CHAPTER 1: DATA STRUCTURES AND ALGORITHMS <P3>

1.16. Filtering Sequence Elements

Problem:

Bạn có dữ liệu bên trong một chuỗi, bạn muốn trích xuất một số giá trị hoặc làm giảm chuỗi theo một số tiêu chí.

Solution:

Cách đơn giản nhất để thực hiện lọc một chuỗi danh sách là sử dụng một list comprehension. Ví dụ:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

Một nhược điểm tiềm năng của việc sử dụng cách này là nó có thể tạo ra một kết quả lớn nếu đầu vào lớn.

Nếu điều này là một mối quan tâm của bạn, bạn có thể sử dụng các biểu thức generator để tạo ra các giá trị được lọc lặp lại. Ví dụ:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
1
4
10
2
3
>>>
```

Thỉnh thoảng, tiêu chí lọc không dễ dàng thể hiện được trong một list comprehension hoặc biểu thức generator. Ví dụ, giả sử rằng xử lý lọc liên quan đến xử lý ngoại lệ hoặc một số chi tiết phức tạp.

Để làm điều này, đặt mã nguồn lọc vào trong chính function của nó và sử dụng hàm filter() được xây dựng sẵn.

Ví dụ:

```
values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
```

```
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']
```

filter() tạo ra một vòng lặp, Vì vậy nếu bạn muốn tạo một danh sách các kết quả, hãy chắc chắn rằng bạn sử dụng danh sách list() như đã hiển thị.

Discussion:

List comprehensions và generator expressions thường là các cách đơn giản nhất và minh bạch nhất để lọc dữ liệu. Chúng có thêm sức mạnh để chuyển đổi dữ liệu cùng một lúc.
Ví dụ:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>
```

Một sự thay đổi trong việc lọc dữ liệu liên quan đến thay thế các giá trị không đáp ứng các tiêu chí với một giá trị mới thay vì loại bỏ chúng.
Ví dụ, có lẽ thay vì chỉ tìm các giá trị dương, bạn muốn cắt các giá trị xấu để vừa với phạm vị được chỉ định. Điều này dễ dàng hoàn thành bằng cách di chuyển tiêu chí lọc vào trong biểu thức điều kiện như thế này:

```
>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Công cụ lọc đáng chú ý khác là itertools.compress(), nó mang một vòng lặp và đi cùng một chuỗi bộ chọn Boolean như đầu vào. Đầu ra, nó sẽ cung cấp cho bạn tất cả các items trong vòng lặp, trong đó phần tử tương ứng trong bộ chọn là True. Điều này là hữu ích nếu bạn cố gắng áp dụng kết quả lọc một chuỗi cho một chuỗi liên quan khác.
Ví dụ, giả sử bạn có 2 cột dữ liệu như sau:

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
```

```
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Bây giờ giả sử bạn muốn tạo một danh sách các địa chỉ tương ứng với giá trị đếm lớn hơn 5. Đây là cách mà bạn làm điều đó:

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

Chìa khoá ở đây là đầu tiên tạo một chuỗi các Booleans mà chỉ ra các phần tử thoả mãn điều kiện mong muốn. Hàm compress() sau đó sẽ chọn ra các items tương ứng với giá trị True.
Giống như filter(), compress() cũng trả về một vòng lặp. Vì vậy, bạn cần sử dụng list để trả về kết quả là một list như mong muốn.

1.17. Extracting a Subset of a Dictionary

Problem:
Bạn muốn tạo một dictionary là tập hợp con của dictionary khác.
Solution:
Điều này là dễ dàng thực hiện bằng cách sử dụng một dictionary comprehension.
Ví dụ:

```
prices = {
   'ACME': 45.23,
   'AAPL': 612.78,
   'IBM': 205.55,
   'HPQ': 37.20,
   'FB': 10.75
}

# Make a dictionary of all prices over 200
p1 = { key:value for key, value in prices.items() if value > 200 }
```

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

Discussion:

Phần lớn những gì có thể thực hiện với một dictionary comprehension có thể được thực hiện bằng cách tạo một chuỗi các tuples và truyền chúng đến function dict().
Vi dụ:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

Tuy nhiên, giải pháp dictionary comprehension có một chút rõ ràng hơn và chạy nhanh hơn một chút (nhanh gấp 2 lần khi test trên dictionary trong ví dụ trên).
Đôi khi có nhiều cách để thực hiện được cùng một vấn đề.
Ví dụ thứ 2 có thể được viết lại như sau:

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

Tuy nhiên, một nghiên cứu thời gian cho thấy rằng giải pháp này chậm 1.6 lần so với giải pháp đầu tiên.

1.18. Mapping Names to Sequence Elements

Problem:

Bạn có code truy cập các phần tử của list hoặc tuple theo vị trí, nhưng điều này làm code của bạn hơi khó đọc vào các thời điểm. Bạn không muốn phụ thuộc vào việc truy cập các phần tử theo vị trí, bạn muốn truy cập chúng bằng tên.

Solution:

collections.namedtuple() cung cấp những lợi ích này, trong khi thêm chi phí tối thiểu trên bằng cách sử dụng đối tượng tuple. collections.namedtuple() thực sự là một phương thức trả về một lớp con của loại chuẩn tuple. Bạn cho nó tên loại, và các trường mà nó nên có, và nó trả về một class mà bạn có thể khởi tạo, truyền giá trị vào các trường bạn đã định nghĩa.
Ví dụ:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
```

```
'2012-10-19'
>>>
```

Mặc dù một instance của một namedtuple nhìn giống như một instance của một lớp, nó có thể hoán đổi với một tuple và hỗ trợ tất cả các thao tác như một tuple thông thường, chẳng hạn như đánh chỉ mục hay unpacking.
Ví dụ:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

Một trường hợp sử dụng chính của việc đặt tên các tuples là để tách code của bạn khỏi vị trí của các elements nó thao tác. Vì vậy nếu bạn lấy lại một danh sách lớn các tuples từ lời gọi cơ sở dữ liệu, sau đó thao tác trên chúng bằng cách truy cập vị trí các phần tử, mã nguồn của bạn có thể bị hỏng nếu bạn thêm một cột vào bảng. Không phải vậy nếu bạn đầu tiên đưa các tuples được trả về cho namedtuple.
Để chứng minh, ví dụ dưới đây thực hiện bằng cách sử dụng tuple bình thường:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

Tham chiếu đến các yếu tố vị trí thường làm cho code của bạn ít biểu cảm hơn và bị phụ thuộc nhiều vào cấu trúc của các bản ghi.
Đây là ví dụ sử dụng một version của một namedtuple:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])

def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
```

```
        total += s.shares * s.price
    return total
```

Đương nhiên, bạn có thể tránh explicit conversion sang Stock namedtuple nếu chuỗi bản ghi trong ví dụ trên đã chứa các trường hợp như vậy.

Discussion:
Có thể sử dụng một namedtuple là một thay thế cho một dict, mà yêu cầu nhiều không gian hơn để lưu trữ. Vì vậy, nếu bạn build một cấu trúc dữ liệu lớn liên quan đến dict, sử dụng một namedtuple sẽ hiệu quả hơn nhiều. Tuy nhiên, lưu ý rằng không giống như dict, một namedtuple là không thể thay đổi.
Ví dụ:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Nếu bạn muốn thay đổi bất kỳ thuộc tính nào, có thể làm bằng cách sử dụng phương thức _replace() của instance namedtuple tạo một tệp hoàn toàn mới với các giá trị được chỉ định được thay thế.
Ví dụ:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

Một cách sử dụng tinh tế phương thức _replace() là nó có thể là một cách thuận tiện để điền các tuples được đặt tên có các trường tuỳ chọn hoặc bị thiếu. Để làm điều này, bạn phải tạo một tuple nguyên mẫu bao gồm các giá trị mặc định và sau đó sử dụng _replace() để tạo một instance mới với các giá trị được thay thế.
Ví dụ:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)
```

```
# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Đây là ví dụ về cách mã này hoạt động:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date':
'12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012',
time=None)
>>>
```

Cuối cùng, nhưng không kém phần quan trọng, nên được lưu ý rằng nếu mục đích của bạn là để định nghĩa một cấu trúc dữ liệu hiệu quả nơi bạn sẽ thay đổi các thuộc tính của instances, bằng cách sử dụng namedtuple không phải là lựa chọn tốt nhất. Thay vào đó, hãy xem xét cách định nghĩa một lớp bằng cách sử dụng __slots__ (có thể xem ở recipe 8.4 sau này).

1.19. Transforming and Reducing Data at The Same Time

Problem:

Bạn cần thực thi một hàm (Ví dụ như min(), max(), sum()...) nhưng trước khi đó bạn cần chuyển đổi hoặc lọc dữ liệu.

Solution:

Có một cách rất elegant để kết hợp việc giảm dữ liệu và chuyển đổi dữ liệu là sử dụng một tham số là một generator-expression.

Ví dụ, nếu bạn muốn tính tổng các ô vuông, hãy làm như sau:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Đây là một vài ví dụ khác:

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
```

```
    print('There be python!')
else:
    print('Sorry, no python.')

# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))

# Data reduction across fields of a data structure
portfolio = [
  {'name':'GOOG', 'shares': 50},
  {'name':'YHOO', 'shares': 75},
  {'name':'AOL', 'shares': 20},
  {'name':'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)
```

Discussion:

Giải pháp cho thấy một khía cạnh cú pháp tinh tế của generator expression khi được cung cấp như một tham số đơn cho một hàm.
Ví dụ, những câu lệnh này là giống nhau:

```
s = sum((x * x for x in nums))    # Pass generator-expr as argument
s = sum(x * x for x in nums)      # More elegant syntax
```

Bằng cách sử dụng một tham số generator thường là một cách tiếp cận hiệu quả hơn và thanh lịch hơn lần đầu tiên tạo danh sách tạm thời.
Ví dụ, nếu bạn không sử dụng một generator expressions, bạn có thể xem xét cách triển khai thay thế này:

```
nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])
```

Điều này có hoạt động nhưng nó giới thiệu một bước bổ sung và một danh sách bổ sung. Đối với danh sách nhỏ như vậy, nó có thể không có vấn đề gì, nhưng nếu nums là rất lớn, bạn sẽ kết thúc việc tạo một cấu trúc dữ liệu tạm thời lớn chỉ để sử dụng 1 lần và bỏ đi. Giải pháp generator chuyển đổi dữ liệu lặp đi lặp lại nên hiệu quả hơn nhiều cho bộ nhớ.
Một số hàm như min(), max() chấp nhận một tham số khoá mà có thể có ích trong các tình huống nơi bạn có thể có khuynh hướng sử dụng generator.
Ví dụ, ở trong ví dụ portfolio, bạn có thể xem xét giải pháp thay thế này:

```
# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)

# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])
```

1.20. Combining Multiple Mappings into a Single Mapping

Problem:

Bạn có nhiều dicts hoặc nhiều mappings và bạn muốn kết hợp chúng một cách hợp lý thành một mapping đơn để thực thi các thao tác nhất định trên chúng, ví dụ như tra cứu các giá trị, kiểm tra sự tồn tại của các khoá.

Solution:

Giả sử bạn có 2 dicts:

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

Bây giờ bạn muốn thực hiện tra cứu nơi bạn phải kiểm tra trên cả 2 dicts. Một cách đơn giản nhất để thực hiện điều này là sử dụng lớp ChainMap từ module collections.
Ví dụ:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x'])     # Outputs 1  (from a)
print(c['y'])     # Outputs 2  (from b)
print(c['z'])     # Outputs 3  (from a)
```

Discussion:

Một ChainMap mang nhiều mappings và làm chúng xuất hiện hợp lý như một. Tuy nhiên, các mappings không được hợp nhất với nhau. Thay vào đó, một ChainMap đơn giản là giữ một danh sách các danh sách cơ bản và định nghĩa lại các thao tác dictionary thông thường để quét danh sách. Nhiều thao tác sẽ hoạt động. Ví dụ:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
```

```
[1, 2, 3]
>>>
```

Nếu có các keys trùng lặp , các giá trị từ mapping đầu tiên được sử dụng. Vì vậy, mục c['z'] trong ví dụ sẽ luôn luôn tham chiếu đến giá trị trong dict a, không phải giá trị trong dict b. Các thao tác thay đổi mapping luôn ảnh hưởng đến mapping đầu tiên được liệt kê.
Ví dụ:

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

Một ChainMap đặc biệt có ích khi làm việc với các giá trị được giới hạn chẳng hạn như các biến trong một ngôn ngữ lập trình. Thực tế, có những phương pháp làm cho điều này dễ dàng:

```
>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
```

```
>>> values
ChainMap({'x': 1})
>>>
```

Như một thay thế cho ChainMap, bạn có thể xem xét trộn các dicts cùng nhau bằng cách sử dụng phương thức update().
Ví dụ:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>
```

Điều này hoạt động nhưng nó yêu cầu bạn tạo một đối tượng dict hoàn toàn tách biệt. Cũng vì vậy, nếu bất kỳ dicts ban đầu thay đổi, việc thay đổi không được reflected trong từ điển đã hợp nhất.
Ví dụ:

```
>>> a['x'] = 13
>>> merged['x']
1
```

Một ChainMap sử dụng dicts gốc, vì vậy nó không có hành vi này.
Ví dụ:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']   # Notice change to merged dicts
42
>>>
```