```
# 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
```

Danh sách

Chương này trình bày một trong những kiểu dữ liệu tích hợp hữu ích nhất của Python, đó là danh sách. Bạn cũng sẽ tìm hiểu thêm về các đối tượng và những gì có thể xảy ra khi nhiều biến tham chiếu đến cùng một đối tượng.

Trong các bài tập ở cuối chương, chúng ta sẽ tạo một danh sách từ và sử dụng nó để tìm kiếm các từ đặc biệt như *palindrome* và *anagram*.

Danh sách là một chuỗi

Giống như một chuỗi, một danh sách là một chuỗi các giá trị. Trong một chuỗi, các giá trị là các ký tự; trong một danh sách, chúng có thể là bất kỳ loại nào. Các giá trị trong một danh sách được gọi là phần tử.

Có nhiều cách để tạo một danh sách mới; cách đơn giản nhất là đặt các phần tử trong dấu ngoặc vuông (`[]`). Ví dụ, đây là một danh sách gồm hai số nguyên.

```
numbers = [42, 123]
```

Và đây là một danh sách gồm ba chuỗi.

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

Các phần tử của một danh sách không nhất thiết phải cùng loại. Danh sách sau đây chứa một chuỗi, một số thực, một số nguyên, và thậm chí là một danh sách khác.

```
t = ['spam', 2.0, 5, [10, 20]]
```

Một danh sách nằm trong một danh sách khác được gọi là danh sách **lồng nhau**.

Một danh sách không chứa phần tử nào được gọi là danh sách rỗng; bạn có thể tạo một danh sách rỗng bằng cách sử dụng dấu ngoặc trống, `[]`.

```
empty = []
```

Hàm `len` trả về độ dài của một danh sách.

```
len(cheeses)
```

Độ dài của một danh sách rỗng là `0`.

```
len(empty)
```

Hình dưới đây cho thấy sơ đồ trạng thái cho `cheeses`, `numbers` và `empty`.

```
from diagram import make_list, Binding, Value

list1 = make_list(cheeses, dy=-0.3, offsetx=0.17)
binding1 = Binding(Value('cheeses'), list1)

list2 = make_list(numbers, dy=-0.3, offsetx=0.17)
binding2 = Binding(Value('numbers'), list2)

list3 = make_list(empty, dy=-0.3, offsetx=0.1)
binding3 = Binding(Value('empty'), list3)
```

```
from diagram import diagram, adjust, Bbox

width, height, x, y = [3.66, 1.58, 0.45, 1.2]
ax = diagram(width, height)
bbox1 = binding1.draw(ax, x, y)
bbox2 = binding2.draw(ax, x+2.25, y)
bbox3 = binding3.draw(ax, x+2.25, y-1.0)

bbox = Bbox.union([bbox1, bbox2, bbox3])
#adjust(x, y, bbox)
```

Các danh sách được biểu diễn bằng những hình hộp có từ "list" ở bên ngoài và các phần tử được đánh số của danh sách ở bên trong.

Danh sách là có thể thay đổi

Để đọc một phần tử của danh sách, chúng ta có thể sử dụng toán tử dấu ngoặc vuông. Chỉ số của phần tử đầu tiên là `0`.

```
cheeses[0]
```

Khác với chuỗi, danh sách là có thể thay đổi. Khi toán tử dấu ngoặc vuông xuất hiện ở phía bên trái của phép gán, nó xác định phần tử của danh sách sẽ được gán giá trị.

```
numbers[1] = 17
numbers
```

Phần tử thứ hai của `numbers`, từng là `123`, bây giờ là `17`.

Chỉ số danh sách hoạt động giống như chỉ số chuỗi:

- Bất kỳ biểu thức số nguyên nào cũng có thể được sử dụng làm chỉ số.
- Nếu bạn cố gắng đọc hoặc ghi một phần tử không tồn tại, bạn sẽ nhận được lỗi `IndexError`.
- Nếu chỉ số có giá trị âm, nó sẽ đếm ngược từ cuối danh sách.

Toán tử `in` hoạt động trên danh sách -- nó kiểm tra xem một phần tử nhất định có xuất hiện ở bất kỳ đâu trong danh sách hay không.

```
'Edam' in cheeses
```

```
'Wensleydale' in cheeses
```

Mặc dù một danh sách có thể chứa một danh sách khác, nhưng danh sách lồng nhau vẫn được tính là một phần tử duy nhất -- vì vậy trong danh sách dưới đây, chỉ có bốn phần tử.

```
t = ['spam', 2.0, 5, [10, 20]]
len(t)
```

Và `10` không được coi là một phần tử của `t` vì nó là một phần tử của một danh sách lồng nhau, không phải của `t`.

```
10 in t
```

Cắt danh sách

Toán tử cắt hoạt động trên danh sách giống như nó hoạt động trên chuỗi. Ví dụ dưới đây chọn phần tử thứ hai và thứ ba từ một danh sách gồm bốn chữ cái

```
letters = ['a', 'b', 'c', 'd']
letters[1:3]
```

Nếu bạn bỏ qua chỉ số đầu tiên, phần cắt sẽ bắt đầu từ đầu danh sách.

```
letters[:2]
```

Nếu bạn bỏ qua chỉ số thứ hai, phần cắt sẽ kéo dài đến cuối danh sách.

```
letters[2:]
```

Vì vậy, nếu bạn bỏ qua cả hai, phần cắt sẽ là một bản sao của toàn bộ danh sách.

```
letters[:]
```

Một cách khác để sao chép một danh sách là sử dụng hàm `list`.

```
list(letters)
```

Bởi vì `list` là tên của một hàm tích hợp sẵn, bạn nên tránh sử dụng nó làm tên biến.

Các phép toán trên danh sách

Toán tử `+` nối các danh sách lại với nhau.

```
t1 = [1, 2]
t2 = [3, 4]
t1 + t2
```

Toán tử `*` lặp lại một danh sách một số lần nhất định.

```
['spam'] * 4
```

Không có toán tử số học nào khác hoạt động với danh sách, nhưng hàm tích hợp sẵn `sum` sẽ cộng các phần tử lại với nhau.

```
sum(t1)
```

Và `min` cùng `max` tìm phần tử nhỏ nhất và lớn nhất.

```
min(t1)
```

```
max(t2)
```

Các phương thức danh sách

Python cung cấp các phương thức hoạt động trên danh sách. Ví dụ, `append` thêm một phần tử mới vào cuối danh sách:

```
letters.append('e')
letters
```

`extend` nhận một danh sách làm đối số và thêm tất cả các phần tử vào danh sách hiện tại:

```
letters.extend(['f', 'g'])
letters
```

Có hai phương thức loại bỏ phần tử khỏi danh sách. Nếu bạn biết chỉ số của phần tử mà bạn muốn, bạn có thể sử dụng `pop`.

```
t = ['a', 'b', 'c']
t.pop(1)
```

Giá trị trả về là phần tử đã bị xóa. Và chúng ta có thể xác nhận rằng danh sách đã được sửa đổi.

```
t
```

Nếu bạn biết phần tử mà bạn muốn xóa (nhưng không biết chỉ số), bạn có thể sử dụng `remove`:

```
t = ['a', 'b', 'c']
t.remove('b')
```

Giá trị trả về từ `remove` là `None`. Nhưng chúng ta có thể xác nhận rằng danh sách đã được sửa đổi.

```
t
```

Nếu phần tử bạn yêu cầu không có trong danh sách, sẽ xảy ra lỗi `ValueError`.

```
%%expect ValueError

t.remove('d')
```

Danh sách và chuỗi

Một chuỗi là một chuỗi các ký tự và một danh sách là một chuỗi các giá trị, nhưng một danh sách các ký tự không giống như một chuỗi. Để chuyển đổi từ chuỗi sang danh sách các ký tự, bạn có thể sử dụng hàm `list`.

```
s = 'spam'
t = list(s)
t
```

Hàm `list` sẽ tách một chuỗi thành các ký tự riêng lẻ. Nếu bạn muốn tách một chuỗi thành các từ, bạn có thể sử dụng phương thức `split`:

```
s = 'pining for the fjords'
t = s.split()
t
```

Một đối số tùy chọn gọi là **dấu phân cách** xác định các ký tự nào được sử dụng làm ranh giới giữa các từ. Ví dụ sau đây sử dụng dấu gạch ngang làm dấu phân cách.

```
s = 'ex-parrot'
t = s.split('-')
t
```

Nếu bạn có một danh sách các chuỗi, bạn có thể nối chúng thành một chuỗi duy nhất bằng cách sử dụng `join`. `join` là một phương thức của chuỗi, vì vậy bạn phải gọi nó trên dấu phân cách và truyền danh sách như một đối số.

```
delimiter = ' '
t = ['pining', 'for', 'the', 'fjords']
s = delimiter.join(t)
s
```

Trong trường hợp này, dấu phân cách là một ký tự khoảng trắng, vì vậy `join` đặt một khoảng trắng giữa các từ. Để nối các chuỗi mà không có khoảng trắng, bạn có thể sử dụng chuỗi rỗng, `''`, làm dấu phân cách.

Duyệt qua các phần tử trong danh sách

Bạn có thể sử dụng câu lệnh `for` để duyệt qua các phần tử của một danh sách.

```
for cheese in cheeses:
    print(cheese)
```

Ví dụ, sau khi sử dụng `split` để tạo một danh sách các từ, chúng ta có thể dùng `for` để lặp qua chúng.

```
s = 'pining for the fjords'

for word in s.split():
    print(word)
```

Một vòng lặp `for` trên một danh sách trống sẽ không bao giờ thực thi các câu lệnh thụt dòng bên trong.

```
for x in []:
    print('Điều này không bao giờ xảy ra.')
```

Sắp xếp danh sách

Python cung cấp một hàm tích hợp gọi là `sorted` để sắp xếp các phần tử của một danh sách.

```
scramble = ['c', 'a', 'b']
sorted(scramble)
```

Danh sách ban đầu không bị thay đổi.

```
scramble
```

`sorted` hoạt động với bất kỳ loại chuỗi nào, không chỉ giới hạn ở danh sách. Vì vậy, chúng ta có thể sắp xếp các chữ cái trong một chuỗi như sau.

```
sorted('letters')
```

Kết quả là một danh sách. Để chuyển danh sách thành chuỗi, chúng ta có thể dùng hàm `join`.

```
''.join(sorted('letters'))
```

Với một chuỗi rỗng làm dấu phân cách, các phần tử của danh sách sẽ được nối lại với nhau mà không có gì ở giữa chúng.

Các đối tượng và giá trị

Nếu chúng ta chạy các câu lệnh gán sau đây:

```
a = 'banana'
b = 'banana'
```

Chúng ta biết rằng `a` và `b` đều tham chiếu đến một chuỗi, nhưng không biết liệu chúng có tham chiếu đến cùng một chuỗi hay không. Có hai trạng thái có thể xảy ra, được minh họa trong hình sau.

```
from diagram import Frame, Stack

s = 'banana'
bindings = [Binding(Value(name), Value(repr(s))) for name in 'ab']
frame1 = Frame(bindings, dy=-0.25)

binding1 = Binding(Value('a'), Value(repr(s)), dy=-0.11)
binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
frame2 = Frame([binding1, binding2], dy=-0.25)

stack = Stack([frame1, frame2], dx=1.7, dy=0)


width, height, x, y = [2.85, 0.76, 0.17, 0.51]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Trong sơ đồ bên trái, `a` và `b` tham chiếu đến hai đối tượng khác nhau nhưng có cùng giá trị. Trong sơ đồ bên phải, chúng tham chiếu đến cùng một đối tượng. Để kiểm tra xem hai biến có tham chiếu đến cùng một đối tượng hay không, bạn có thể sử dụng toán tử `is`.

```
a = 'banana'
b = 'banana'
a is b
```

Trong ví dụ này, Python chỉ tạo ra một đối tượng chuỗi và cả `a` và `b` đều tham chiếu đến nó. Nhưng khi bạn tạo hai danh sách, bạn sẽ nhận được hai đối tượng khác nhau.

```
a = [1, 2, 3]
b = [1, 2, 3]
a is b
```

Vậy thì sơ đồ trạng thái trông như thế này.

```
t = [1, 2, 3]
binding1 = Binding(Value('a'), Value(repr(t)))
binding2 = Binding(Value('b'), Value(repr(t)))
frame = Frame([binding1, binding2], dy=-0.25)


width, height, x, y = [1.16, 0.76, 0.21, 0.51]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y)
# adjust(x, y, bbox)
```

Trong trường hợp này, chúng ta sẽ nói rằng hai danh sách là **tương đương**, vì chúng có cùng các phần tử, nhưng **không giống hệt nhau**, vì chúng không phải là cùng một đối tượng. Nếu hai đối tượng giống hệt nhau, chúng cũng sẽ tương đương, nhưng nếu chúng tương đương, thì không nhất thiết chúng phải giống hệt nhau.

Tham chiếu

Nếu `a` tham chiếu đến một đối tượng và bạn gán `b = a`, thì cả hai biến đều tham chiếu đến cùng một đối tượng.

```
a = [1, 2, 3]
b = a
b is a
```

Vì vậy, sơ đồ trạng thái trông như thế này.

```
t = [1, 2, 3]
binding1 = Binding(Value('a'), Value(repr(t)), dy=-0.11)
binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
frame = Frame([binding1, binding2], dy=-0.25)
```

```
width, height, x, y = [1.11, 0.81, 0.17, 0.56]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y)
# adjust(x, y, bbox)
```

Liên kết của một biến với một đối tượng được gọi là **tham chiếu**. Trong ví dụ này, có hai tham chiếu đến cùng một đối tượng.

Một đối tượng có hơn một tham chiếu có hơn một tên, vì vậy chúng ta nói rằng đối tượng đó **đã được tham chiếu**. Nếu đối tượng đã được tham chiếu có thể thay đổi, thì những thay đổi được thực hiện bằng một tên sẽ ảnh hưởng đến tên còn lại. Trong ví dụ này, nếu chúng ta thay đổi đối tượng mà `b` tham chiếu đến, thì chúng ta cũng đang thay đổi đối tượng mà `a` tham chiếu đến.

```
b[0] = 5
a
```

Vì vậy, chúng ta có thể nói rằng `a` "nhìn thấy" sự thay đổi này. Mặc dù hành vi này có thể hữu ích, nhưng nó cũng dễ gây lỗi. Nói chung, an toàn hơn khi tránh việc tham chiếu khi bạn làm việc với các đối tượng có thể thay đổi.

Đối với các đối tượng không thể thay đổi như chuỗi, việc tham chiếu không phải là vấn đề lớn. Trong ví dụ này:

```
a = 'banana'
b = 'banana'
```

Nó hầu như không bao giờ ảnh hưởng đến việc `a` và `b` có tham chiếu đến cùng một chuỗi hay không.

Tham số danh sách

Khi bạn truyền một danh sách cho một hàm, hàm đó nhận được một tham chiếu đến danh sách. Nếu hàm thay đổi danh sách, người gọi sẽ thấy sự thay đổi. Ví dụ, `pop_first` sử dụng phương thức danh sách `pop` để loại bỏ phần tử đầu tiên khỏi danh sách.

```
def pop_first(lst):
    return lst.pop(0)
```

Chúng ta có thể sử dụng nó như sau.

```
letters = ['a', 'b', 'c']
pop_first(letters)
```

Giá trị trả về là phần tử đầu tiên, đã bị xóa khỏi danh sách — như chúng ta có thể thấy bằng cách hiển thị danh sách đã được sửa đổi.

```
letters
```

Trong ví dụ này, tham số `lst` và biến `letters` là các bí danh cho cùng một đối tượng, vì vậy sơ đồ trạng thái trông như sau:

```
lst = make_list('abc', dy=-0.3, offsetx=0.1)
binding1 = Binding(Value('letters'), draw_value=False)
frame1 = Frame([binding1], name='__main__', loc='left')

binding2 = Binding(Value('lst'), draw_value=False, dx=0.61, dy=0.35)
frame2 = Frame([binding2], name='pop_first', loc='left', offsetx=0.08)

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

```
width, height, x, y = [2.04, 1.24, 1.06, 0.85]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = lst.draw(ax, x+0.5, y)
bbox = Bbox.union([bbox1, bbox2])
adjust(x, y, bbox)
```

Việc truyền một tham chiếu đến một đối tượng như một tham số cho một hàm tạo ra một hình thức tham chiếu. Nếu hàm sửa đổi đối tượng, những thay đổi đó sẽ tồn tại sau khi hàm kết thúc.

Tạo danh sách từ

Trong chương trước, chúng ta đã đọc tệp `words.txt` và tìm kiếm các từ có những thuộc tính nhất định, như sử dụng chữ cái `e`. Nhưng chúng ta đã đọc toàn bộ tệp nhiều lần, điều này không hiệu quả. Tốt hơn là đọc tệp một lần và đưa các từ vào một danh sách. Vòng lặp sau đây cho thấy cách thực hiện.

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

```
word_list = []

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

len(word_list)
```

Trước vòng lặp, `word_list` được khởi tạo với một danh sách rỗng. Mỗi lần lặp qua vòng lặp, phương thức `append` sẽ thêm một từ vào cuối danh sách. Khi vòng lặp hoàn tất, có hơn 113.000 từ trong danh sách.

Một cách khác để làm điều tương tự là sử dụng `read` để đọc toàn bộ tệp vào một chuỗi.

```
string = open('words.txt').read()
len(string)
```

Kết quả là một chuỗi đơn với hơn một triệu ký tự. Chúng ta có thể sử dụng phương thức `split` để chia nó thành một danh sách các từ.

```
word_list = string.split()
len(word_list)
```

Bây giờ, để kiểm tra xem một chuỗi có xuất hiện trong danh sách hay không, chúng ta có thể sử dụng toán tử `in`. Ví dụ, `demotic` có trong danh sách.

```
'demotic' in word_list
```

Nhưng `contrafibularities` thì không.

```
'contrafibularities' in word_list
```

Và tôi phải nói rằng, tôi cảm thấy `anaspeptic` về điều đó.

Gỡ lỗi

Điều cần lưu ý là hầu hết các phương thức danh sách sẽ sửa đổi đối số và trả về `None`. Điều này ngược lại với các phương thức chuỗi, mà trả về một chuỗi mới và để nguyên chuỗi gốc.

Nếu bạn đã quen với việc viết mã chuỗi như thế này:

```
word = 'plumage!'
word = word.strip('!')
word
```

Thật dễ dàng để viết mã danh sách như thế này:

```
t = [1, 2, 3]
t = t.remove(3)           # SAI!
```

Phương thức `remove` sửa đổi danh sách và trả về `None`, vì vậy thao tác tiếp theo mà bạn thực hiện với `t` có khả năng sẽ thất bại.

```
%%expect AttributeError

t.remove(2)
```

Thông báo lỗi này cần một chút giải thích. Một **thuộc tính** của một đối tượng là một biến hoặc phương thức gắn liền với nó. Trong trường hợp này, giá trị của `t` là `None`, đây là một đối tượng kiểu `NoneType`, không có thuộc tính nào có tên là `remove`, vì vậy kết quả là một `AttributeError`.

Nếu bạn thấy một thông báo lỗi như thế này, bạn nên nhìn lại chương trình và xem liệu bạn có thể đã gọi một phương thức danh sách không đúng cách hay không.

Thuật ngữ

list - danh sách: Một đối tượng chứa một chuỗi các giá trị.

element - phần tử: Một trong những giá trị trong một danh sách hoặc chuỗi khác.

nested list - danh sách lồng nhau: Một danh sách là một phần tử của danh sách khác.

delimiter - kí tự phân tách: Một kí tự hoặc chuỗi được sử dụng để chỉ ra vị trí mà một chuỗi nên được chia.

equivalent - tương đương: Có cùng giá trị.

identical - đồng nhất: Là cùng một đối tượng (điều này ngụ ý tính tương đương).

reference - tham chiếu: Sự liên kết giữa một biến và giá trị của nó.

aliased - được tham chiếu: Nếu có nhiều biến tham chiếu đến một đối tượng, đối tượng đó được gọi là được tham chiếu.

attribute - thuộc tính: Một trong những giá trị được đặt tên gắn liền với một đối tượng.

Bài tập

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

%xmode Verbose
```

Hỏi một trợ lý ảo

Trong chương này, tôi đã sử dụng các từ "contrafibularities" và "anaspeptic", nhưng chúng thực sự không phải là từ tiếng Anh. Chúng được sử dụng trong chương trình truyền hình của Anh có tên "Black Adder", Mùa 3, Tập 2, "Mực và Sự bất lực".

Tuy nhiên, khi tôi hỏi ChatGPT 3.5 (phiên bản ngày 3 tháng 8 năm 2023) nguồn gốc của những từ này, nó ban đầu tuyên bố rằng chúng đến từ Monty Python, và sau đó tuyên bố rằng chúng đến từ vở kịch của Tom Stoppard Rosencrantz and Guildenstern Are Dead.

Nếu bạn hỏi bây giờ, bạn có thể nhận được kết quả khác nhau. Nhưng ví dụ này nhắc nhở rằng các trợ lý ảo không phải lúc nào cũng chính xác, vì vậy bạn nên kiểm tra xem kết quả có đúng hay không. Khi bạn có thêm kinh nghiệm, bạn sẽ có cảm nhận về những câu hỏi mà các trợ lý ảo có thể trả lời một cách đáng tin cậy. Trong ví dụ này, một tìm kiếm web thông thường có thể nhanh chóng xác định nguồn gốc của những từ này.

Nếu bạn gặp khó khăn với bất kỳ bài tập nào trong chương này, hãy xem xét việc hỏi một trợ lý ảo để được giúp đỡ. Nếu bạn nhận được kết quả sử dụng các tính năng mà chúng ta chưa học, bạn có thể chỉ định cho trợ lý ảo một "vai trò".

Ví dụ, trước khi bạn đặt câu hỏi, hãy thử gõ "Vai trò: Giảng viên Lập trình Python Cơ bản". Sau đó, các phản hồi bạn nhận được nên chỉ sử dụng các tính năng cơ bản. Nếu bạn vẫn thấy các tính năng mà bạn chưa học, bạn có thể hỏi tiếp "Bạn có thể viết điều đó chỉ sử dụng các tính năng cơ bản của Python không?"

Bài tập 1

Hai từ là "hoán vị (anagram)" của nhau nếu bạn có thể sắp xếp lại các chữ cái từ một từ để tạo thành từ kia. Ví dụ, `tops` là một hoán vị của `stop`.

Một cách để kiểm tra xem hai từ có phải là hoán vị của nhau hay không là sắp xếp các chữ cái trong cả hai từ. Nếu danh sách các chữ cái đã sắp xếp giống nhau, thì các từ là hoán vị.

Viết một hàm có tên là `is_anagram` nhận vào hai chuỗi và trả về `True` nếu chúng là hoán vị.

Để giúp bạn bắt đầu, đây là một phác thảo của hàm với doctests.

```
def is_anagram(word1, word2):
    """Kiểm tra xem hai từ có phải là hoán vị cho nhau hay không.

    >>> is_anagram('tops', 'stop')
    True
    >>> is_anagram('skate', 'takes')
    True
    >>> is_anagram('tops', 'takes')
    False
    >>> is_anagram('skate', 'stop')
    False
    """
    return None
```

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

```
from doctest import run_docstring_examples

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

run_doctests(is_anagram)
```

Bạn hãy sử dụng hàm của bạn và danh sách từ để tìm tất cả các từ đồng âm của từ `takes`.

Bài tập 2

Python cung cấp một hàm tích hợp có tên là `reversed` nhận vào một chuỗi các phần tử - như danh sách hoặc chuỗi - và trả về một đối tượng `reversed` chứa các phần tử theo thứ tự ngược lại.

```
reversed('parrot')
```

Nếu bạn muốn các phần tử bị đảo ngược trong một danh sách, bạn có thể sử dụng hàm `list`.

```
list(reversed('parrot'))
```

Nếu bạn muốn chúng ở dạng chuỗi, bạn có thể sử dụng phương thức `join`.

```
''.join(reversed('parrot'))
```

Vì vậy, chúng ta có thể viết một hàm để đảo ngược một từ như sau.

```
def reverse_word(word):
    return ''.join(reversed(word))
```

Một chuỗi **đối xứng (palindrome)** là một từ được viết giống nhau từ phía sau ra phía trước và ngược lại, chẳng hạn như "noon" và "rotator". Viết một hàm có tên là `is_palindrome` nhận một đối số chuỗi và trả về `True` nếu nó là một chuỗi đối xứng và `False` nếu không phải.

Dưới đây là một phác thảo của hàm cùng với các bài kiểm tra `doctest` mà bạn có thể sử dụng để kiểm tra hàm của mình.

```
def is_palindrome(word):
    """Kiểm tra xem một từ có phải là đối xứng nhau hay không.

    >>> is_palindrome('bob')
    True
    >>> is_palindrome('alice')
    False
    >>> is_palindrome('a')
    True
    >>> is_palindrome('')
    True
    """
    return False


run_doctests(is_palindrome)
```

Bạn có thể sử dụng vòng lặp sau đây để tìm tất cả các từ đối xưng trong danh sách từ có ít nhất 7 chữ cái.

```
for word in word_list:
    if len(word) >= 7 and is_palindrome(word):
        print(word)
```

Bài tập 3

Viết một hàm có tên là `reverse_sentence` nhận vào một chuỗi chứa bất kỳ số lượng từ nào được phân tách bằng khoảng trắng. Hàm này nên trả về một chuỗi mới chứa các từ giống nhau nhưng theo thứ tự ngược lại. Ví dụ, nếu đối số là "Reverse this sentence", kết quả sẽ là "Sentence this reverse".

Gợi ý: Bạn có thể sử dụng phương thức `capitalize` để viết hoa chữ cái đầu tiên và chuyển đổi các từ còn lại thành chữ thường.

Để giúp bạn bắt đầu, dưới đây là một phác thảo cho hàm cùng với các `doctests`.

```
def reverse_sentence(input_string):
    '''Đảo ngược các từ trong một chuỗi và viết hoa chữ cái đầu tiên.

    >>> reverse_sentence('Reverse this sentence')
    'Sentence this reverse'

    >>> reverse_sentence('Python')
    'Python'

    >>> reverse_sentence('')
    ''

    >>> reverse_sentence('One for all and all for one')
    'One for all and all for one'
    '''
    return None


run_doctests(reverse_sentence)
```

Bài tập 4

Viết một hàm gọi là `total_length` nhận vào một danh sách các chuỗi và trả về tổng độ dài của các chuỗi đó. Tổng độ dài của các từ trong `word_list` nên là $902,728$.