CHAPTER 1: DATA STRUCTURES AND ALGORITHMS <P3>

1.13. Sorting a List of Dictionaries by a Common Key

Problem:

Bạn có danh sách các dicts và bạn muốn sắp xếp chúng theo một hoặc nhiều giá trị của dicts

Solution:

Sắp xếp loại này là dễ dàng bằng cách sử dụng hàm itemgetter của module operator. Giả sử bạn có truy vấn một bảng dữ liệu để lấy ra danh sách các thành viên trong website của bạn, và bạn nhận kết quả cấu trúc dữ liệu trả lại như dưới đây:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

Khá dễ dàng để xuất những hàng này theo thứ tự bằng bất kỳ trường phổ biến nào cho tất cả các dictionaries.

Ví dụ:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

Đoạn code trước đó sẽ xuất ra như sau:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

Hàm itemgetter() có thể chấp nhận nhiều keys.

Ví dụ:

```
rows_by_lfname = sorted(rows, key=itemgetter('lname','fname'))
print(rows_by_lfname)
```

Output sẽ như thế này;

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

Discussion:

Trong ví dụ, các rows được truyền đến cho hàm sorted(), nó chấp nhận đối số key. Đối số này có thể có thể chấp nhận một item đơn từ các hàng khi nhập vào và return một giá trị mà sẽ được sử dụng cho sorting. Hàm itemgetter tạo ra một lời gọi như vậy.

Hàm operator.itemgetter() lấy làm đối số các chỉ mục tra cứu được sử dụng để trích xuất các giá trị mong muốn từ các hàng. Nó có thể là một tên khoá từ điển, một phần tử danh sách dạng số, hoặc bất kỳ giá trị mà có thể nạp được một phương thức __getitem__() của đối tượng. Nếu bạn cung cấp nhiều chỉ mục cho itemgetter(), việc gọi nó tạo ra sẽ trả về 1 tuple với tất cả các phần tử trong nó, và sorted() sẽ sắp xếp thứ tự đầu ra theo thứ tự các tuples. Điều này có thể có ích nếu bạn muốn đồng thời sắp xếp nhiều trường (chẳng hạn như họ và tên trong ví dụ).

Hàm itemgetter() thỉnh thoảng được thay thế bởi biểu thức lambda. Ví dụ:

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'],r['fname']))
```

Giải pháp này thường hoạt động tốt. Tuy nhiên, giải pháp liên quan đến itemgetter() thường chạy nhanh hơn một chút. Vì vậy bạn có thể thich hơn nếu hiệu suất là một mối quan tâm của bạn.

Cuối cùng, nhưng không kém phần quan trọng, đừng quên rằng kỹ thuật được thể hiện trong recipe này có thể được áp dụng cho các hàm chẳng hạn như min() và max(). Ví dụ:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

1.14. Sorting Objects Without Native Comparison Support

Problem:

Bạn muốn sắp xếp các đối tượng của cùng một lớp, nhưng chúng không hỗ trợ các thao tác so sánh.

Solution:

Hàm sorted() mang một tham số key mà có thể truyền cho một lời gọi sẽ trả về một số giá trị của đối tượng mà sorted() sẽ dùng nó để so sánh các đối tượng. Ví dụ, nếu bạn có một chuỗi các instances User trong ứng dụng của bạn, bạn muốn sắp xếp chúng theo thuộc tính user_id, bạn muốn cung cấp một lời gọi mà đầu vào là một instance của User và return về giá trị user_id của instance đó. Ví dụ:

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>
```

Thay vì sử dụng lambda, có thể sử dụng operator.attrgetter():

```
>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>
```

Discussion:

Việc lựa chọn sử dụng lambda hay operator.attrgetter() có thể phụ thuộc vào sở thích cá nhân. Tuy nhiên, attrgetter() nhanh hơn một chút và có thêm các tính năng cho phép nhiều trường để trích xuất đồng thời. Điều này tương tự như sử dụng operator.attrgetter() cho các dicts (có thể xem recipe 1.13). Ví dụ, instance của User có thêm các thuộc tính last name và first name, bạn có thể làm việc sắp xếp như thế này:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

Đáng lưu ý rằng kỹ thuật sử dụng trong recipe này cũng có thể cũng được áp dụng cho các hàm min() và max(). Ví dụ:

```
>>> min(users, key=attrgetter('user_id')
User(3)
>>> max(users, key=attrgetter('user_id')
```

```
User(99)
>>>
```

1.15. Grouping Records Together Based on a Field

Problem:

Bạn có một chuỗi các dicts hoặc các instances và bạn muốn lặp qua dữ liệu các nhóm dựa trên giá trị của một trường cụ thể, ví dụ như date.

Solution:

Hàm itertools.groupby() đặc biệt hữu ích cho việc nhóm các dữ liệu với nhau như thế này. Để chứng minh, giả sử bạn có danh sách các dicts như sau:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Bây giờ bạn muốn lặp qua các dữ liệu trong các nhóm được nhóm theo date. Để làm điều này, trước hết chúng ta sắp xếp theo trường mong muốn (ở ví dụ này là trường date), sau đó sử dụng hàm itertools.groupby() như sau:

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print('    ', i)
```

Output sẽ như sau:

```
07/01/2012
        {'date': '07/01/2012', 'address': '5412 N CLARK'}
        {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
    07/02/2012
        {'date': '07/02/2012', 'address': '5800 E 58TH'}
        {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
        {'date': '07/02/2012', 'address': '1060 W ADDISON'}
```

```
07/03/2012
    {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
    {'date': '07/04/2012', 'address': '5148 N CLARK'}
    {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

Discussion:

Hàm groupby() hoạt động bằng cách quét chuỗi và tìm tuần tự “runs" các giá trị giống nhau. Trong mỗi một lần lặp, nó sẽ trả về giá trị cùng với trình lặp tạo ra tất cả các mục trong một nhóm có giá trị giống nhau.

Một bước quan trọng nhất là sắp xếp dữ liệu theo trường mà mình muốn. Vì groupby() chỉ kiểm tra các items liên tục, fail trong lần sắp xếp đầu tiên sẽ không nhóm các bản ghi đúng như bạn mong muốn.

Nếu mục đích của bạn chỉ đơn giản là nhóm dữ liệu cùng nhau theo ngày thành một cấu trúc dữ liệu lớn mà cho phép bạn truy cập ngẫu nhiên, thì tốt hơn là bạn nên dùng phương thức defaultdict() để xây dựng một multidict, giống như đã mô tả ở recipe 1.6.

Ví dụ:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

Điều này cho phép các bản ghi cho mỗi ngày được truy cập dễ dàng hơn như thế này:

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

Ở ví dụ này, không cần thiết là phải sắp xếp các bản ghi trước. Vì vậy nếu bộ nhớ không là mối quan tâm của bạn, thì làm thế này sẽ nhanh hơn là bạn sắp xếp trước rồi mới dùng groupby() để lặp.