc sử dụng trợ lý ảo để hỗ trợ bạn với các bài tập liên quan.

Bài tập 1

Trong trò chơi bài bridge, một “trick” là một lượt chơi mà trong đó mỗi người trong bốn người chơi đánh ra một lá bài. Để đại diện cho những lá bài đó, chúng ta sẽ định nghĩa một lớp kế thừa từ lớp `Deck`.

```
class Trick(Deck):
    """Biểu diễn một ván bài trong trò chơi bài bridge"""
```

Ví dụ, hãy xem xét một lượt chơi sau: người chơi đầu tiên đánh lá 3 Bích, điều này có nghĩa là Bích là chất dẫn đầu. Người chơi thứ hai và thứ ba theo chất, nghĩa là họ đánh một lá bài có cùng chất dẫn đầu. Người chơi thứ tư đánh một lá bài khác chất, điều đó có nghĩa là họ không thể thắng trong lượt chơi này. Do đó, người thắng trong lượt chơi này là người chơi thứ ba, vì họ đã đánh lá bài cao nhất trong chất dẫn đầu.

```
cards = [Card(1, 3),
         Card(1, 10),
         Card(1, 12),
         Card(2, 13)]
trick = Trick(cards)
print(trick)
```

Viết một phương thức `Trick` có tên là `find_winner` để lặp qua các lá bài trong `Trick` và trả về chỉ số của lá bài chiến thắng. Trong ví dụ trước, chỉ số của lá bài chiến thắng là `2`.

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

```
%%add_method_to Trick

    def find_winner(self):
        return 0


# Đáp án được viết ở đây
```

Nếu bạn kiểm tra phương thức của mình với ví dụ trước, chỉ số của lá bài chiến thắng sẽ là `2`.

```
trick.find_winner()
```

Bài tập 2

Các bài tập tiếp theo yêu cầu bạn viết các hàm phân loại các bộ bài poker. Nếu bạn không quen thuộc với poker, tôi sẽ giải thích những gì bạn cần biết. Chúng ta sẽ sử dụng lớp sau để đại diện cho các bộ bài poker.

```
class PokerHand(Hand):
    """Biểu diễn một bộ bài poker."""

    def get_suit_counts(self):
        counter = {}
        for card in self.cards:
            key = card.suit
            counter[key] = counter.get(key, 0) + 1
        return counter

    def get_rank_counts(self):
        counter = {}
        for card in self.cards:
            key = card.rank
            counter[key] = counter.get(key, 0) + 1
        return counter
```

`PokerHand` cung cấp hai phương thức sẽ giúp bạn trong các bài tập.

- `get_suit_counts` lặp qua các lá bài trong `PokerHand`, đếm số lượng bài trong mỗi chất và trả về một từ điển ánh xạ từ mã chất đến số lần nó xuất hiện.
- `get_rank_counts` làm điều tương tự với các hạng của các lá bài, trả về một từ điển ánh xạ từ mã hạng đến số lần nó xuất hiện.

Tất cả các bài tập tiếp theo đều có thể thực hiện bằng cách sử dụng chỉ các tính năng Python mà chúng ta đã học cho đến nay, nhưng một số bài tập khó hơn so với hầu hết các bài tập trước đây. Tôi khuyến khích bạn yêu cầu sự trợ giúp từ AI.

Đối với những bài toán như thế này, thường thì yêu cầu lời khuyên chung về chiến lược và thuật toán sẽ rất hữu ích. Sau đó, bạn có thể tự viết mã hoặc yêu cầu mã. Nếu bạn yêu cầu mã, bạn có thể muốn cung cấp các định nghĩa lớp liên quan như một phần của yêu cầu.

Bài tập đầu tiên, chúng ta sẽ viết một phương thức gọi là `has_flush` để kiểm tra xem bộ bài có phải là “flush” hay không – tức là, xem nó có ít nhất năm lá bài cùng chất hay không.

Trong hầu hết các loại poker, một bộ bài chứa năm hoặc bảy lá bài, nhưng có một số biến thể kỳ lạ trong đó bộ bài có số lượng lá bài khác. Tuy nhiên, bất kể số lá bài trong bộ bài là bao nhiêu, chỉ có năm lá bài tạo thành bộ bài mạnh nhất mới được tính.

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

```
%%add_method_to PokerHand

    def has_flush(self):
        """Kiểm tra xem bộ bài này có phải là một flush không."""
        return False
```

Để kiểm tra phương thức này, chúng ta sẽ tạo một bộ bài với năm lá bài đều là Cơ, vì vậy nó sẽ có một "flush".

```
good_hand = PokerHand('good_hand')

suit = 0
for rank in range(10, 15):
    card = Card(suit, rank)
    good_hand.put_card(card)

print(good_hand)
```

Nếu chúng ta gọi phương thức `get_suit_counts`, chúng ta có thể xác nhận rằng mã hạng `0` xuất hiện `5` lần.

```
good_hand.get_suit_counts()
```

ậy nên, phương thức `has_flush` sẽ trả về `True`.

```
good_hand.has_flush()
```

Trong bài kiểm tra thứ hai, chúng ta sẽ tạo một bộ bài với ba lá Cơ và hai chất khác.

```
cards = [Card(0, 2),
         Card(0, 3),
         Card(2, 4),
         Card(3, 5),
         Card(0, 7),
        ]

bad_hand = PokerHand('bad hand')
for card in cards:
    bad_hand.put_card(card)

print(bad_hand)
```

Vậy nên, phương thức `has_flush` sẽ trả về `False`.

```
bad_hand.has_flush()
```

Bài tập 3

Viết một phương thức có tên là `has_straight` để kiểm tra xem bộ bài có chứa một sảnh rồng hay không, tức là một tập hợp năm lá bài có hạng liên tiếp. Ví dụ, nếu bộ bài chứa các hạng `5`, `6`, `7`, `8`, và `9`, thì đó là một sảnh rồng.

Một lá Át có thể đứng trước một lá 2 hoặc sau một lá Già (K), vì vậy `Át, 2, 3, 4, 5` là một sảnh rồng và `10, Bồi, Đầm, Già, Át` cũng vậy. Tuy nhiên, một sảnh rồng không thể "quay vòng", vì vậy `Già, Át, 2, 3, 4` không phải là một sảnh rồng.

Bạn có thể sử dụng dàn ý sau để bắt đầu. Nó bao gồm một vài dòng mã đếm số lá `Át` – được biểu diễn bằng mã `1` hoặc `14` – và lưu tổng vào cả hai vị trí của bộ đếm.

```
%%add_method_to PokerHand

    def has_straight(self, n=5):
        """Kiểm tra xem bộ bài này có phải là một chuỗi liên tiếp với ít nhất n l
        counter = self.get_rank_counts()
        aces = counter.get(1, 0) + counter.get(14, 0)
        counter[1] = aces
        counter[14] = aces

        return False
```

`good_hand`, mà chúng ta đã tạo ra cho bài tập trước, chứa một sảnh rồng. Nếu chúng ta sử dụng phương thức `get_rank_counts`, chúng ta có thể xác nhận rằng bộ bài có ít nhất một lá bài với mỗi hạng liên tiếp trong năm hạng.

```
good_hand.get_rank_counts()
```

Vậy nên, phương thức `has_straight` trả về `True`.

```
good_hand.has_straight()
```

`bad_hand` không chứa một sảnh rồng, vì vậy phương thức `has_straight` sẽ trả về `False`.

```
bad_hand.has_straight()
```

Bài tập 4

Một bộ bài có một sảnh rồng nếu nó chứa một tập hợp năm lá bài vừa là một sảnh rồng vừa là một chất — tức là năm lá bài cùng chất với các hạng liên tiếp. Viết một phương thức `PokerHand` để kiểm tra xem bộ bài có sảnh rồng cùng chất hay không.

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

```
%%add_method_to PokerHand

    def has_straightflush(self):
        """Kiểm tra xem bộ bài này có phải là một sảnh rồng đồng chất không"""
        return False
```

Sử dụng các ví dụ sau để kiểm tra phương thức của bạn.

```
good_hand.has_straightflush()     # nên trả về True
```

```
bad_hand.has_straightflush()     # nên trả về False
```

Lưu ý rằng không chỉ cần kiểm tra xem bộ bài có một sảnh rồng và cùng chất hay không. Để hiểu lý do, hãy xem xét bộ bài sau.

```
from copy import deepcopy

straight_and_flush = deepcopy(bad_hand)
straight_and_flush.put_card(Card(0, 6))
straight_and_flush.put_card(Card(0, 9))
print(straight_and_flush)
```

Bộ bài này chứa một sảnh rồng và cùng chất, nhưng chúng không phải là năm lá bài giống nhau.

```
 straight_and_flush.has_straight(), straight_and_flush.has_flush()
```

Vì vậy, bộ bài này không chứa một sảnh rồng cùng chất.

```
straight_and_flush.has_straightflush()    # nên trả về True
```

Bài tập 5

Một bộ bài poker có một đôi nếu nó chứa hai hoặc nhiều lá bài có cùng hạng. Hãy viết một phương thức `PokerHand` để kiểm tra xem bộ bài có chứa một đôi hay không.

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

```
%%add_method_to PokerHand

    def check_sets(self, *need_list):
        return True
```

Để kiểm tra phương thức của bạn, đây là một bộ bài có một đôi.

```
pair = deepcopy(bad_hand)
pair.put_card(Card(1, 2))
print(pair)
```

```
pair.has_pair()    # nên trả về True
```

```
bad_hand.has_pair()    # nên trả về False
```

```
good_hand.has_pair()   # nên trả về False
```

Bài tập 6

Một bộ bài có một "full house" nếu nó chứa ba lá bài cùng hạng và hai lá bài cùng hạng khác. Hãy viết một phương thức `PokerHand` để kiểm tra xem bộ bài có một "full house" hay không.

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

```
%%add_method_to PokerHand

    def has_full_house(self):
        return False
```

Bạn có thể sử dụng bộ bài này để kiểm tra phương thức của mình.

```
boat = deepcopy(pair)
boat.put_card(Card(2, 2))
boat.put_card(Card(2, 3))
print(boat)
```

```
boat.has_full_house()     # nên trả về True
```

```
pair.has_full_house()     # nên trả về False
```

```
good_hand.has_full_house()     # nên trả về False
```

Bài tập 7

Bài tập này là một câu chuyện cảnh báo về một lỗi phổ biến có thể khó gỡ lỗi. Hãy xem xét định nghĩa lớp sau.

```
class Kangaroo:
    """Một con Kangaroo là một loài thú có túi."""

    def __init__(self, name, contents=[]):
        """Khởi tạo nội dung trong túi.

        name: chuỗi
        contents: khởi tạo nội dung trong túi.
        """
        self.name = name
        self.contents = contents

    def __str__(self):
        """Trả về một chuỗi đại diện cho con Kangaroo này.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Thêm một phần tử mới vào nội dung trong túi.

        item: đối tượng đã được thêm vào
        """
        self.contents.append(item)
```

`__init__` nhận hai tham số: `name` là bắt buộc, nhưng `contents` là tùy chọn — nếu không được cung cấp, giá trị mặc định sẽ là một danh sách rỗng.

`__str__` trả về một biểu diễn chuỗi của đối tượng, bao gồm tên và nội dung của chiếc túi.

`put_in_pouch` nhận bất kỳ đối tượng nào và thêm nó vào `contents`.

Bây giờ, hãy xem cách lớp này hoạt động. Chúng ta sẽ tạo hai đối tượng `Kangaroo` với tên `'Kanga'` và `'Roo'`.

```
kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
```

Chúng ta sẽ thêm hai chuỗi và đối tượng Roo vào túi của Kanga.

```
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
kanga.put_in_pouch(roo)
```

Nếu chúng ta in ra `kanga`, có vẻ như mọi thứ đã hoạt động đúng.

```
print(kanga)
```

Nhưng điều gì sẽ xảy ra nếu chúng ta in ra `roo`?

```
print(roo)
```

Giỏ của Roo chứa cùng một nội dung như giỏ của Kanga, bao gồm cả tham chiếu đến `roo`!

Hãy thử tìm hiểu xem điều gì đã sai. Sau đó, bạn có thể hỏi trợ lý ảo: “Chương trình sau có vấn đề gì?” và dán vào định nghĩa của lớp `Kangaroo`.