CHAPTER 1: DATA STRUCTURES AND ALGORITHMS <P2>

1.4. Finding the Largest or Smallest N Items

Problem:

Bạn muốn tạo danh sách N items có giá trị lớn nhất hoặc nhỏ nhất từ một collection.

Solution:

Module heapq có 2 functions là ***nlargest()*** và ***nsmallest()*** mà thực hiện chính xác những điều bạn muốn như ở trên.
Ví dụ:

```
import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums))  # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]
```

Cả 2 functions này đều cho phép một parameter là key mà cho phép chúng được sử dụng với nhiều dữ liệu phức tạp hơn. Ví dụ:

```
portfolio = [
  {'name': 'IBM', 'shares': 100, 'price': 91.1},
  {'name': 'AAPL', 'shares': 50, 'price': 543.22},
  {'name': 'FB', 'shares': 200, 'price': 21.09},
  {'name': 'HPQ', 'shares': 35, 'price': 31.75},
  {'name': 'YHOO', 'shares': 45, 'price': 16.35},
  {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

Discussion:

Nếu bạn muốn tạo danh sách N phần tử lớn nhất hoặc nhỏ nhất và N nhỏ hơn kích thước tổng thể của cả collection thì các chức năng này cung cấp hiệu quả vô cùng vượt trội. Như covers bên dưới, chúng work bằng cách trước tiên chuyển data thành 1 danh sách các items có thứ tự như 1 heap. Ví dụ:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
```

```
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

Tính năng quan trọng nhất của 1 heap là heap[0] luôn luôn là item nhỏ nhất. Hơn nữa, Các items tiếp theo có thể dễ dàng tìm thấy bằng phương thức ***heapq.heappop()***. Phương thức này sẽ pop ra phần tử đầu tiên và thay thế nó với item nhỏ nhất tiếp theo (một thao tác yêu cầu O(log N) thao tác với N là kích thước của heap). Ví dụ, muốn tìm 3 phần tử nhỏ nhất ta làm như sau:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

Các hàm ***nlargest()*** và ***nsmallest()*** là thích hợp nhất nếu bạn muốn tìm một số lượng các items tương đối nhỏ. Nếu đơn giản bạn muốn tìm một item nhỏ nhất hoặc lớn nhất (N=1), dùng min(), max() sẽ nhanh hơn. Tương tư nếu N chính bằng kích thước của cả collection, thường sử dụng cách nhanh hơn là sắp xếp nó trước và lấy một slice (sử dụng **sorted(items)[:N]** hoặc **sorted(items)[-N:]**). Cần được lưu ý rằng việc thực thi ***nlargest()*** và ***nsmallest()*** thực tế là adaptive trong cách nó implementation và sẽ thực hiện một số tối ưu hoá thay cho bạn (ví dụ, bằng cách sử dụng sorting nếu N gần cùng với kích thước như đầu vào).
Mặc dù nó là không cần thiết để sử dụng recipe này, việc implementation của một heap là một chủ đề nghiên cứu thú vị và đáng giá. Điều này có thể được tìm thấy trong bất kỳ cuốn sách nào về thuật toán và cấu trúc dữ liệu. Tài liệu về module **heapq** cũng được discuss chi tiết.

1.5. Implementing a Priority Queue

Problem:
Bạn muốn implements một hàng đợi mà sắp xếp các items bằng một sự ưu tiên đã cho và luôn luôn trả về item với độ ưu tiên cao nhất trong mỗi thao tác pop.

Solution:
Lớp dưới đây sử dụng module **heapq** để implement một hàng đợi ưu tiên đơn giản:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0
```

```
    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Đây là một ví dụ về cách mà nó được sử dụng:

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Quan sát thấy thao tác pop() đầu tiên trả về item có độ ưu tiên cao nhất. Quan sát với 2 phần tử có độ ưu tiên như nhau (foo và grok) được return đúng thứ tự mà nó được chèn vào queue trước đó.

Discussion:

Cốt lõi của recipe này có liên quan đến việc sử dụng **heapq** module. Các functions **heapq.heappush()** và **heapq.heappop()** để chèn và remove các items từ một list **_queue** theo cách sao cho phần tử đầu tiên trong list có độ ưu tiên thấp nhất (đã được đề cập trong recipe 1.4). Phương thức **heappop**() luôn luôn trả về item nhỏ nhất, vì vậy đó là từ khoá để làm cho hàng đợi pop ra đúng các items.

Hơn thế nữa vì các thao tác push và pop có độ phức tập O(log N) với N là số items của heap, vì vậy chúng khá hiệu quả kể cả đối với các giá trị khá lớn của N.

Trong recipe này, hàng đợi bao gồm tuples có form **(-priority, index, item).** Giá trị priority bị phủ nhận để get hàng đợi sắp xếp các items từ độ ưu tiên cao nhất xuống độ ưu tiên thấp

nhất. Điều này ngược với thứ tự heap thông thường là sắp xếp các giá trị từ thấp nhất đến cao nhất.
Vai trò của biến **index** là để sắp xếp đúng các items đúng với độ ưu tiên. Bằng cách giữ một index tăng liên tục, các items sẽ được sắp xếp theo thứ tự mà chúng được chèn. Tuy nhiên, index cung cấp một vai trò quan trọng là làm cho việc so sánh work với các items có độ ưu tiên giống nhau.
Xem xét kĩ hơn điều này, chúng ta có thể xét các Item trong ví dụ trên như sau:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Nếu bạn tạo tuples (priority, item), chúng có thể được so sánh miễn là khác độ ưu tiên. Tuy nhiên nếu 2 tuples với độ ưu tiên giống nhau được so sánh, thao tác so sánh cũng failed như trên. Ví dụ:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

Bằng cách sử dụng index bổ sung và tạp tuples (priority, index, item), bạn có thể tránh vấn đề này hoàn toàn vì sẽ không có 2 tuples nào giống nhau đối với 2 index có giá trị giống nhau. Ví dụ:

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
```

```
True
>>>
```

1.6. Mapping Keys to Multiple Values in a Dictionary

Problem:

Bạn muốn tạo một dictionary để map các keys cho nhiều hơn 1 value

Solution:

Một dict là một mapping mà mỗi key được map một value. Nếu bạn muốn map các keys cho multi-value, bạn cần lưu các value ấy trong một container khác ví dụ như một list hoặc một set. Ví dụ, bạn có thể tạo các dict như thế này:

```
d = {
  'a' : [1, 2, 3],
  'b' : [4, 5]
}

e = {
  'a' : {1, 2, 3},
  'b' : {4, 5}
}
```

Việc chọn sử dụng list hay set phụ thuộc vào mục đích sử dụng. Sử dụng list nếu bạn muốn giữ đúng thứ tự các items như khi được chèn. Sử dụng set nếu bạn muốn không lặp các item(hoặc đơn giản là không care về thứ tự).

Để dễ dàng xây dựng các dict như thế, bạn có thể sử dụng defaultdict trong module collections. Một tính năng của default là tự động khởi tạo giá trị đầu tiên vì vậy bạn có thể focus vào adding items. Ví dụ:

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
```

```
d['b'].add(4)
...
```

defaultdict sẽ tự động tạo ra các entries dict cho các keys được access sau đó (ngay cả khi chúng không được tìm thấy trong dict). Nếu bạn không muốn có hành vi này, bạn phải sử dụng **setdefault**(). Ví dụ:

```
d = {}    # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...
```

Tuy nhiên nhiều lập trình viên thấy setdefault() có chút không tự nhiên - không đề cập đến thực tế là nó luôn luôn tạo một instance mới của các giá trị đã init trên mỗi lần gọi. (list [ ] empty là một ví dụ).

Discussion:

Theo nguyên tắc, xây dựng dict đa giá trị là đơn giản. Tuy nhiên, việc khởi tạo giá trị đầu tiên có thể lộn xộn nếu bạn cố tự làm điều đó. Ví dụ, bạn có đoạn code như thế này:

```
d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)
```

Bằng cách sử dụng một defaultdict đơn giản làm cho code clean hơn nhiều:

```
d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)
```

Recipe này liên quan mạnh mẽ đến vấn đề của việc nhóm các records cùng nhau trong vấn đề xử lý data. Chúng ta có thể xem một ví dụ trong recipe 1.15 sau này.