```
# 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
```

Từ điển

Chương này trình bày một kiểu dữ liệu tích hợp sẵn gọi là **từ điển**. Đây là một trong những tính năng tốt nhất của Python - và là nền tảng của nhiều thuật toán hiệu quả và thanh lịch.

Chúng ta sẽ sử dụng từ điển để tính số lượng từ duy nhất trong một cuốn sách và số lần mỗi từ xuất hiện. Trong các bài tập, chúng ta sẽ sử dụng từ điển để giải các câu đố về từ.

Một từ điển là một phép ánh xạ

Một từ điển giống như một danh sách, nhưng tổng quát hơn. Trong một danh sách, các chỉ số phải là số nguyên; trong một từ điển, chúng có thể là (hầu như) bất kỳ loại nào. Ví dụ, giả sử chúng ta tạo một danh sách các từ số, như sau.

```
lst = ['zero', 'one', 'two']
```

Chúng ta có thể sử dụng một số nguyên làm chỉ số để lấy từ tương ứng.

```
lst[1]
```

```
'one'
```

Nhưng giả sử chúng ta muốn thực hiện theo chiều ngược lại, và tra cứu một từ để lấy số nguyên tương ứng. Chúng ta không thể làm điều đó với một danh sách, nhưng có thể làm với một từ điển. Chúng ta sẽ bắt đầu bằng cách tạo một từ điển rỗng và gán nó cho biến `numbers`.

```
numbers = {}
numbers
```

```
{}
```

Cặp dấu ngoặc nhọn `{}`, đại diện cho một từ điển rỗng. Để thêm mục vào từ điển, chúng ta sẽ sử dụng dấu ngoặc vuông `[]`.

```
numbers['zero'] = 0
```

Phép gán này thêm vào từ điển **một mục**, đại diện cho mối liên kết giữa **một khóa** và **một giá trị**. Trong ví dụ này, khóa là chuỗi `zero` và giá trị là số nguyên `0`. Nếu chúng ta hiển thị từ điển, chúng ta sẽ thấy nó chứa một mục, trong đó có một khóa và một giá trị được phân tách bởi dấu hai chấm `:`.

```
numbers
```

```
{'zero': 0}
```

Chúng ta có thể thêm nhiều mục khác như thế này.

```
numbers['one'] = 1
numbers['two'] = 2
numbers
```

```
{'zero': 0, 'one': 1, 'two': 2}
```

Bây giờ từ điển chứa ba mục.

Để tra cứu một khóa và lấy giá trị tương ứng, chúng ta sử dụng toán tử dấu ngoặc vuông.

```
numbers['two']
```

```
2
```

Nếu khóa không có trong từ điển, chúng ta sẽ nhận được một lỗi `KeyError`.

```
%%expect KeyError
numbers['three']
```

```
KeyError: 'three'
```

Hàm `len` hoạt động trên các từ điển; nó trả về số lượng mục có trong từ điển.

```
len(numbers)
```

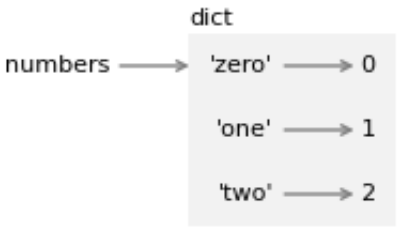
```
3
```

Trong ngôn ngữ toán học, một từ điển đại diện cho **một phép ánh xạ** từ khóa đến giá trị, vì vậy bạn cũng có thể nói rằng mỗi khóa "ánh xạ đến" một giá trị. Trong ví dụ này, mỗi từ số ánh xạ đến số nguyên tương ứng.

Hình dưới đây cho thấy sơ đồ trạng thái cho `numbers`.

```
from diagram import make_dict, Binding, Value

d1 = make_dict(numbers, dy=-0.3, offsetx=0.37)
binding1 = Binding(Value('numbers'), d1)
```

```
from diagram import diagram, adjust, Bbox

width, height, x, y = [1.83, 1.24, 0.49, 0.85]
ax = diagram(width, height)
bbox = binding1.draw(ax, x, y)
# adjust(x, y, bbox)
```



Một từ điển được biểu diễn bằng một hộp có từ "dict" bên ngoài và các mục ở bên trong. Mỗi mục được biểu diễn bởi một khóa và một mũi tên chỉ đến một giá trị. Các dấu ngoặc kép cho biết rằng các khóa ở đây là chuỗi, không phải tên biến.

Tạo từ điển

Trong phần trước, chúng ta đã tạo một từ điển rỗng và thêm các mục một cách lần lượt bằng cách sử dụng toán tử dấu ngoặc vuông. Thay vào đó, chúng ta có thể tạo từ điển tất cả cùng một lúc như sau.

```
numbers = {'zero': 0, 'one': 1, 'two': 2}
```

Mỗi mục bao gồm một khóa và một giá trị được phân tách bằng dấu hai chấm. Các mục được phân tách bằng dấu phẩy và được đặt trong cặp dấu ngoặc nhọn.

Một cách khác để tạo một từ điển là sử dụng hàm `dict`. Chúng ta có thể tạo một từ điển rỗng như sau.

```
empty = dict()
empty
```

```
{}
```

Và chúng ta có thể tạo một bản sao của một từ điển như sau.

```
numbers_copy = dict(numbers)
numbers_copy
```

```
{'zero': 0, 'one': 1, 'two': 2}
```

Thường thì việc tạo một bản sao trước khi thực hiện các phép toán sửa đổi từ điển là rất hữu ích.

Toán tử `in`

Toán tử `in` cũng hoạt động trên các từ điển; nó cho bạn biết liệu một cái gì đó có xuất hiện như một khóa trong từ điển hay không.

```
'one' in numbers
```

```
True
```

Toán tử `in` không kiểm tra xem một cái gì đó có xuất hiện như một giá trị hay không.

```
1 in numbers
```

```
False
```

Để xem một cái gì đó có xuất hiện như một giá trị trong một từ điển hay không, bạn có thể sử dụng phương thức `values`, phương thức này trả về một chuỗi các giá trị, sau đó sử dụng toán tử `in`.

```
1 in numbers.values()
```

```
True
```

Các mục trong một từ điển Python được lưu trữ trong **một bảng băm**, đây là một cách tổ chức dữ liệu có một đặc điểm đáng chú ý: toán tử `in` mất khoảng thời gian tương tự bất kể số lượng mục có trong từ điển là bao nhiêu. Điều này làm cho việc viết một số thuật toán hiệu quả một cách đáng kể trở nên khả thi.

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

```
Downloaded words.txt
```

Để minh họa, chúng ta sẽ so sánh hai thuật toán để tìm các cặp từ mà một từ là đảo ngược của từ kia — như `stressed` và `desserts`. Chúng ta sẽ bắt đầu bằng cách đọc danh sách từ.

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

```
113783
```

Và đây là hàm `reverse_word` từ chương trước.

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

Hàm sau đây lặp qua các từ trong danh sách. Đối với mỗi từ, nó đảo ngược các chữ cái và sau đó kiểm tra xem từ đảo ngược có trong danh sách từ hay không.

```
def too_slow():
    count = 0
    for word in word_list:
        if reverse_word(word) in word_list:
            count += 1
    return count
```

Hàm này mất hơn một phút để chạy. Vấn đề là toán tử `in` kiểm tra các từ trong danh sách một cách lần lượt, bắt đầu từ đầu.

Nếu nó không tìm thấy những gì mình đang tìm kiếm — điều này xảy ra hầu hết thời gian — nó phải tìm kiếm đến cuối danh sách.

```
# %time too_slow()
```

Và toán tử `in` nằm bên trong vòng lặp, vì vậy nó chạy một lần cho mỗi từ. Vì có hơn 100.000 từ trong danh sách, và đối với mỗi từ, chúng ta kiểm tra hơn 100.000 từ, tổng số phép so sánh là số từ bình phương — đại khái — điều này gần như đạt đến 13 tỷ phép so sánh.

```
len(word_list)**2
```

```
12946571089
```

Chúng ta có thể làm cho hàm này nhanh hơn rất nhiều bằng cách sử dụng một từ điển. Vòng lặp sau đây tạo ra một từ điển chứa các từ làm khóa.

```
word_dict = {}
for word in word_list:
    word_dict[word] = 1
```

Các giá trị trong `word_dict` đều là `1`, nhưng chúng có thể là bất kỳ giá trị nào, vì chúng ta sẽ không bao giờ tra cứu chúng — chúng ta chỉ sử dụng từ điển này để kiểm tra xem một khóa có tồn tại hay không.

Bây giờ đây là một phiên bản của hàm trước đó thay thế `word_list` bằng `word_dict`.

```
def much_faster():
    count = 0
    for word in word_dict:
        if reverse_word(word) in word_dict:
            count += 1
    return count
```

Hàm này mất chưa đầy một phần trăm giây, vì vậy nó nhanh hơn khoảng 10.000 lần so với phiên bản trước.

```
%time much_faster()
```

```
CPU times: user 141 ms, sys: 357 µs, total: 142 ms
Wall time: 159 ms
885
```

Nói chung, thời gian để tìm một phần tử trong một danh sách tỷ lệ với độ dài của danh sách. Thời gian để tìm một khóa trong một từ điển là gần như không thay đổi — bất kể số lượng mục có trong từ điển.

```
d = {'a': 1, 'b': 2}
d['a'] = 3
d
```

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

Một bộ đếm

Giả sử bạn được cung cấp một chuỗi và bạn muốn đếm xem mỗi chữ cái xuất hiện bao nhiêu lần. Một từ điển là công cụ tốt cho công việc này. Chúng ta sẽ bắt đầu với một từ điển rỗng.

```
counter = {}
```

Khi chúng ta lặp qua các chữ cái trong chuỗi, giả sử chúng ta thấy chữ cái `'a'` lần đầu tiên. Chúng ta có thể thêm nó vào từ điển như sau.

```
counter['a'] = 1
```

Giá trị `1` cho biết rằng chúng ta đã thấy chữ cái đó một lần. Sau đó, nếu chúng ta thấy cùng một chữ cái một lần nữa, chúng ta có thể tăng biến đếm lên như sau.

```
counter['a'] += 1
```

Bây giờ giá trị liên kết với `'a'` là `2`, vì chúng ta đã thấy chữ cái này hai lần.

```
counter
```

```
{'a': 2}
```

Hàm sau đây sử dụng những tính năng này để đếm số lần mỗi chữ cái xuất hiện trong một chuỗi.

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

Mỗi lần lặp qua vòng lặp, nếu `letter` không có trong từ điển, chúng ta tạo một mục mới với khóa `letter` và giá trị `1`. Nếu `letter` đã có trong từ điển, chúng ta tăng giá trị liên kết với `letter` lên.

Dưới đây là một ví dụ.

```
counter = value_counts('brontosaurus')
counter
```

```
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

Các mục trong bộ đếm cho thấy rằng chữ cái `'b'` xuất hiện một lần, chữ cái `'r'` xuất hiện hai lần, và cứ như vậy.

Lặp và từ điển

Nếu bạn sử dụng một từ điển trong một câu lệnh `for`, nó sẽ duyệt qua các khóa của từ điển. Để minh họa, hãy tạo một từ điển đếm số chữ cái trong chuỗi `'banana'`.

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

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

Vòng lặp sau đây in ra các khóa, là các chữ cái.

```
for key in counter:
    print(key)
```

```
b
a
n
```

Để in ra các giá trị, chúng ta có thể sử dụng phương thức `values` .

```
for value in counter.values():
    print(value)
```

```
1
3
2
```

Để in ra các khóa và giá trị, chúng ta có thể lặp qua các khóa và tra cứu các giá trị tương ứng.

```
for key in counter:
    value = counter[key]
    print(key, value)
```

```
b 1
a 3
n 2
```

Trong chương tiếp theo, chúng ta sẽ thấy một cách ngắn gọn hơn để làm điều tương tự.

Danh sách và từ điển

Bạn có thể đặt một danh sách vào một từ điển như là một giá trị. Ví dụ, đây là một từ điển ánh xạ từ số `4` đến một danh sách gồm bốn chữ cái.

```
d = {4: ['r', 'o', 'u', 's']}
d
```

```
{4: ['r', 'o', 'u', 's']}
```

Nhưng bạn không thể đặt một danh sách vào một từ điển như là một khóa. Dưới đây là điều gì sẽ xảy ra nếu chúng ta cố gắng làm như vậy.

```
%%expect TypeError
letters = list('abcd')
d[letters] = 4
```

```
TypeError: unhashable type: 'list'
```

Tôi đã đề cập trước đó rằng các từ điển sử dụng bảng băm, và điều đó có nghĩa là các khóa phải **có thể băm**.

Một **hàm băm** là một hàm nhận một giá trị (của bất kỳ loại nào) và trả về một số nguyên. Các từ điển sử dụng các số nguyên này, được gọi là giá trị băm, để lưu trữ và tra cứu các khóa.

Hệ thống này chỉ hoạt động nếu một khóa là không thay đổi, vì vậy giá trị băm của nó luôn giống nhau. Nhưng nếu một khóa là thay đổi được, giá trị băm của nó có thể thay đổi, và từ điển sẽ không hoạt động. Đó là lý do tại sao các khóa phải có thể băm và tại sao các kiểu dữ liệu thay đổi như danh sách thì không.

Vì các từ điển là thay đổi được, chúng cũng không thể được sử dụng làm khóa. Nhưng chúng có thể được sử dụng làm giá trị.

Tích lũy một danh sách

Đối với nhiều tác vụ lập trình, việc lặp qua một danh sách hoặc từ điển trong khi xây dựng một danh sách khác là rất hữu ích. Một ví dụ, chúng ta sẽ lặp qua các từ trong `word_dict` và tạo một danh sách các từ đối xứng — tức là, những từ được đánh vần giống nhau khi đọc từ trái sang phải và từ phải sang trái, như "noon" và "rotator".

Trong chương trước, một trong những bài tập yêu cầu bạn viết một hàm để kiểm tra xem một từ có phải là từ đối xứng hay không. Dưới đây là một giải pháp sử dụng hàm `reverse_word`.

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

Nếu chúng ta lặp qua các từ trong `word_dict`, chúng ta có thể đếm số lượng từ đối xứng như sau.

```
count = 0

for word in word_dict:
    if is_palindrome(word):
        count +=1

count
```

```
91
```

Bây giờ, mẫu này đã quen thuộc.

- Trước vòng lặp, biến `count` được khởi tạo bằng `0`.
- Bên trong vòng lặp, nếu từ `word` là một từ đối xứng, chúng ta tăng `count` lên.
- Khi vòng lặp kết thúc, `count` chứa tổng số lượng từ đối xứng.

Chúng ta có thể sử dụng một mẫu tương tự để tạo một danh sách các từ đối xứng.

```
palindromes = []

for word in word_dict:
    if is_palindrome(word):
        palindromes.append(word)

palindromes[:10]
```

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

Dưới đây là cách hoạt động của nó:

- Trước vòng lặp, `palindromes` được khởi tạo với một danh sách rỗng.
- Bên trong vòng lặp, nếu `word` là một từ đối xứng, chúng ta thêm nó vào cuối danh sách `palindromes`.
- Khi vòng lặp kết thúc, `palindromes` là một danh sách các từ đối xứng.

Trong vòng lặp này, `palindromes` được sử dụng như một biến tích lũy, là một biến thu thập hoặc tích lũy dữ liệu trong quá trình tính toán.

Bây giờ giả sử chúng ta muốn chọn chỉ các từ đối xứng có bảy chữ cái trở lên. Chúng ta có thể lặp qua `palindromes` và tạo một danh sách mới chỉ chứa các từ đối xứng dài.

```
long_palindromes = []

for word in palindromes:
    if len(word) >= 7:
        long_palindromes.append(word)

long_palindromes
```

```
['deified', 'halalah', 'reifier', 'repaper', 'reviver', 'rotator', 'sememes']
```

Lặp qua một danh sách như thế này, chọn một số phần tử và bỏ qua những phần tử khác, được gọi là **lọc**.

Bộ nhớ đệm

Nếu bạn đã chạy hàm `fibonacci` từ Chương 6, có thể bạn nhận thấy rằng đối số càng lớn thì hàm càng mất nhiều thời gian để chạy.

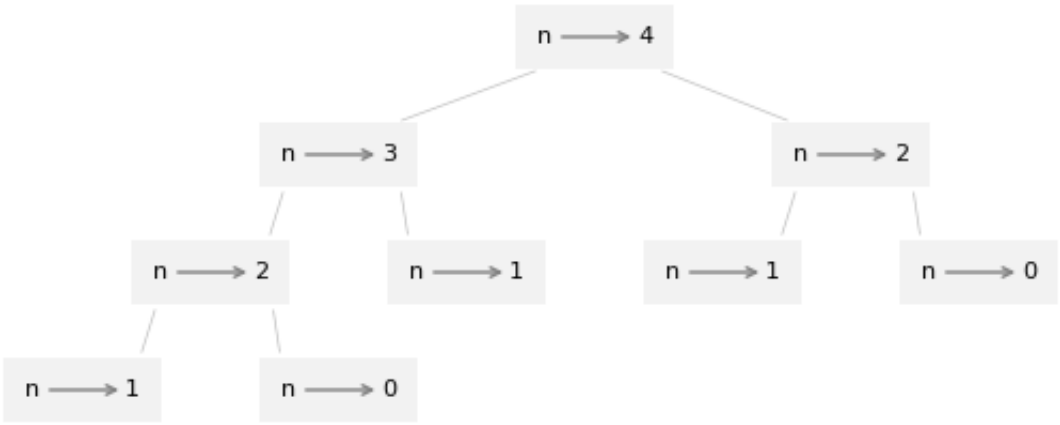
```
def fibonacci(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fibonacci(n-1) + fibonacci(n-2)
```

Hơn nữa, thời gian chạy tăng lên rất nhanh. Để hiểu lý do tại sao, hãy xem xét hình dưới đây, minh họa **đồ thị** các lệnh gọi cho hàm `fibonacci` với `n=4`.

```
from diagram import make_binding, Frame, Arrow

bindings = [make_binding('n', i) for i in range(5)]
frames = [Frame([binding]) for binding in bindings]
```

```
arrowprops = dict(arrowstyle="-", color='gray', alpha=0.5, ls='-', lw=0.5)

def left_arrow(ax, bbox1, bbox2):
    x = bbox1.xmin + 0.1
    y = bbox1.ymin
    dx = bbox2.xmax - x - 0.1
    dy = bbox2.ymax - y
    arrow = Arrow(dx=dx, dy=dy, arrowprops=arrowprops)
    return arrow.draw(ax, x, y)

def right_arrow(ax, bbox1, bbox2):
    x = bbox1.xmax - 0.1
    y = bbox1.ymin
    dx = bbox2.xmin - x + 0.1
    dy = bbox2.ymax - y
    arrow = Arrow(dx=dx, dy=dy, arrowprops=arrowprops)
    return arrow.draw(ax, x, y)


from diagram import diagram, adjust, Bbox

width, height, x, y = [4.94, 2.16, -1.03, 1.91]
ax = diagram(width, height)

dx = 0.6
dy = 0.55

bboxes = []
bboxes.append(frames[4].draw(ax, x+6*dx, y))

bboxes.append(frames[3].draw(ax, x+4*dx, y-dy))
bboxes.append(frames[2].draw(ax, x+8*dx, y-dy))

bboxes.append(frames[2].draw(ax, x+3*dx, y-2*dy))
bboxes.append(frames[1].draw(ax, x+5*dx, y-2*dy))
bboxes.append(frames[1].draw(ax, x+7*dx, y-2*dy))
bboxes.append(frames[0].draw(ax, x+9*dx, y-2*dy))

bboxes.append(frames[1].draw(ax, x+2*dx, y-3*dy))
bboxes.append(frames[0].draw(ax, x+4*dx, y-3*dy))

left_arrow(ax, bboxes[0], bboxes[1])
left_arrow(ax, bboxes[1], bboxes[3])
left_arrow(ax, bboxes[3], bboxes[7])
left_arrow(ax, bboxes[2], bboxes[5])

right_arrow(ax, bboxes[0], bboxes[2])
right_arrow(ax, bboxes[1], bboxes[4])
```

```
right_arrow(ax, bboxes[2], bboxes[6])
right_arrow(ax, bboxes[3], bboxes[8])

bbox = Bbox.union(bboxes)
# adjust(x, y, bbox)
```



Đồ thị các lệnh gọi hiển thị một tập hợp các khung hàm (function frame), với các đường nối mỗi khung với các khung của các hàm mà nó gọi. Ở đầu đồ thị, `fibonacci` với `n=4` gọi `fibonacci` với `n=3` và `n=2`. Lần lượt, `fibonacci` với `n=3` gọi `fibonacci` với `n=2` và `n=1`, và cứ tiếp tục như vậy.

Hãy đếm xem `fibonacci(0)` và `fibonacci(1)` được gọi bao nhiêu lần. Đây là một giải pháp không hiệu quả cho vấn đề, và nó sẽ càng kém hiệu quả khi đối số càng lớn.

Một giải pháp là theo dõi các giá trị đã được tính toán bằng cách lưu trữ chúng trong một từ điển. Một giá trị đã được tính toán trước đó và được lưu trữ để sử dụng lại sau này được gọi là **bộ nhớ đệm**. Dưới đây là phiên bản `fibonacci` có sử dụng "bộ nhớ đệm":

```
known = {0:0, 1:1}

def fibonacci_memo(n):
    if n in known:
        return known[n]

    res = fibonacci_memo(n-1) + fibonacci_memo(n-2)
    known[n] = res
    return res
```

`known` là một từ điển theo dõi các số `Fibonacci` mà chúng ta đã biết. Nó bắt đầu với hai mục: `0` ánh xạ đến `0` và `1` ánh xạ đến `1`.

Mỗi khi `fibonacci_memo` được gọi, hàm sẽ kiểm tra `known`. Nếu kết quả đã có sẵn, hàm có thể trả về ngay lập tức. Nếu không, hàm phải tính toán giá trị mới, thêm nó vào từ điển và trả về giá trị đó.

Khi so sánh hai hàm, `fibonacci(40)` mất khoảng `30` giây để chạy, trong khi `fibonacci_memo(40)` chỉ mất khoảng `30` micro giây, tức là nhanh hơn khoảng một triệu lần. Trong notebook của chương này, bạn sẽ thấy các phép đo này được thực hiện như thế nào.

Để đo thời gian chạy của một hàm, chúng ta có thể sử dụng `%time`, một trong các "lệnh ma thuật tích hợp" của Jupyter. Các lệnh này không phải là một phần của ngôn ngữ Python, nên chúng có thể không hoạt động trong các môi trường phát triển khác.

```
# %time fibonacci(40)
```

```
%time fibonacci_memo(40)
```

```
CPU times: user 30 µs, sys: 5 µs, total: 35 µs
Wall time: 47.4 µs
102334155
```

Gỡ lỗi

Khi làm việc với các tập dữ liệu lớn, việc gỡ lỗi bằng cách `in` và kiểm tra thủ công có thể trở nên khó khăn. Dưới đây là một số gợi ý để gỡ lỗi khi làm việc với các tập dữ liệu lớn:

1. **Thu nhỏ đầu vào**: Nếu có thể, hãy giảm kích thước của tập dữ liệu. Ví dụ,
    - Nếu chương trình đọc từ một tệp văn bản, hãy bắt đầu chỉ với `10` dòng đầu tiên hoặc với ví dụ nhỏ nhất mà bạn có thể tìm thấy. Bạn có thể chỉnh sửa trực tiếp các tệp hoặc (tốt hơn) chỉnh sửa chương trình để nó chỉ đọc `n` dòng đầu tiên.
    - Nếu xuất hiện lỗi, bạn có thể giảm `n` xuống giá trị nhỏ nhất tại đó lỗi xảy ra. Khi bạn tìm và sửa lỗi, bạn có thể tăng dần `n` lên.

2. **Kiểm tra tóm tắt và kiểu dữ liệu**:

- Thay vì `in` và kiểm tra toàn bộ tập dữ liệu, hãy xem xét `in` các tóm tắt của dữ liệu — ví dụ, số lượng phần tử trong một từ điển hoặc tổng của một danh sách số.
- Một nguyên nhân phổ biến của các lỗi trong thời gian chạy là giá trị không có đúng kiểu. Để gỡ lỗi loại lỗi này, thường chỉ cần in ra kiểu của một giá trị là đủ.

3. **Viết các kiểm tra tự động**:

- Đôi khi bạn có thể viết mã để kiểm tra lỗi tự động. Ví dụ, nếu bạn đang tính trung bình của một danh sách số, bạn có thể kiểm tra rằng kết quả không lớn hơn phần tử lớn nhất trong danh sách hoặc nhỏ hơn phần tử nhỏ nhất. Đây được gọi là "kiểm tra hợp lý" vì nó phát hiện ra các kết quả "không hợp lý".
- Một loại kiểm tra khác so sánh kết quả của hai phép tính khác nhau để xem chúng có nhất quán hay không. Điều này được gọi là "kiểm tra tính nhất quán".

4. **Định dạng đầu ra**:

- Định dạng đầu ra gỡ lỗi có thể giúp dễ dàng nhận ra lỗi hơn. Chúng ta đã thấy một ví dụ trong Chương 6. Một công cụ khác mà bạn có thể thấy hữu ích là mô-đun `pprint`, cung cấp hàm `pprint` để hiển thị các kiểu dữ liệu tích hợp theo cách dễ đọc hơn cho con người (viết tắt của "pretty print" - *in đẹp*).
- Một lần nữa, thời gian bạn dành để xây dựng cấu trúc hỗ trợ có thể giúp giảm thời gian gỡ lỗi.

Thuật ngữ

dictionary - từ điển: Một đối tượng chứa các cặp khóa-giá trị, còn gọi là các mục.

item - mục: Trong một từ điển, tên gọi khác của một cặp khóa-giá trị.

key - khóa: Một đối tượng xuất hiện trong từ điển như là phần đầu của một cặp khóa-giá trị.

value - giá trị: Một đối tượng xuất hiện trong từ điển như là phần thứ hai của một cặp khóa-giá trị. Điều này cụ thể hơn so với cách sử dụng từ "giá trị" trước đây.

mapping - mối quan hệ ánh xạ: Một mối quan hệ trong đó mỗi phần tử của một tập hợp tương ứng với một phần tử của tập hợp khác.

hash table - bảng băm: Một tập hợp các cặp khóa-giá trị được tổ chức sao cho chúng ta có thể tra cứu một khóa và tìm giá trị của nó một cách hiệu quả.

hashable - có thể băm: Các kiểu dữ liệu bất biến như số nguyên (integers), số thực (floats) và chuỗi (strings) là có thể băm. Các kiểu dữ liệu có thể thay đổi như danh sách (lists) và từ điển (dictionaries) thì không thể băm.

hash function - hàm băm: Một hàm nhận vào một đối tượng và tính toán một số nguyên dùng để xác định vị trí của một khóa trong bảng băm.

accumulator - biến tích lũy: Một biến được sử dụng trong một vòng lặp để cộng dồn hoặc tích lũy kết quả.

filtering - lọc: Lặp qua một chuỗi và chọn lọc hoặc bỏ qua các phần tử.

call graph - đồ thị các lệnh gọi: Một sơ đồ thể hiện mỗi khung hàm (frame) được tạo ra trong quá trình thực thi của một chương trình, với các mũi tên chỉ từ mỗi hàm gọi (caller) đến các hàm được gọi (callee).

memo - bộ nhớ đệm: Một giá trị đã được tính toán và lưu trữ để tránh tính toán lại trong tương lai không cần thiết.

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 nói rằng các khóa trong một từ điển phải có thể băm và đã giải thích ngắn gọn. Nếu bạn muốn biết thêm chi tiết, hãy hỏi trợ lý ảo: "Tại sao các khóa trong từ điển Python phải có thể băm?"

Trong một phần trước, chúng ta đã lưu trữ một danh sách các từ làm khóa trong một từ điển để có thể sử dụng một phiên bản hiệu quả của toán tử `in`. Chúng ta cũng có thể làm điều tương tự với một `set`, là một kiểu dữ liệu tích hợp khác. Hỏi trợ lý ảo: "Làm thế nào để tôi tạo một `set` trong Python từ một danh sách các chuỗi và kiểm tra xem một chuỗi có phải là phần tử của `set` không?"

Bài tập 1

Từ điển có một phương thức gọi là `get`, nhận một khóa và một giá trị mặc định. Nếu khóa xuất hiện trong từ điển, `get` trả về giá trị tương ứng; nếu không, nó trả về giá trị mặc định. Ví dụ, đây là một từ điển ánh xạ các chữ cái trong một chuỗi đến số lần chúng xuất hiện.

```
counter = value_counts('brontosaurus')
```

Nếu chúng ta tra cứu một chữ cái xuất hiện trong từ, phương thức `get` sẽ trả về số lần nó xuất hiện.

```
counter.get('b', 0)
```

Nếu chúng ta tra cứu một chữ cái không xuất hiện, phương thức get sẽ trả về giá trị mặc định là `0`.

```
counter.get('c', 0)
```

Sử dụng phương thức get để viết một phiên bản ngắn gọn hơn của hàm `value_counts`. Bạn nên có thể loại bỏ câu lệnh `if`.

Bài tập 2

Từ nào dài nhất mà bạn có thể nghĩ ra, trong đó mỗi chữ cái chỉ xuất hiện một lần? Hãy thử xem chúng ta có thể tìm ra từ nào dài hơn từ `unpredictably` không.

Viết một hàm có tên là `has_duplicates` nhận một dãy (như danh sách hoặc chuỗi) làm tham số và trả về True nếu có bất kỳ phần tử nào xuất hiện trong dãy nhiều hơn một lần.

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

```
def has_duplicates(t):
    """Kiểm tra xem có phần tử nào trong một dãy xuất hiện nhiều hơn một lần hay kh

    >>> has_duplicates('banana')
    True
    >>> has_duplicates('ambidextrously')
    False
    >>> has_duplicates([1, 2, 2])
    True
    >>> has_duplicates([1, 2, 3])
    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(has_duplicates)
```

Bạn có thể sử dụng vòng lặp này để tìm các từ dài nhất không có chữ cái lặp lại.

```
no_repeats = []

for word in word_list:
    if len(word) > 12 and not has_duplicates(word):
        no_repeats.append(word)

no_repeats
```

Bài tập 3

Viết một hàm có tên là `find_repeats` nhận một từ điển ánh xạ từ mỗi khóa đến một bộ đếm, giống như kết quả từ `value_counts`. Hàm này sẽ lặp qua từ điển và trả về một danh sách các khóa có giá trị đếm lớn hơn `1`. Bạn có thể sử dụng phác thảo sau để bắt đầu.

```
def find_repeats(counter):
    """Tạo danh sách các khóa có giá trị lớn hơn 1.

    counter: một từ điển ánh xạ từ khóa đến số lần xuất hiện.

    returns: danh sách các khóa.
    """
    return []
```

Bạn có thể sử dụng các ví dụ sau để kiểm tra mã của mình. Trước tiên, chúng ta sẽ tạo một từ điển ánh xạ từ các chữ cái đến các bộ đếm.

```
counter1 = value_counts('banana')
counter1
```

Kết quả từ `find_repeats` nên là `['a', 'n']`.

```
repeats = find_repeats(counter1)
repeats
```

Dưới đây là một ví dụ khác bắt đầu với một danh sách các số. Kết quả sẽ là `[1, 2]`.

```
counter1 = value_counts([1, 2, 3, 2, 1])
repeats = find_repeats(counter1)
repeats
```

Bài tập 4

Giả sử bạn chạy `value_counts` với hai từ khác nhau và lưu kết quả vào hai từ điển.

```
counter1 = value_counts('brontosaurus')
counter2 = value_counts('apatosaurus')
```

Mỗi từ điển ánh xạ từ một tập hợp các chữ cái đến số lần chúng xuất hiện. Viết một hàm có tên là `add_counters` nhận hai từ điển như vậy và trả về một từ điển mới chứa tất cả các chữ cái và tổng số lần chúng xuất hiện trong bất kỳ từ nào.

Có nhiều cách để giải quyết bài toán này. Khi bạn có một giải pháp hoạt động, hãy thử yêu cầu trợ lý ảo cho các giải pháp khác.

Bài tập 5

Một từ được gọi là "interlocking" nếu chúng ta có thể chia nó thành hai từ bằng cách lấy các chữ cái xen kẽ nhau. Ví dụ, "schooled" là một từ `interlocking` vì nó có thể được chia thành "shoe" và "cold".

Để chọn các chữ cái xen kẽ từ một chuỗi, bạn có thể sử dụng toán tử cắt với ba thành phần chỉ định nơi bắt đầu, nơi kết thúc và "bước nhảy" giữa các chữ cái.

Trong phép cắt sau, thành phần đầu tiên là `0`, vì vậy chúng ta bắt đầu từ chữ cái đầu tiên. Thành phần thứ hai là `None`, có nghĩa là chúng ta sẽ đi đến hết chuỗi. Và thành phần thứ ba là `2`, vì vậy sẽ có hai bước nhảy giữa các chữ cái chúng ta chọn.

```
word = 'schooled'
first = word[0:None:2]
first
```

Instead of providing `None` as the second component, we can get the same effect by leaving it out altogether. For example, the following slice selects alternating letters, starting with the second letter.

```
second = word[1::2]
second
```

Viết một hàm có tên là `is_interlocking` nhận một từ làm đối số và trả về `True` nếu từ đó có thể được chia thành hai từ `interlocking` .

Bạn có thể sử dụng vòng lặp sau đây để tìm các từ `interlocking` trong danh sách từ.