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

from os.path import basename, exists

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

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

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

import thinkpython
```

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

Tuple trong Python

Chương này giới thiệu một kiểu dữ liệu tích hợp sẵn khác, là **tuple (hay còn gọi là bộ giá trị)**, và sau đó giải thích cách các danh sách, từ điển, và `tuple` hoạt động cùng nhau. Chương cũng trình bày về việc gán một `tuple` - một tính năng hữu ích cho các hàm với danh sách tham số có độ dài linh hoạt: các toán tử đóng gói và mở gói.

Trong phần bài tập, chúng ta sẽ sử dụng `tuple`, cùng với danh sách và từ điển, để giải các câu đố từ ngữ và triển khai các thuật toán hiệu quả.

Lưu ý: Từ "tuple" có hai cách phát âm. Một số người phát âm là "tuh-ple", vần với "supple". Nhưng trong ngữ cảnh lập trình, hầu hết mọi người phát âm là "too-ple", vần với "quadruple".

Tuple giống như danh sách

Tuple là một chuỗi các giá trị. Các giá trị trong một tuple có thể thuộc bất kỳ loại nào, và chúng được đánh chỉ số bằng các số nguyên, vì vậy tuple rất giống với danh sách. Điểm khác biệt quan trọng, tuple là bất biến.

Để tạo một tuple, bạn có thể viết một danh sách các giá trị, cách nhau bằng dấu phẩy.

```
t = 'l', 'u', 'p', 'i', 'n'
type(t)
```

```
tuple
```

Mặc dù không bắt buộc, nhưng thông thường người ta đặt các tuple trong dấu ngoặc đơn.

```
t = ('l', 'u', 'p', 'i', 'n')
type(t)
```

```
tuple
```

Để tạo một tuple với một phần tử duy nhất, bạn cần thêm một dấu phẩy ở cuối.

```
t1 = 'p',
type(t1)
```

```
tuple
```

Một giá trị đơn lẻ trong dấu ngoặc đơn không phải là một tuple.

```
t2 = ('p')
type(t2)
```

```
str
```

Một cách khác để tạo một tuple là sử dụng hàm tích hợp `tuple`. Khi không có đối số, nó sẽ tạo ra một tuple trống.

```
t = tuple()
t
```

```
()
```

Nếu đối số là một chuỗi (chuỗi, danh sách hoặc tuple), kết quả sẽ là một tuple chứa các phần tử của chuỗi đó.

```
t = tuple('lupin')
t
```

```
('l', 'u', 'p', 'i', 'n')
```

Vì `tuple` là tên của một hàm tích hợp sẵn, bạn nên tránh sử dụng nó làm tên biến.

Hầu hết các toán tử dùng cho danh sách cũng hoạt động với `tuple`. Ví dụ, toán tử dấu ngoặc vuông sẽ truy cập một phần tử theo chỉ mục.

```
t[0]
```

```
'l'
```

Và toán tử cắt lát chọn một dãy các phần tử.

```
t[1:3]
```

```
('u', 'p')
```

Toán tử `+` nối các tuple lại với nhau.

```
tuple('lup') + ('i', 'n')
```

```
('l', 'u', 'p', 'i', 'n')
```

Và toán tử `*` sao chép một tuple một số lần nhất định.

```
tuple('spam') * 2
```

```
('s', 'p', 'a', 'm', 's', 'p', 'a', 'm')
```

Hàm `sorted` hoạt động với các tuple — nhưng kết quả là một danh sách, chứ không phải một tuple.

```
sorted(t)
```

```
['i', 'l', 'n', 'p', 'u']
```

Hàm `reversed` cũng hoạt động với các tuple.

```
reversed(t)
```

```
<reversed at 0x7e2c3fd65540>
```

Kết quả là một đối tượng `reversed`, mà chúng ta có thể chuyển đổi thành danh sách hoặc tuple.

```
tuple(reversed(t))
```

```
('n', 'i', 'p', 'u', 'l')
```

Dựa trên các ví dụ cho đến nay, có vẻ như tuple và danh sách là giống nhau.

Nhưng tuple là bất biến

Nếu bạn cố gắng sửa đổi một tuple bằng toán tử dấu ngoặc vuông, bạn sẽ nhận được lỗi `TypeError`.

```
%%expect TypeError
t[0] = 'L'
```

```
TypeError: 'tuple' object does not support item assignment
```

Và tuple không có bất kỳ phương thức nào để sửa đổi danh sách, như `append` và `remove`.

```
%%expect AttributeError

t.remove('l')
```

```
AttributeError: 'tuple' object has no attribute 'remove'
```

Nhớ rằng `thuộc tính` là một biến hoặc phương thức gắn liền với một đối tượng — thông báo lỗi này có nghĩa là tuple không có phương thức có tên `remove`.

Vì tuple là bất biến, chúng có thể được băm, điều này có nghĩa là chúng có thể được sử dụng làm khóa trong một từ điển. Ví dụ, từ điển dưới đây chứa hai tuple làm khóa, mỗi khóa ánh xạ đến một số nguyên.

```
d = {}
d[1, 2] = 3
d[3, 4] = 7
```

Chúng ta có thể tra cứu một tuple trong một từ điển như thế này:

```
d[1, 2]
```

```
3
```

Hoặc nếu chúng ta có một biến tham chiếu đến một tuple, chúng ta có thể sử dụng nó làm khóa.

```
t = (3, 4)
d[t]
```

```
7
```

Tuple cũng có thể xuất hiện như là giá trị trong một từ điển.

```
t = tuple('abc')
s = [1, 2, 3]
d = {t: s}
d
```

```
{('a', 'b', 'c'): [1, 2, 3]}
```

Gán tuple

Bạn có thể đặt các biến của một tuple ở bên trái dấu gán, và các gía trị của một tuple ở bên phải.

```
a, b = 1, 2
```

Giá trị được gán cho các biến từ trái sang phải — trong ví dụ này, `a` nhận giá trị `1` và `b` nhận giá trị `2`. Chúng ta có thể hiển thị kết quả như sau:

```
a, b
```

```
(1, 2)
```

Hơn nữa, nếu bên trái của phép gán là một tuple, bên phải có thể là bất kỳ loại dữ liệu nào như chuỗi, danh sách hoặc tuple. Ví dụ, để tách một địa chỉ email thành tên người dùng và tên miền, bạn có thể viết:

```
email = 'monty@python.org'
username, domain = email.split('@')
```

Giá trị trả về từ `split` là một danh sách với hai phần tử — phần tử đầu tiên được gán cho `username`, phần tử thứ hai được gán cho `domain`.

```
username, domain
```

```
('monty', 'python.org')
```

Số lượng biến ở bên trái và số lượng giá trị ở bên phải phải giống nhau — nếu không, bạn sẽ nhận được lỗi `ValueError`.

```
%%expect ValueError
a, b = 1, 2, 3
```

```
ValueError: too many values to unpack (expected 2)
```

Gán tuple rất hữu ích nếu bạn muốn hoán đổi giá trị của hai biến. Với phép gán thông thường, bạn phải sử dụng một biến tạm thời, như thế này:

```
temp = a
a = b
b = temp
```

Cách đó cũng hoạt động, nhưng với toán tử gán tuple, chúng ta có thể làm điều tương tự mà không cần một biến tạm thời.

```
a, b = b, a
```

Cách này hoạt động vì tất cả các biểu thức ở bên phải đều được đánh giá trước khi thực hiện bất kỳ phép gán nào.

Chúng ta cũng có thể sử dụng phép gán tuple trong câu lệnh `for`. Ví dụ, để lặp qua các phần tử trong một từ điển, chúng ta có thể sử dụng phương thức `items`.

```
d = {'one': 1, 'two': 2}

for item in d.items():
    key, value = item
    print(key, '->', value)
```

```
one -> 1
two -> 2
```

Mỗi lần lặp qua vòng lặp, `item` được gán cho một tuple chứa một khóa và giá trị tương ứng.

Chúng ta có thể viết vòng lặp này một cách ngắn gọn hơn, như sau:

```
for key, value in d.items():
    print(key, '->', value)
```

```
one -> 1
two -> 2
```

Mỗi lần lặp qua vòng lặp, một khóa và giá trị tương ứng được gán trực tiếp cho `key` và `value`.

Tuple trả về nhiều giá trị

Nói một cách chính xác, một hàm chỉ có thể trả về một giá trị, nhưng nếu giá trị đó là một tuple, hiệu quả là giống như trả về nhiều giá trị. Ví dụ, nếu bạn muốn chia hai số nguyên, tính thương và số dư, thì việc tính `x // y` và sau đó `x % y` sẽ không hiệu quả. Tốt hơn là tính cả hai giá trị cùng một lúc.

Hàm tích hợp `divmod` nhận hai đối số và trả về một tuple chứa hai giá trị, thương và số dư.

```
divmod(7, 3)
```

```
(2, 1)
```

Chúng ta có thể sử dụng phép gán tuple để lưu trữ các phần tử của tuple vào hai biến.

```
quotient, remainder = divmod(7, 3)
quotient
```

```
2
```

```
remainder
```

```
1
```

Dưới đây là một ví dụ về một hàm trả về một tuple.

```
def min_max(t):
    return min(t), max(t)
```

`max` và `min` là các hàm tích hợp tìm phần tử lớn nhất và nhỏ nhất trong một dãy. `min_max` tính cả hai giá trị và trả về một tuple chứa hai giá trị.

```
min_max([2, 4, 1, 3])
```

```
(1, 4)
```

Chúng ta có thể gán kết quả cho các biến như thế này:

```
low, high = min_max([2, 4, 1, 3])
low, high
```

```
(1, 4)
```

Gói đối số

Hàm có thể nhận một số lượng đối số biến đổi. Một tên tham số bắt đầu với toán tử `*` có nghĩa là nó đang **gói** các đối số thành một tuple. Ví dụ, hàm dưới đây nhận bất kỳ số lượng đối số nào và tính giá trị trung bình cộng của chúng — tức là tổng của chúng chia cho số lượng đối số.

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

Tham số có thể có bất kỳ tên nào bạn thích, nhưng `args` là tên thường dùng theo thông lệ. Chúng ta có thể gọi hàm như sau:

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

```
2.0
```

Nếu bạn có một dãy giá trị và muốn truyền chúng vào một hàm dưới dạng nhiều đối số, bạn có thể sử dụng toán tử `*` để **tháo gói** tuple. Ví dụ, `divmod` yêu cầu chính xác hai đối số — nếu bạn truyền một tuple làm tham số, bạn sẽ gặp lỗi.

```
%%expect TypeError
t = (7, 3)
divmod(t)
```

```
TypeError: divmod expected 2 arguments, got 1
```

Mặc dù tuple chứa hai phần tử, nhưng nó được tính như một đối số duy nhất. Tuy nhiên, nếu bạn tháo gói tuple, nó sẽ được xem như hai đối số.

```
divmod(*t)
```

```
(2, 1)
```

Gói và tháo gói có thể hữu ích nếu bạn muốn điều chỉnh hành vi của một hàm có sẵn. Ví dụ, hàm này nhận bất kỳ số lượng đối số nào, loại bỏ giá trị nhỏ nhất và lớn nhất, sau đó tính trung bình cộng của các giá trị còn lại.

```
def trimmed_mean(*args):
    low, high = min_max(args)
    trimmed = list(args)
    trimmed.remove(low)
    trimmed.remove(high)
    return mean(*trimmed)
```

Đầu tiên, hàm sử dụng `min_max` để tìm phần tử nhỏ nhất và lớn nhất. Sau đó, nó chuyển `args` thành một danh sách để có thể sử dụng phương thức `remove`. Cuối cùng, nó tháo gói danh sách để các phần tử được truyền vào `mean` dưới dạng các đối số riêng lẻ, thay vì dưới dạng một danh sách duy nhất.

Dưới đây là một ví dụ cho thấy hiệu quả của hàm.

```
mean(1, 2, 3, 10)
```

```
4.0
```

```
trimmed_mean(1, 2, 3, 10)
```

```
2.5
```

Loại trung bình "đã cắt gọt" này được sử dụng trong một số môn thể thao có tính chất chấm điểm chủ quan — như nhảy cầu và thể dục dụng cụ — để giảm ảnh hưởng của một giám khảo có điểm số lệch so với những người khác.

Hàm zip

Tuple rất hữu ích khi lặp qua các phần tử của hai dãy và thực hiện các phép toán trên các phần tử tương ứng. Ví dụ, giả sử hai đội chơi một loạt bảy trận đấu, và chúng ta ghi lại điểm số của họ trong hai danh sách, một cho mỗi đội.

```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]
```

Hãy xem mỗi đội đã thắng bao nhiêu trận. Chúng ta sẽ sử dụng `zip`, một hàm tích hợp sẵn lấy hai hoặc nhiều dãy và trả về một **đối tượng zip**. Nó được gọi là như vậy vì nó ghép nối các phần tử của các dãy giống như răng của một khóa kéo.

```
zip(scores1, scores2)
```

```
<zip at 0x7e2c3fd9a280>
```

Chúng ta có thể sử dụng **đối tượng zip** để lặp qua các giá trị trong các dãy theo từng cặp.

```
for pair in zip(scores1, scores2):
     print(pair)
```

```
(1, 5)
(2, 5)
(4, 2)
(5, 2)
(1, 5)
(5, 2)
(2, 3)
```

Mỗi lần lặp qua vòng lặp, `pair` được gán một tuple chứa điểm số. Vì vậy, chúng ta có thể gán các điểm số cho các biến và đếm số chiến thắng cho đội đầu tiên, như thế này:

```
wins = 0
for team1, team2 in zip(scores1, scores2):
    if team1 > team2:
        wins += 1

wins
```

```
3
```

Đáng buồn thay, đội đầu tiên chỉ thắng ba trận và thua hàng loạt trận.

Nếu bạn có hai danh sách và muốn có một danh sách các cặp, bạn có thể sử dụng `zip` và `list`.

```
t = list(zip(scores1, scores2))
t
```

```
[(1, 5), (2, 5), (4, 2), (5, 2), (1, 5), (5, 2), (2, 3)]
```

Kết quả là một danh sách các tuple, vì vậy chúng ta có thể lấy kết quả của trận đấu cuối cùng như thế này:

```
t[-1]
```

```
(2, 3)
```

Nếu bạn có một danh sách các khóa và một danh sách các giá trị, bạn có thể sử dụng `zip` và `dict` để tạo một từ điển. Ví dụ, đây là cách chúng ta có thể tạo một từ điển ánh xạ từ mỗi chữ cái đến vị trí của nó trong bảng chữ cái.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

Bây giờ, chúng ta có thể tra cứu một chữ cái và nhận được chỉ số của nó trong bảng chữ cái.

```
letter_map['a'], letter_map['z']
```

```
(0, 25)
```

Trong phép ánh xạ này, chỉ số của `'a'` là 0 và chỉ số của `'z'` là 25.

Nếu bạn cần lặp qua các phần tử của một dãy và chỉ số của chúng, bạn có thể sử dụng hàm tích hợp sẵn `enumerate`.

```
enumerate('abc')
```

```
<enumerate at 0x7e2c4c5b9a80>
```

Kết quả là một đối tượng `enumerate` lặp qua một dãy các cặp, trong đó mỗi cặp chứa một chỉ số (bắt đầu từ 0) và một phần tử từ dãy đã cho.

```
for index, element in enumerate('abc'):
    print(index, element)
```

```
0 a
1 b
2 c
```

So sánh và sắp xếp

Các toán tử quan hệ hoạt động với tuple và các dãy khác. Ví dụ, nếu bạn sử dụng toán tử `<` với tuple, nó bắt đầu bằng cách so sánh phần tử đầu tiên của mỗi dãy. Nếu chúng bằng nhau, nó tiếp tục với cặp phần tử tiếp theo, và cứ thế, cho đến khi nó tìm thấy một cặp khác nhau.

```
(0, 1, 2) < (0, 3, 4)
```

```
True
```

Các phần tử tiếp theo không được xem xét - ngay cả khi chúng thực sự lớn.

```
(0, 1, 2000000) < (0, 3, 4)
```

```
True
```

Cách so sánh tuple này hữu ích để sắp xếp một danh sách các tuple hoặc tìm giá trị nhỏ nhất hoặc lớn nhất.

Ví dụ, hãy tìm chữ cái xuất hiện nhiều nhất trong một từ. Trong chương trước, chúng ta đã viết `value_counts`, hàm này nhận một chuỗi và trả về một từ điển ánh xạ từ mỗi chữ cái đến số lần xuất hiện của nó.

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

Kết quả cho chuỗi ký tự `'banana'` là:

```
counter = value_counts('banana')
counter
```

```
{'b': 1, 'a': 3, 'n': 2}
```

Với chỉ ba phần tử, chúng ta dễ dàng thấy rằng chữ cái xuất hiện nhiều nhất là `'a'`, xuất hiện ba lần. Nhưng nếu có nhiều phần tử hơn, việc sắp xếp tự động sẽ hữu ích.

Chúng ta có thể lấy các phần tử từ `counter` như thế này.

```
items = counter.items()
items
```

```
dict_items([('b', 1), ('a', 3), ('n', 2)])
```

Kết quả là một đối tượng `dict_items` hoạt động giống như một danh sách các tuple, vì vậy chúng ta có thể sắp xếp nó như thế này.

```
sorted(items)
```

```
[('a', 3), ('b', 1), ('n', 2)]
```

Hành vi mặc định là sử dụng phần tử đầu tiên của mỗi tuple để sắp xếp danh sách, và sử dụng phần tử thứ hai để phá vỡ các trường hợp bằng nhau.

Tuy nhiên, để tìm các phần tử có số lần xuất hiện cao nhất, chúng ta muốn sử dụng phần tử thứ hai để sắp xếp danh sách. Chúng ta có thể làm điều đó bằng cách viết một hàm nhận một tuple và trả về phần tử thứ hai.

```
def second_element(t):
    return t[1]
```

Sau đó, chúng ta có thể truyền hàm đó cho `sorted` dưới dạng một đối số tùy chọn được gọi là `key`, cho biết rằng hàm này nên được sử dụng để tính toán **sort key** cho mỗi mục.

```
sorted_items = sorted(items, key=second_element)
sorted_items
```

```
[('b', 1), ('n', 2), ('a', 3)]
```

Sort key xác định thứ tự của các phần tử trong danh sách. Chữ cái có số lần xuất hiện thấp nhất xuất hiện đầu tiên, và chữ cái có số lần xuất hiện cao nhất xuất hiện cuối cùng. Vì vậy, chúng ta có thể tìm chữ cái phổ biến nhất như thế này.

```
sorted_items[-1]
```

```
('a', 3)
```

Nếu chúng ta chỉ muốn tìm giá trị lớn nhất, chúng ta không cần phải sắp xếp danh sách. Chúng ta có thể sử dụng hàm `max`, hàm này cũng nhận `key` làm một đối số tùy chọn.

```
max(items, key=second_element)
```

```
('a', 3)
```

Để tìm chữ cái có tần suất thấp nhất, chúng ta có thể dùng hàm `min` với cách thức tương tự.

Đảo ngược từ điển

Giả sử bạn muốn đảo ngược một từ điển để có thể tra cứu một giá trị và nhận được khóa tương ứng. Ví dụ, nếu bạn có một bộ đếm từ ánh xạ từ mỗi từ đến số lần xuất hiện của nó, bạn có thể tạo một từ điển ánh xạ từ các số nguyên đến các từ xuất hiện số lần đó.

Nhưng có một vấn đề - các khóa trong một từ điển phải là duy nhất, nhưng các giá trị thì không. Ví dụ, trong một bộ đếm từ, có thể có nhiều từ có cùng số lần xuất hiện.

Vì vậy, một cách để đảo ngược một từ điển là tạo một từ điển mới trong đó các giá trị là các danh sách các khóa từ bản gốc. Ví dụ, hãy đếm các chữ cái trong từ `'parrot'`.

```
d =  value_counts('parrot')
d
```

```
{'p': 1, 'a': 1, 'r': 2, 'o': 1, 't': 1}
```

Nếu chúng ta đảo ngược từ điển này, kết quả sẽ là `{1: ['p', 'a', 'o', 't'], 2: ['r']}`, cho biết các chữ cái xuất hiện một lần là `'p'`, `'a'`, `'o'` và `'t'`, và chữ cái xuất hiện hai lần là `'r'`.

Hàm sau đây nhận một từ điển và trả về nghịch đảo của nó dưới dạng một từ điển mới.

```
def invert_dict(d):
    new = {}
    for key, value in d.items():
        if value not in new:
            new[value] = [key]
        else:
            new[value].append(key)
    return new
```

Cấu trúc lặp `for` lặp qua các khóa và giá trị trong `d`. Nếu giá trị chưa có trong từ điển mới, nó được thêm vào và liên kết với một danh sách chứa một phần tử duy nhất. Ngược lại, nó được nối vào danh sách hiện có.

Chúng ta có thể kiểm tra nó như thế này:

```
invert_dict(d)
```

```
{1: ['p', 'a', 'o', 't'], 2: ['r']}
```

Và chúng ta nhận được kết quả như mong đợi.

Đây là ví dụ đầu tiên mà chúng ta thấy các giá trị trong từ điển là danh sách. Chúng ta sẽ thấy nhiều hơn nữa!

Gỡ lỗi

Danh sách, từ điển và tuple là các cấu trúc dữ liệu. Trong chương này, chúng ta bắt đầu thấy các cấu trúc dữ liệu hỗn hợp, như danh sách các tuple hoặc các từ điển chứa các tuple làm khóa và danh sách làm giá trị. Các cấu trúc dữ liệu hỗn hợp rất hữu ích, nhưng chúng dễ bị lỗi do các cấu trúc dữ liệu có kiểu, kích thước, hoặc cấu trúc sai. Ví dụ, nếu một hàm mong đợi một danh sách các số nguyên và bạn cung cấp cho nó một số nguyên đơn giản (không nằm trong danh sách), nó có thể sẽ không hoạt động.

Để hỗ trợ gỡ lỗi các loại lỗi này, tôi đã viết một mô-đun được gọi là `structshape` cung cấp một hàm, cũng được gọi là `structshape`, nhận bất kỳ loại cấu trúc dữ liệu nào làm đối số và trả về một chuỗi tóm tắt cấu trúc của nó.

Bạn có thể tải xuống từ https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.py.

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

```
Downloaded structshape.py
```

Chúng ta có thể nhập nó như thế này.

```
from structshape import structshape
```

Đây là một ví dụ với một danh sách đơn giản.

```
t = [1, 2, 3]
structshape(t)
```

```
'list of 3 int'
```

Đây là một danh sách của các danh sách.

```
t2 = [[1,2], [3,4], [5,6]]
structshape(t2)
```

```
'list of 3 list of 2 int'
```

Nếu các phần tử của danh sách không cùng kiểu, `structshape` nhóm chúng theo kiểu.

```
t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
structshape(t3)
```

```
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Đây là một danh sách các tuple.

```
s = 'abc'
lt = list(zip(t, s))
structshape(lt)
```

```
'list of 3 tuple of (int, str)'
```

Và đây là một từ điển với ba mục ánh xạ các số nguyên thành các chuỗi.

```
d = dict(lt)
structshape(d)
```

Nếu bạn gặp khó khăn trong việc theo dõi các cấu trúc dữ liệu của mình, `structshape` có thể giúp đỡ.

Thuật ngữ

pack - đóng gói: Đóng gói nhiều đối số vào một tuple.

unpack - mở gói: Mở rộng một tuple (hoặc chuỗi khác) thành nhiều đối số riêng biệt.

zip object - đối tượng zip: Kết quả của việc gọi hàm tích hợp sẵn zip, được sử dụng để lặp qua một dãy các tuple.

enumerate object - đối tượng liệt kê: Kết quả của việc gọi hàm tích hợp sẵn `enumerate`, được sử dụng để lặp qua một dãy các tuple kèm theo chỉ số.

sort key - sort key: Một giá trị hoặc hàm tính toán một giá trị, được sử dụng để sắp xếp các phần tử của một tập hợp.

data structure - cấu trúc dữ liệu: Một tập hợp các giá trị được tổ chức để thực hiện các thao tác nhất định một cách hiệu quả.

Bài tập

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

%xmode Verbose
```

Hỏi một trợ lý ảo

Các bài tập trong chương này có thể khó hơn các bài tập trong các chương trước, vì vậy tôi khuyến khích bạn nhờ trợ giúp từ trợ lý ảo. Khi bạn đặt câu hỏi khó hơn, bạn có thể thấy rằng câu trả lời không chính xác ngay lần đầu tiên, vì vậy đây là cơ hội để thực hành xây dựng các lời nhắc tốt và theo dõi bằng các cải tiến tốt.

Một chiến lược bạn có thể cân nhắc là chia nhỏ một vấn đề lớn thành các phần có thể được giải quyết bằng các hàm đơn giản. Yêu cầu trợ lý ảo viết các hàm và kiểm tra chúng. Sau đó, khi chúng hoạt động, hãy yêu cầu giải pháp cho vấn đề ban đầu.

Đối với một số bài tập dưới đây, tôi đưa ra các đề xuất về cấu trúc dữ liệu và thuật toán nào nên sử dụng. Bạn có thể thấy những đề xuất này hữu ích khi làm việc với các bài toán, nhưng chúng cũng là những lời nhắc tốt để chuyển tiếp cho trợ lý ảo.

Bài tập 1

Trong chương này, tôi đã nói rằng các tuple có thể được sử dụng làm khóa trong các từ điển vì chúng có thể băm, và chúng có thể băm được vì chúng không thể thay đổi. Nhưng điều đó không phải lúc nào cũng đúng.

Nếu một tuple chứa một giá trị có thể thay đổi, như một danh sách hoặc một từ điển, thì tuple đó không còn có thể băm được nữa vì nó chứa các phần tử không thể băm. Ví dụ, đây là một tuple chứa hai danh sách số nguyên.

```
list0 = [1, 2, 3]
list1 = [4, 5]

t = (list0, list1)
t
```

Viết một dòng mã lệnh để thêm giá trị `6` vào cuối danh sách thứ hai trong tuple `t`. Nếu in ra `t`, kết quả sẽ là (`[1, 2, 3], [4, 5, 6]`).

Hãy thử tạo một từ điển ánh xạ từ `t` đến một chuỗi và xác nhận rằng bạn nhận được lỗi `TypeError`.

Để tìm hiểu thêm về chủ đề này, hãy hỏi trợ lý ảo: "Liệu tuple trong Python có luôn luôn có thể băm không?"

Bài tập 2

Trong chương này, chúng ta đã tạo một từ điển ánh xạ từ mỗi chữ cái đến chỉ số của nó trong bảng chữ cái.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

Ví dụ, chỉ số của `'a'` là 0.

```
letter_map['a']
```

Để đi theo hướng ngược lại, chúng ta có thể sử dụng lập chỉ mục danh sách. Ví dụ, chữ cái ở chỉ số 1 là `'b'`.

```
letters[1]
```

Chúng ta có thể sử dụng `letter_map` và `letters` để mã hóa và giải mã các từ bằng mật mã Caesar.

Mật mã Caesar là một hình thức mã hóa yếu liên quan đến việc dịch chuyển mỗi chữ cái một số vị trí cố định trong bảng chữ cái, quấn quanh trở lại đầu nếu cần. Ví dụ: `'a'` dịch chuyển 2 vị trí là `'c'` và `'z'` dịch chuyển 1 vị trí là `'a'`.

Viết một hàm có tên là `shift_word` nhận hai tham số là một chuỗi và một số nguyên, và trả về một chuỗi mới chứa các chữ cái từ chuỗi được dịch chuyển bởi số vị trí đã cho.

Để kiểm tra hàm của bạn, hãy xác nhận rằng "cheer" dịch chuyển 7 vị trí là "jolly" và "melon" dịch chuyển 16 vị trí là "cubed".

Gợi ý: Sử dụng toán tử modulo để quấn quanh từ `'z'` trở lại `'a'`. Lặp qua các chữ cái của từ, dịch chuyển từng chữ cái và nối kết quả vào một danh sách các chữ cái. Sau đó sử dụng join để nối các chữ cái thành một chuỗi.

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

```
def shift_word(word, n):
    """Dịch chuyển các chữ cái của `word` đi `n` vị trí.

    >>> shift_word('cheer', 7)
    'jolly'
    >>> shift_word('melon', 16)
    'cubed'
    """
    return None


shift_word('cheer', 7)


shift_word('melon', 16)
```

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

```
from doctest import run_docstring_examples

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

run_doctests(shift_word)
```

Bài tập 3

Viết một hàm có tên là `most_frequent_letters` nhận một chuỗi và in ra các chữ cái theo thứ tự giảm dần về tần suất xuất hiện.

Để lấy các phần tử theo thứ tự giảm dần, bạn có thể sử dụng `reversed` cùng với `sorted` hoặc bạn có thể truyền `reverse=True` làm tham số từ khóa cho `sorted`.

Bạn có thể sử dụng dàn ý của hàm này để bắt đầu.

```
def most_frequent_letters(string):
    return None
```

Và ví dụ này để kiểm tra hàm của bạn.

```
most_frequent_letters('brontosaurus')
```

Một khi hàm của bạn hoạt động, bạn có thể sử dụng đoạn mã sau để in ra các chữ cái phổ biến nhất trong *Dracula*, mà chúng ta có thể tải xuống từ **Project Gutenberg**.

```
download('https://www.gutenberg.org/cache/epub/345/pg345.txt');
```

```
string = open('pg345.txt').read()
most_frequent_letters(string)
```

Theo "Mật mã và viết bí mật" của Zim, dãy chữ cái theo thứ tự giảm dần về tần suất trong tiếng Anh bắt đầu bằng "ETAONRISH". Dãy này so sánh với kết quả từ Dracula như thế nào?

Bài tập 4

Trong một bài tập trước, chúng ta đã kiểm tra xem hai chuỗi có phải là từ đồng nghĩa của nhau bằng cách sắp xếp các chữ cái trong cả hai từ và kiểm tra xem các chữ cái đã sắp xếp có giống nhau hay không. Bây giờ, hãy làm cho vấn đề trở nên khó khăn hơn một chút.

Chúng ta sẽ viết một chương trình nhận một danh sách các từ và in ra tất cả các tập hợp các từ là từ đồng nghĩa. Dưới đây là một ví dụ về những gì đầu ra có thể trông như thế nào:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Gợi ý: Đối với mỗi từ trong danh sách từ, sắp xếp các chữ cái và nối chúng lại thành một chuỗi. Tạo một từ điển ánh xạ từ chuỗi đã sắp xếp này đến một danh sách các từ là từ đồng nghĩa của nó.

Các ô sau đây tải xuống `words.txt` và đọc các từ vào một danh sách.

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

```
word_list = open('words.txt').read().split()
```

Đây là hàm `sort_word` mà chúng ta đã sử dụng trước đây.

```
def sort_word(word):
    return ''.join(sorted(word))
```

Để tìm danh sách các từ đồng nghĩa dài nhất, bạn có thể sử dụng hàm sau, hàm này nhận một cặp khóa - giá trị trong đó khóa là một chuỗi và giá trị là một danh sách các từ.

Nó trả về độ dài của danh sách.

```
def value_length(pair):
    key, value = pair
    return len(value)
```

Chúng ta có thể sử dụng hàm này làm khóa sắp xếp để tìm các danh sách từ đồng nghĩa dài nhất.

```
anagram_items = sorted(anagram_dict.items(), key=value_length)
for key, value in anagram_items[-10:]:
    print(value)
```

Nếu bạn muốn biết các từ dài nhất có các từ đồng nghĩa, bạn có thể sử dụng vòng lặp sau để in một số từ đó.

```
longest = 7

for key, value in anagram_items:
    if len(value) > 1:
        word_len = len(value[0])
        if word_len > longest:
            longest = word_len
            print(value)
```

Bài tập 5

Viết một hàm có tên là `word_distance` nhận hai từ có cùng độ dài và trả về số vị trí mà hai từ khác nhau.

Gợi ý: Sử dụng `zip` để lặp qua các chữ cái tương ứng của các từ.

Đây là một dàn ý của hàm với các `doctest` mà bạn có thể sử dụng để kiểm tra hàm của mình.

```
def word_distance(word1, word2):
    """Tính số vị trí mà hai từ khác nhau.

    >>> word_distance("hello", "hxllo")
    1
    >>> word_distance("ample", "apply")
    2
    >>> word_distance("kitten", "mutton")
    3
    """
    return None
```

```
from doctest import run_docstring_examples

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

run_doctests(word_distance)
```

Bài tập 6

"Metathesis" là sự hoán đổi vị trí các chữ cái trong một từ. Hai từ tạo thành một "cặp metathesis" nếu bạn có thể chuyển đổi một từ thành từ kia bằng cách hoán đổi hai chữ cái, giống như "converse" và "conserve". Viết một chương trình tìm tất cả các cặp metathesis trong danh sách từ.

Gợi ý: Các từ trong một cặp metathesis phải là các từ đồng nghĩa của nhau.

Nguồn: Bài tập này được lấy cảm hứng từ một ví dụ tại http://puzzlers.org.

Bài tập 7

Đây là một bài tập bổ sung, không có trong sách. Nó khó hơn các bài tập khác trong chương này, vì vậy bạn có thể nhờ trợ lý ảo giúp đỡ hoặc quay lại sau khi đọc thêm một vài chương nữa.

Dưới đây là một câu đố Car Talk khác (http://www.cartalk.com/content/puzzlers):

> Từ tiếng Anh dài nhất nào mà khi ta xóa đi từng chữ cái một, các từ còn lại vẫn là từ tiếng Anh hợp lệ?
>
> Bây giờ, các chữ cái có thể được xóa từ đầu, cuối hoặc giữa, nhưng bạn không thể sắp xếp lại bất kỳ chữ cái nào. Mỗi khi bạn bỏ một chữ cái, bạn sẽ có một từ tiếng Anh khác. Nếu bạn làm như vậy, cuối cùng bạn sẽ có một chữ cái và chữ cái đó cũng sẽ là một từ tiếng Anh - một từ được tìm thấy trong từ điển. Tôi muốn biết từ dài nhất là gì và nó có bao nhiêu chữ cái.
>
> Tôi sẽ đưa cho bạn một ví dụ nhỏ: `Sprite`. Được chứ? Bạn bắt đầu với `sprite`, bạn lấy một chữ cái ra, một chữ cái từ bên trong từ, lấy chữ `r` ra, và chúng ta còn lại từ `spite`, sau đó chúng ta lấy chữ `e` ra khỏi cuối, chúng ta còn lại `spit`, chúng ta lấy `s` ra, chúng ta còn lại `pit`, `it` và `I`.

Viết một chương trình để tìm tất cả các từ có thể được rút gọn theo cách này, và sau đó tìm từ dài nhất.

Bài tập này khó hơn một chút so với hầu hết các bài tập khác, vì vậy đây là một số gợi ý:

1. Bạn có thể viết một hàm nhận một từ và tính toán danh sách tất cả các từ có thể được tạo ra bằng cách xóa một chữ cái. Đây là "con cái" của từ đó.
2. Đệ quy, một từ có thể rút gọn được nếu bất kỳ con cái nào của nó có thể rút gọn được. Là trường hợp cơ bản, bạn có thể coi chuỗi rỗng là có thể rút gọn được.
3. Danh sách từ mà chúng ta đã sử dụng không chứa các từ đơn chữ cái. Vì vậy, bạn có thể thêm "I" và "a".
4. Để cải thiện hiệu suất của chương trình, bạn có thể lưu trữ các từ đã biết là có thể rút gọn được.