

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

Dưới đây là các phiên bản của các lớp Card, Deck, và Hand từ [Chương 17](#), mà chúng ta sẽ sử dụng trong một số ví dụ trong chương này.

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

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

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

```
import random
```

```
class Deck:
```

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

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

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

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

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

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

```
class Hand(Deck):
```

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

✓ Các tính năng bổ sung trong Python

Một trong những mục tiêu của tôi trong cuốn sách này là dạy bạn càng ít Python càng tốt. Khi có hai cách để làm một việc, tôi chọn một cách và tránh đề cập đến cách còn lại. Hoặc đôi khi tôi đưa cách thứ hai vào bài tập.

Bây giờ tôi muốn quay lại và giới thiệu một số tính năng hay mà chúng ta đã bỏ qua. Python cung cấp một số tính năng mà thực ra không phải là cần thiết — bạn có thể viết mã tốt mà không cần chúng — nhưng khi sử dụng chúng, bạn có thể viết mã ngắn gọn, dễ đọc hoặc hiệu quả hơn, và đôi khi là cả ba điều này.

✓ Tập hợp trong Python

Python cung cấp một lớp gọi là `set` dùng để biểu diễn một tập hợp các phần tử duy nhất. Để tạo một tập hợp rỗng, chúng ta có thể sử dụng đối tượng lớp như một hàm.

```
s1 = set()  
s1
```

```
⇒ set()
```

Chúng ta có thể sử dụng phương thức `add` để thêm các phần tử.

```
s1.add('a')  
s1.add('b')  
s1
```

```
⇒ {'a', 'b'}
```

Hoặc chúng ta có thể truyền bất kỳ loại chuỗi nào vào `set`.

```
s2 = set('acd')  
s2
```

```
⇒ {'a', 'c', 'd'}
```

Một phần tử chỉ có thể xuất hiện một lần trong `set`. Nếu bạn thêm một phần tử đã tồn tại, nó sẽ không có hiệu lực.

```
s1.add('a')  
s1
```

⇒ {'a', 'b'}

Hoặc nếu bạn tạo một `set` từ một chuỗi chứa các phần tử trùng lặp, kết quả sẽ chỉ chứa các phần tử duy nhất.

```
set('banana')
```

⇒ {'a', 'b', 'n'}

Một số bài tập trong cuốn sách này có thể được thực hiện một cách ngắn gọn và hiệu quả bằng cách sử dụng `set`. Ví dụ, đây là một lời giải cho một bài tập trong Chương 11, sử dụng từ điển để kiểm tra xem có phần tử nào trùng lặp trong một chuỗi hay không.

```
def has_duplicates(t):  
    d = {}  
    for x in t:  
        d[x] = True  
    return len(d) < len(t)
```

Phiên bản này thêm các phần tử của `t` làm khóa trong một từ điển, sau đó kiểm tra xem số lượng khóa có ít hơn số lượng phần tử hay không. Sử dụng `set`, chúng ta có thể viết cùng một hàm như sau.

```
def has_duplicates(t):  
    s = set(t)  
    return len(s) < len(t)
```

```
has_duplicates('abba')
```

⇒ True

Một phần tử chỉ có thể xuất hiện một lần trong `set`, vì vậy nếu một phần tử trong `t` xuất hiện nhiều hơn một lần, `set` sẽ nhỏ hơn `t`. Nếu không có phần tử trùng lặp, kích thước của `set` sẽ bằng với `t`.

Đối tượng `set` cung cấp các phương thức thực hiện các phép toán trên tập hợp. Ví dụ, phương thức `union` tính hợp của hai tập hợp, tạo ra một tập hợp mới chứa tất cả các phần tử xuất hiện trong một trong hai tập hợp.

```
s1.union(s2)
```

```
➡ {'a', 'b', 'c', 'd'}
```

Một số toán tử số học hoạt động với `set`. Ví dụ, toán tử `-` thực hiện phép trừ tập hợp – kết quả là một tập hợp mới chứa tất cả các phần tử từ tập hợp đầu tiên không có trong tập hợp thứ hai.

```
s1 - s2
```

```
➡ {'b'}
```

Trong [Chương 12](#) chúng ta đã sử dụng từ điển để tìm các từ xuất hiện trong một tài liệu nhưng không có trong danh sách từ. Chúng ta đã sử dụng hàm sau, hàm này nhận hai từ điển và trả về một từ điển mới chỉ chứa các khóa từ từ điển thứ nhất mà không xuất hiện trong từ điển thứ hai.

```
def subtract(d1, d2):
    res = {}
    for key in d1:
        if key not in d2:
            res[key] = d1[key]
    return res
```

Với `set`, chúng ta không cần tự viết hàm này. Nếu `word_counter` là một từ điển chứa các từ duy nhất trong tài liệu và `word_list` là một danh sách các từ hợp lệ, chúng ta có thể tính phép trừ tập hợp như sau.

```
# Ô này tạo một ví dụ nhỏ để chúng ta có thể chạy
# ô tiếp theo mà không cần tải dữ liệu thực tế.
```

```
word_counter = {'word': 1}
word_list = ['word']
```

```
set(word_counter) - set(word_list)
```

⇒ `set()`

Kết quả là một tập hợp chứa các từ trong tài liệu không xuất hiện trong danh sách từ.

Các toán tử quan hệ hoạt động với `set`. Ví dụ, toán tử `<=` kiểm tra xem một tập hợp có phải là tập con của một tập hợp khác hay không, bao gồm cả khả năng chúng bằng nhau.

```
set('ab') <= set('abc')
```

⇒ `True`

Với các toán tử này, chúng ta có thể sử dụng `set` để giải một số bài tập trong Chương 7. Ví dụ, đây là một phiên bản của hàm `uses_only` sử dụng vòng lặp.

```
def uses_only(word, available):  
    for letter in word:  
        if letter not in available:  
            return False  
    return True
```

`uses_only` kiểm tra xem tất cả các chữ cái trong `word` có nằm trong `available` hay không. Với `set`, chúng ta có thể viết lại nó như sau.

```
def uses_only(word, available):  
    return set(word) <= set(available)
```

Nếu các chữ cái trong `word` là tập con của các chữ cái trong `available`, điều đó có nghĩa là `word` chỉ sử dụng các chữ cái trong `available`.

✓ Hàm counter trong Python

Một Counter giống như một set, ngoại trừ việc nếu một phần tử xuất hiện nhiều hơn một lần, Counter sẽ theo dõi số lần nó xuất hiện. Nếu bạn quen thuộc với khái niệm toán học về "tập hợp", thì Counter là một cách tự nhiên để biểu diễn một tập hợp.

Lớp Counter được định nghĩa trong một mô-đun có tên là `collections`, vì vậy bạn cần phải `import` nó. Sau đó, bạn có thể sử dụng đối tượng lớp như một hàm và truyền vào một chuỗi, danh sách, hoặc bất kỳ loại chuỗi nào khác.

```
from collections import Counter
```

```
counter = Counter('banana')  
counter
```

```
⇒ Counter({'b': 1, 'a': 3, 'n': 2})
```

```
from collections import Counter
```

```
t = (1, 1, 1, 2, 2, 3)  
counter = Counter(t)  
counter
```

```
⇒ Counter({1: 3, 2: 2, 3: 1})
```

Một đối tượng Counter giống như một từ điển, mà trong đó mỗi khóa được ánh xạ đến số lần nó xuất hiện. Giống như trong từ điển, các khóa phải là các đối tượng có thể băm.

Khác với từ điển, đối tượng Counter sẽ không ném ra một ngoại lệ nếu bạn truy cập vào một phần tử không xuất hiện. Thay vào đó, nó sẽ trả về giá trị 0.

```
counter['d']
```

```
⇒ 0
```

Chúng ta có thể sử dụng đối tượng Counter để giải một bài tập trong Chương 10, bài tập yêu cầu viết một hàm nhận vào hai từ và kiểm tra xem chúng có phải là từ đảo từ hay không — tức là, liệu các chữ cái của một từ có thể được sắp xếp lại để tạo thành từ còn lại hay không.

Dưới đây là một giải pháp sử dụng đối tượng Counter.

```
def is_anagram(word1, word2):  
    return Counter(word1) == Counter(word2)
```

Nếu hai từ là từ đảo từ, chúng chứa các chữ cái giống nhau với số lần xuất hiện giống nhau, vì vậy các đối tượng Counter của chúng là tương đương.

Counter cung cấp một phương thức gọi là `most_common`, phương thức này trả về một danh sách các cặp giá trị-tần suất, được sắp xếp từ tần suất cao nhất đến thấp nhất.

```
counter.most_common()
```

```
⇒ [(1, 3), (2, 2), (3, 1)]
```

Chúng cũng cung cấp các phương thức và toán tử để thực hiện các phép toán giống như tập hợp, bao gồm cộng, trừ, hợp, và giao. Ví dụ, toán tử `+` kết hợp hai đối tượng Counter và tạo ra một Counter mới chứa các khóa từ cả hai đối tượng và tổng số lần xuất hiện của chúng.

Chúng ta có thể kiểm tra điều này bằng cách tạo một Counter với các chữ cái từ 'bans' và cộng nó với các chữ cái từ 'banana'.

```
counter2 = Counter('bans')  
counter + counter2
```

```
⇒ Counter({'1': 3, '2': 2, '3': 1, 'b': 1, 'a': 1, 'n': 1, 's': 1})
```

Bạn sẽ có cơ hội khám phá các phép toán khác của Counter trong các bài tập ở cuối chương này.

✓ Hàm defaultdict trong Python

Mô-đun `collections` cũng cung cấp `defaultdict`, giống như một từ điển, ngoại trừ việc nếu bạn truy cập vào một khóa không tồn tại, nó sẽ tự động tạo ra một giá trị mới.

Khi bạn tạo một `defaultdict`, bạn cung cấp một hàm được sử dụng để tạo các giá trị mới. Một hàm tạo ra các đối tượng đôi khi được gọi là `factory` (mẫu thiết kế). Các hàm có sẵn tạo ra danh sách, tập hợp, và các kiểu dữ liệu khác có thể được sử dụng làm `factory`.

Ví dụ, dưới đây là một `defaultdict` tạo ra một `list` mới khi cần.


```
from collections import defaultdict
```

```
d = defaultdict(list)
d
```

```
⇒ defaultdict(list, {})
```

Lưu ý rằng đối số là `list`, một đối tượng lớp, chứ không phải hàm `list()`, là một lời gọi hàm tạo ra một danh sách mới. Hàm `factory` không được gọi trừ khi chúng ta truy cập vào một khóa không tồn tại.

```
t = d['new key']
t
```

```
⇒ []
```

Danh sách mới, mà chúng ta gọi là `t`, cũng được thêm vào từ điển. Vì vậy, nếu chúng ta thay đổi `t`, sự thay đổi sẽ xuất hiện trong `d`:

```
t.append('new value')
d['new key']
```

```
⇒ ['new value']
```

Nếu bạn đang tạo một từ điển các danh sách, bạn thường có thể viết mã đơn giản hơn bằng cách sử dụng `defaultdict`.

Trong một bài tập trong [Chương 11](#), tôi đã tạo một từ điển ánh xạ từ một chuỗi chữ cái đã sắp xếp sang danh sách các từ có thể được đánh vần từ các chữ cái đó. Ví dụ, chuỗi 'opst' ánh xạ đến danh sách ['opts', 'post', 'pots', 'spot', 'stop', 'tops']. Dưới đây là mã gốc.

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

Và đây là phiên bản đơn giản hơn sử dụng `defaultdict`.

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

Trong các bài tập ở cuối chương, bạn sẽ có cơ hội thực hành sử dụng các đối tượng `defaultdict`.

```
from collections import defaultdict
```

```
d = defaultdict(list)
key = ('into', 'the')
d[key].append('woods')
d[key]
```

```
↔ ['woods']
```

✓ Biểu thức điều kiện

Câu lệnh điều kiện thường được sử dụng để chọn một trong hai giá trị, như thế này:

```
import math
x = 5
```

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

```
y
```


```
↔ 1.6094379124341003
```

Câu lệnh này kiểm tra xem x có phải là số dương không. Nếu có, nó tính lô-ga-rit của x . Nếu không, `math.log` sẽ đưa ra một lỗi `ValueError`. Để tránh việc dừng chương trình, chúng ta tạo ra một giá trị `NaN`, là một giá trị dấu thập phân đặc biệt đại diện cho "Không phải là một số" (Not a Number).

Chúng ta có thể viết câu lệnh này ngắn gọn hơn bằng cách sử dụng biểu thức điều kiện.

```
y = math.log(x) if x > 0 else float('nan')
```

`y`

 1.6094379124341003

Bạn có thể đọc gần như câu này như sau: "`y` nhận giá trị `log-x` nếu x lớn hơn 0; ngược lại, nó nhận giá trị `NaN`."

Các hàm đệ quy đôi khi có thể được viết ngắn gọn bằng cách sử dụng biểu thức điều kiện. Ví dụ, đây là một phiên bản của hàm `factorial` với một câu lệnh điều kiện.

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n-1)
```

Và đây là một phiên bản với một *biểu thức điều kiện*.

```
def factorial(n):  
    return 1 if n == 0 else n * factorial(n-1)
```

Một ứng dụng khác của biểu thức điều kiện là xử lý các đối số tùy chọn. Ví dụ, đây là định nghĩa lớp với phương thức `__init__` sử dụng câu lệnh điều kiện để kiểm tra một tham số với giá trị mặc định.

```
class Kangaroo:
    def __init__(self, name, contents=None):
        self.name = name
        if contents is None:
            contents = []
        self.contents = contents
```

Dưới đây là phiên bản sử dụng biểu thức điều kiện.

```
def __init__(self, name, contents=None):
    self.name = name
    self.contents = [] if contents is None else contents
```

Nói chung, bạn có thể thay thế câu lệnh điều kiện bằng một biểu thức điều kiện nếu cả hai nhánh đều chứa một biểu thức đơn và không có câu lệnh nào.

✓ Biểu thức danh sách

Trong các chương trước, chúng ta đã thấy một vài ví dụ khi bắt đầu với một danh sách rỗng và thêm các phần tử, từng phần tử một, sử dụng phương thức `append`. Ví dụ, giả sử chúng ta có một chuỗi chứa tiêu đề của một bộ phim và chúng ta muốn viết hoa tất cả các từ trong đó.

```
title = 'monty python and the holy grail'
```

Chúng ta có thể tách chuỗi thành một danh sách các chuỗi, lặp qua các chuỗi, viết hoa chúng, và thêm chúng vào danh sách.

```
t = []
for word in title.split():
    t.append(word.capitalize())

' '.join(t)

➡ 'Monty Python And The Holy Grail'
```

Chúng ta có thể làm điều tương tự một cách ngắn gọn hơn bằng cách sử dụng **biểu thức danh sách**.

```
t = [word.capitalize() for word in title.split()]
```

```
' '.join(t)
```

```
➦ 'Monty Python And The Holy Grail'
```

Các toán tử dấu ngoặc vuông chỉ ra rằng chúng ta đang xây dựng một danh sách mới. Biểu thức bên trong dấu ngoặc vuông chỉ rõ các phần tử của danh sách, và câu lệnh `for` cho biết dãy mà chúng ta đang lặp qua.

Cú pháp của biểu thức danh sách có thể có vẻ lạ, vì biến vòng lặp – các từ trong ví dụ này – xuất hiện trong biểu thức trước khi chúng ta định nghĩa nó. Tuy nhiên, bạn sẽ dần quen với điều này.

Ví dụ khác, trong [Chương 9](#), chúng ta đã sử dụng vòng lặp này để đọc các từ từ một tệp và thêm chúng vào một danh sách.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython2/master/code/')
```

```
➦ Downloaded words.txt
```

```
word_list = []
```

```
for line in open('words.txt'):
    word = line.strip()
    word_list.append(word)
```

```
len(word_list)
```

```
➦ 113783
```

Dưới đây là cách chúng ta có thể viết điều đó dưới dạng một biểu thức danh sách.

```
word_list = [line.strip() for line in open('words.txt')]
```

```
len(word_list)
```

```
➦ 113783
```

Một biểu thức danh sách cũng có thể có câu lệnh `if` để xác định các phần tử nào sẽ được đưa vào danh sách. Ví dụ, dưới đây là một vòng lặp `for` mà chúng ta đã sử dụng trong [Chương 10](#) để tạo ra danh sách chỉ chứa các từ trong `word_list` là từ đối xứng.

```
def is_palindrome(word):  
    return list(reversed(word)) == list(word)
```

```
palindromes = []
```

```
for word in word_list:  
    if is_palindrome(word):  
        palindromes.append(word)
```

```
palindromes[:10]
```

```
⇒ ['aa', 'aba', 'aga', 'aha', 'ala', 'alula', 'ama', 'ana', 'anna', 'ava']
```

Dưới đây là cách chúng ta có thể làm điều tương tự với một biểu thức danh sách.

```
palindromes = [word for word in word_list if is_palindrome(word)]
```

```
palindromes[:10]
```

```
⇒ ['aa', 'aba', 'aga', 'aha', 'ala', 'alula', 'ama', 'ana', 'anna', 'ava']
```

Khi một biểu thức danh sách được sử dụng làm đối số cho một hàm, chúng ta thường có thể bỏ qua dấu ngoặc vuông. Ví dụ, giả sử chúng ta muốn tính tổng của $\frac{1}{2^n}$ cho các giá trị của n từ 0 đến 9. Chúng ta có thể sử dụng một biểu thức danh sách như sau.

```
sum([1/2**n for n in range(10)])
```

```
⇒ 1.998046875
```

Hoặc chúng ta có thể bỏ qua dấu ngoặc vuông như thế này.

```
sum(1/2**n for n in range(10))
```

```
⇒ 1.998046875
```

Trong ví dụ này, đối số về mặt kỹ thuật là một biểu thức tạo sinh, chứ không phải là một biểu thức danh sách, và nó thực sự không tạo ra một danh sách. Nhưng ngoài điều đó ra, hành vi của nó vẫn giống như một biểu thức danh sách.

Biểu thức danh sách và biểu thức sinh đều ngắn gọn và dễ đọc, ít nhất là đối với các biểu thức đơn giản. Chúng thường nhanh hơn so với vòng lặp for tương đương, đôi khi là nhanh hơn rất nhiều. Vì vậy, nếu bạn cảm thấy tôi không đề cập đến chúng sớm hơn, tôi hiểu.

Tuy nhiên, để bảo vệ mình, biểu thức danh sách khó gỡ lỗi hơn vì bạn không thể đặt câu lệnh in vào trong vòng lặp. Tôi khuyên bạn chỉ nên sử dụng chúng nếu phép toán đủ đơn giản để bạn có thể làm đúng ngay từ lần đầu. Hoặc bạn có thể xem xét việc viết và gỡ lỗi một vòng lặp for và sau đó chuyển nó thành một biểu thức danh sách.

✓ Hàm any và all trong Python

Python cung cấp một hàm tích hợp sẵn, any, nhận một dãy các giá trị boolean và trả về True nếu bất kỳ giá trị nào trong dãy đó là True.

```
any([False, False, True])
```

⇒ True

any thường được sử dụng với các biểu thức tạo sinh

```
any(letter == 't' for letter in 'monty')
```

⇒ True

Ví dụ đó không hữu ích lắm vì nó thực hiện cùng một điều như toán tử in. Tuy nhiên, chúng ta có thể sử dụng any để viết các giải pháp ngắn gọn cho một số bài tập trong [Chương 7](#). Ví dụ, chúng ta có thể viết uses_none như thế này.

```
def uses_none(word, forbidden):  
    """Kiểm tra xem một từ có tránh được các chữ cái bị cấm không."""  
    return not any(letter in forbidden for letter in word)
```

```
uses_none('banana', 'xyz')
```

⇒ True

```
uses_none('apple', 'efg')
```

⇒ False

Hàm này lặp qua các chữ cái trong `word` và kiểm tra xem có chữ cái nào trong đó nằm trong `forbidden` không. Sử dụng `any` với một biểu thức tạo sinh là hiệu quả vì nó dừng ngay lập tức nếu tìm thấy giá trị `True`, vì vậy không cần phải lặp qua toàn bộ dãy.

Python cung cấp một hàm tích hợp sẵn khác, `all`, trả về `True` nếu mọi phần tử trong dãy đều là `True`. Chúng ta có thể sử dụng nó để viết một phiên bản ngắn gọn của `uses_all`.

```
def uses_all(word, required):  
    """Kiểm tra xem một từ có sử dụng tất cả các chữ cái yêu cầu không."""  
    return all(letter in word for letter in required)
```

```
uses_all('banana', 'ban')
```

⇒ True

```
uses_all('apple', 'api')
```

⇒ False

Các biểu thức sử dụng `any` và `all` có thể ngắn gọn, hiệu quả và dễ đọc.

✓ Hàm `namedtuple` trong Python

Mô-đun `collections` cung cấp một hàm gọi là `namedtuple` có thể được sử dụng để tạo ra các lớp đơn giản. Ví dụ, đối tượng `Point` trong [Chương 16](#) chỉ có hai thuộc tính, `x` và `y`. Dưới đây là cách chúng ta đã định nghĩa nó.


```
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'({self.x}, {self.y})'
```

Đó là một lượng mã lệnh khá lớn để truyền đạt một lượng thông tin nhỏ. `namedtuple` cung cấp một cách ngắn gọn hơn để định nghĩa các lớp như thế này.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
```

Đối số đầu tiên là tên của lớp mà bạn muốn tạo. Đối số thứ hai là một danh sách các thuộc tính mà các đối tượng `Point` nên có. Kết quả là một đối tượng lớp, đó là lý do tại sao nó được gán cho một tên biến viết hoa.

Một lớp được tạo ra với `namedtuple` cung cấp một phương thức `__init__` để gán giá trị cho các thuộc tính và một phương thức `__str__` để hiển thị đối tượng dưới dạng có thể đọc được. Vì vậy, chúng ta có thể tạo và hiển thị một đối tượng `Point` như thế này.

```
p = Point(1, 2)
p
```

```
➡ Point(x=1, y=2)
```

`Point` cũng cung cấp một phương thức `__eq__` để kiểm tra xem hai đối tượng `Point` có bằng nhau không – tức là, kiểm tra xem các thuộc tính của chúng có giống nhau hay không.

```
p == Point(1, 2)
```

```
➡ True
```

Bạn có thể truy cập các phần tử của một **tuple có tên (namedtuple)** bằng tên hoặc bằng chỉ số.

```
p.x, p.y
```

```
⇒ (1, 2)
```

```
p[0], p[1]
```

```
⇒ (1, 2)
```

Bạn cũng có thể xem một **tuple có tên** như một tuple, như trong phép gán này.

```
x, y = p
```

```
x, y
```

```
⇒ (1, 2)
```

Tuy nhiên, các đối tượng `namedtuple` là bất biến. Sau khi các thuộc tính được khởi tạo, chúng không thể bị thay đổi.

```
%%expect TypeError
```

```
p[0] = 3
```

```
⇒ TypeError: 'Point' object does not support item assignment
```

```
%%expect AttributeError
```

```
p.x = 3
```

```
⇒ AttributeError: can't set attribute
```

`namedtuple` cung cấp một cách nhanh chóng để định nghĩa các lớp đơn giản. Nhược điểm là các lớp đơn giản không phải lúc nào cũng giữ được sự đơn giản. Bạn có thể quyết định sau này rằng bạn muốn thêm các phương thức vào một tuple có tên. Trong trường hợp đó, bạn có thể định nghĩa một lớp mới kế thừa từ tuple có tên.

```
class Pointier(Point):  
    """Lớp này kế thừa từ lớp Point."""
```

Hoặc vào thời điểm đó, bạn có thể chuyển sang việc định nghĩa lớp thông thường.

✓ Đóng gói đối số theo từ khoá

Trong [Chương 11](#), chúng ta đã viết một hàm đóng gói các đối số của nó vào một tuple.

```
def mean(*args):  
    return sum(args) / len(args)
```

Bạn có thể gọi hàm này với bất kỳ số lượng đối số nào.

```
mean(1, 2, 3)
```

```
⇒ 2.0
```

Tuy nhiên, toán tử `*` không đóng gói các đối số theo từ khóa. Vì vậy, gọi hàm này với một đối số từ khóa sẽ gây ra lỗi.

```
%%expect TypeError
```

```
mean(1, 2, start=3)
```

```
⇒ TypeError: mean() got an unexpected keyword argument 'start'
```

Để đóng gói các đối số theo từ khóa, chúng ta có thể sử dụng toán tử `**`:

```
def mean(*args, **kwargs):  
    print(kwargs)  
    return sum(args) / len(args)
```

Tham số đóng gói theo từ khóa có thể có bất kỳ tên gì, nhưng `kwargs` là một lựa chọn phổ biến. Kết quả là một từ điển ánh xạ từ các từ khóa đến các giá trị.

```
mean(1, 2, start=3)
```

```
➡ {'start': 3}  
1.5
```

Trong ví dụ này, giá trị của `kwargs` được in ra, nhưng ngoài ra nó không có tác dụng gì.

Tuy nhiên, toán tử `**` cũng có thể được sử dụng trong danh sách đối số để giải nén một từ điển. Ví dụ, đây là phiên bản của hàm `mean` đóng gói bất kỳ đối số từ khóa nào mà nó nhận được và sau đó giải nén chúng như các đối số từ khóa cho hàm `sum`.

```
def mean(*args, **kwargs):  
    return sum(args, **kwargs) / len(args)
```

Bây giờ, nếu chúng ta gọi hàm `mean` với `start` là một đối số từ khóa, nó sẽ được chuyển tiếp đến hàm `sum`, hàm này sử dụng `start` như là điểm bắt đầu của phép tính tổng.

Trong ví dụ dưới đây, `start=3` sẽ cộng thêm 3 vào tổng trước khi tính giá trị trung bình, vì vậy tổng là 6 và kết quả là 3.

```
mean(1, 2, start=3)
```

```
➡ 3.0
```

Ví dụ khác, nếu chúng ta có một từ điển với các khóa `x` và `y`, chúng ta có thể sử dụng nó với toán tử giải nén `**` để tạo ra một đối tượng `Point`.

```
d = dict(x=1, y=2)  
Point(**d)
```

```
➡ Point(x=1, y=2)
```

Không có toán tử giải nén, `d` sẽ được xử lý như một đối số vị trí duy nhất, vì vậy nó sẽ được gán cho `x`, và chúng ta sẽ gặp lỗi `TypeError` vì không có đối số thứ hai để gán cho `y`.

```
%%expect TypeError
```

```
d = dict(x=1, y=2)  
Point(d)
```

Khi làm việc với các hàm có số lượng đối số từ khóa lớn, thường thì việc tạo và truyền các từ điển xác định các tùy chọn thường xuyên sử dụng là rất hữu ích.

```
def pack_and_print(**kwargs):  
    print(kwargs)
```

```
pack_and_print(a=1, b=2)
```

```
⇒ {'a': 1, 'b': 2}
```

✓ Gỡ lỗi

Trong các chương trước, chúng ta đã sử dụng `doctest` để kiểm tra các hàm. Ví dụ, đây là một hàm có tên `add` nhận hai số và trả về tổng của chúng. Hàm này bao gồm một `doctest` để kiểm tra xem `2 + 2` có bằng 4 không.

```
def add(a, b):  
    '''Cộng hai số.  
  
    >>> add(2, 2)  
    4  
    ...  
    return a + b
```

Hàm này nhận một đối tượng hàm và chạy các bài kiểm tra trong phần `doctest` của nó.

```
from doctest import run_docstring_examples
```

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

Vì vậy, chúng ta có thể kiểm tra `add` như thế này.

```
run_doctests(add)
```



```
PYDEV DEBUGGER WARNING:
sys.settrace() should not be used when the debugger is being used.
This may cause the debugger to stop working correctly.
If this is needed, please check:
http://pydev.blogspot.com/2007/06/why-cant-pydev-debugger-work-with.html
to see how to restore the debug tracing back correctly.
Call Location:
  File "/usr/lib/python3.11/doctest.py", line 1523, in run
    sys.settrace(save_trace)
```

Không có đầu ra nào, điều đó có nghĩa là tất cả các bài kiểm tra đã vượt qua.

Python cung cấp một công cụ khác để chạy các bài kiểm tra tự động, gọi là `unittest`. Nó hơi phức tạp hơn để sử dụng, nhưng đây là một ví dụ.

```
from unittest import TestCase

class TestExample(TestCase):

    def test_add(self):
        result = add(2, 2)
        self.assertEqual(result, 4)
```

Đầu tiên, chúng ta nhập `TestCase`, một lớp trong mô-đun `unittest`. Để sử dụng nó, chúng ta cần định nghĩa một lớp mới kế thừa từ `TestCase` và cung cấp ít nhất một phương thức kiểm tra. Tên của phương thức kiểm tra phải bắt đầu bằng `test` và nên biểu thị rõ hàm nào nó kiểm tra.

Trong ví dụ này, `test_add` kiểm tra hàm `add` bằng cách gọi hàm, lưu kết quả và sử dụng `assertEqual`, được kế thừa từ `TestCase`. `assertEqual` nhận hai đối số và kiểm tra xem chúng có bằng nhau hay không.

Để chạy phương thức kiểm tra này, chúng ta cần chạy một hàm trong `unittest` gọi là `main` và cung cấp một số tham số từ khóa. Hàm sau đây hiển thị chi tiết – nếu bạn tò mò, bạn có thể hỏi một trợ lý ảo để giải thích cách nó hoạt động.

```
import unittest

def run_unittest():
    unittest.main(argv=[''], verbosity=0, exit=False)
```

`run_unittest` không nhận `TestExample` làm tham số — thay vào đó, nó tìm kiếm các lớp kế thừa từ `TestCase`. Sau đó, nó tìm các phương thức bắt đầu bằng `test` và chạy chúng. Quá trình này được gọi là phát hiện kiểm tra.

Đây là những gì xảy ra khi chúng ta gọi `run_unittest`.

```
run_unittest()
```



```
-----  
Ran 1 test in 0.000s
```

```
OK
```

`unittest.main` báo cáo số lượng kiểm tra đã chạy và kết quả. Trong trường hợp này, `OK` cho biết các kiểm tra đã vượt qua.

Để xem điều gì xảy ra khi một kiểm tra thất bại, chúng ta sẽ thêm một phương thức kiểm tra sai vào `TestExample`

```
%add_method_to TestExample
```

```
def test_add_broken(self):  
    result = add(2, 2)  
    self.assertEqual(result, 100)
```

Đây là những gì xảy ra khi chúng ta chạy các bài kiểm tra.

```
run_unittest()
```



```
=====
```

```
FAIL: test_add_broken (__main__.TestExample.test_add_broken)
```

```
-----
```

```
Traceback (most recent call last):
```

```
  File "<ipython-input-90-0c44ca9326b3>", line 3, in test_add_broken
```

```
    self.assertEqual(result, 100)
```

```
AssertionError: 4 != 100
```

```
-----
```

```
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

Báo cáo bao gồm phương thức kiểm tra bị lỗi và một thông báo lỗi chỉ ra vị trí lỗi. Phần tóm tắt cho biết hai bài kiểm tra đã chạy và một bài kiểm tra thất bại.

Trong các bài tập dưới đây, tôi sẽ gợi ý một số câu hỏi bạn có thể sử dụng để hỏi trợ lý ảo thêm thông tin về `unittest`.

Thuật ngữ

- **factory - hàm tạo**: Một hàm được sử dụng để tạo đối tượng, thường được truyền như một tham số cho một hàm.
- **conditional expression - biểu thức điều kiện**: Một biểu thức sử dụng điều kiện để chọn một trong hai giá trị.
- **list comprehension - biểu thức tạo danh sách**: Một cách ngắn gọn để lặp qua một dãy và tạo ra một danh sách.
- **generator expression - biểu thức tạo sinh**: Tương tự như biểu thức tạo danh sách, nhưng nó không tạo ra một danh sách.
- **test discovery - khám phá kiểm thử**: Quá trình được sử dụng để tìm và chạy các bài kiểm tra.

✓ 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

Có một vài chủ đề trong chương này mà bạn có thể muốn tìm hiểu thêm. Dưới đây là một số câu hỏi bạn có thể hỏi một AI.

- “Các phương thức và toán tử của lớp set trong Python là gì?”
- “Các phương thức và toán tử của lớp Counter trong Python là gì?”
- “Sự khác biệt giữa biểu thức tạo danh sách và biểu thức tạo sinh trong Python là gì?”
- “Khi nào tôi nên sử dụng `namedtuple` trong Python thay vì định nghĩa một lớp mới?”
- “Một số cách sử dụng của việc đóng gói và giải nén các đối số từ khóa là gì?”
- “Cách `unittest` thực hiện việc phát hiện khám phá kiểm thử như thế nào?”
- “Cùng với `assertEqual`, những phương thức nào là thường được sử dụng nhất trong `unittest.TestCase`?”
- “Ưu và nhược điểm của `doctest` và `unittest` là gì?”

Đối với các bài tập tiếp theo, bạn có thể xem xét yêu cầu sự trợ giúp của AI, nhưng như mọi khi, hãy nhớ kiểm tra kết quả.

✓ Bài tập 1

Một trong các bài tập trong Chương 7 yêu cầu một hàm gọi là `uses_none` nhận vào một từ và một chuỗi các chữ cái bị cấm, và trả về `True` nếu từ đó không sử dụng bất kỳ chữ cái nào trong chuỗi. Dưới đây là một giải pháp.

```
def uses_none(word, forbidden):  
    for letter in word.lower():  
        if letter in forbidden.lower():  
            return False  
    return True
```

Viết một phiên bản của hàm này sử dụng các phép toán set thay vì vòng lặp `for`.

Gợi ý: Hỏi một AI "Làm thế nào để tính toán giao nhau của các tập hợp (sets) trong Python?"

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

```
def uses_none(word, forbidden):
    """Kiểm tra xem một từ có tránh được các chữ cái bị cấm không.

    >>> uses_none('banana', 'xyz')
    True
    >>> uses_none('apple', 'efg')
    False
    >>> uses_none('', 'abc')
    True
    """
    return False
```

```
from doctest import run_docstring_examples
```

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

```
run_doctests(uses_none)
```

✓ Bài tập 2

Scrabble là một trò chơi trên bàn cờ, trong đó mục tiêu là sử dụng các miếng chữ cái để đánh vần các từ. Ví dụ, nếu chúng ta có các miếng chữ cái T, A, B, L, E, chúng ta có thể đánh vần BELT và LATE sử dụng một tập con các miếng chữ cái – nhưng không thể đánh vần BEET vì chúng ta không có hai chữ E.

Hãy viết một hàm nhận một chuỗi các chữ cái và một từ, và kiểm tra xem các chữ cái có thể đánh vần được từ đó hay không, tính đến số lần xuất hiện của mỗi chữ cái.

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

```
def can_spell(letters, word):
    """Kiểm tra xem các chữ cái có thể tạo thành từ này không.

    >>> can_spell('table', 'belt')
    True
    >>> can_spell('table', 'late')
    True
    >>> can_spell('table', 'beet')
    False
    """
    return False

run_doctests(can_spell)
```

✓ Bài tập 3

Trong một bài tập từ [Chương 17](#), giải pháp của tôi cho hàm `has_straightflush` sử dụng phương thức sau, phương thức này phân tách một `PokerHand` thành một danh sách gồm bốn bộ bài, mỗi bộ bài chứa các lá bài cùng chất.

```
def partition(self):
    """Tạo một danh sách gồm bốn bộ bài, mỗi bộ chỉ chứa một chất."""
    hands = []
    for i in range(4):
        hands.append(PokerHand())

    for card in self.cards:
        hands[card.suit].add_card(card)

    return hands
```

Viết một phiên bản đơn giản hóa của hàm này sử dụng `defaultdict`.

Dưới đây là một phác thảo của lớp `PokerHand` và hàm `partition_suits` mà bạn có thể sử dụng để bắt đầu.

```
class PokerHand(Hand):

    def partition(self):
        return {}
```

Để kiểm tra mã của bạn, chúng ta sẽ tạo một bộ bài và xáo trộn nó.

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

Sau đó, tạo một PokerHand và thêm bảy lá bài vào đó.

```
random_hand = PokerHand('random')  
  
for i in range(7):  
    card = deck.pop_card()  
    random_hand.add_card(card)  
  
print(random_hand)
```

Nếu bạn gọi partition và in kết quả, mỗi tay bài sẽ chỉ chứa các lá bài của một chất duy nhất.

```
hand_dict = random_hand.partition()  
  
for hand in hand_dict.values():  
    print(hand)  
    print()
```

✓ Bài tập 4

Dưới đây là hàm từ Chương 11 tính các số Fibonacci.

```
def fibonacci(n):  
    if n == 0:  
        return 0  
  
    if n == 1:  
        return 1  
  
    return fibonacci(n-1) + fibonacci(n-2)
```

Viết một phiên bản của hàm này với một câu lệnh return duy nhất sử dụng hai biểu thức điều kiện, một trong số đó lồng vào trong biểu thức còn lại.

Đáp án được viết ở đây

```
fibonacci(10)    # nên là 55
```

```
fibonacci(20)    # nên là 6765
```

✓ Bài tập 5

Dưới đây là một hàm tính hệ số nhị thức theo cách đệ quy.

```
def binomial_coeff(n, k):
    """Tính hệ số nhị thức “n chọn k”.

    n: Số lần thử nghiệm
    k: Số lần thành công

    returns: kiểu int
    """
    if k == 0:
        return 1

    if n == 0:
        return 0

    return binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
```

Viết lại phần thân của hàm sử dụng các biểu thức điều kiện lồng nhau.

Hàm này không hiệu quả lắm vì nó sẽ tính lại cùng một giá trị nhiều lần. Hãy làm cho nó hiệu quả hơn bằng cách lưu trữ kết quả đã tính, như đã mô tả trong [Chương 10](#).

Đáp án được viết ở đây

```
binomial_coeff(10, 4)    # should be 210
```

✓ Bài tập 6

Dưới đây là phương thức `__str__` của lớp `Deck` trong [Chương 17](#).

```
%%add_method_to Deck
```

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

iết một phiên bản ngắn gọn hơn của phương thức này sử dụng biểu thức tạo danh sách hoặc biểu thức tạo sinh.

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

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