CHAPTER 1: DATA STRUCTURES AND ALGORITHMS <P3>

1.10. Removing Duplicates from a Sequence while Maintaining Order

Problem:

Bạn muốn loại bỏ các giá trị duplicates trong một chuỗi, nhưng vẫn giữ đúng thứ tự của các phần tử còn lại.

Solution:

Nếu các giá trị trong chuỗi là có thể băm, vấn đề có thể dễ dàng được giải quyết bằng cách sử dụng một set và một generator:

Ví dụ:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Đây là một ví dụ về cách sử dụng function của bạn:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

Điều này chỉ làm việc nếu các items trong chuỗi là có thể băm. Nếu bạn cố gắng loại từ các phần từ lặp trong một chuỗi không thể băm (ví dụ như dict), bạn có thể tạo ra một chút thay đổi nhỏ cho recipe này như sau:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Ở đây, giả sử tham số key là để chỉ định một hàm mà chuyển chuỗi các items thành một loại có thể băm cho mục đích phát hiện trùng lặp. Đây là cách nó làm việc:

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
```

```
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

Giải pháp sau này cũng hoạt động tốt nếu bạn muốn loại bỏ các trùng lặp dựa trên giá trị của một trường đơn hoặc thuộc tính hoặc cấu trúc dữ liệu dài hơn.

Discussion:

Nếu tất cả những gì bạn muốn làm là loại bỏ các trùng lặp, nó thường dễ dàng đủ để tạo một set.

Ví dụ:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

Tuy nhiên, cách tiếp cận này không giữ nguyên bất kỳ loại thứ tự nào. Vì vậy, kết quả dữ liệu sẽ bị lộn xộn sau đó. Giải pháp được trình bày tránh điều này.

Việc sử dụng một hàm generator trong recipe này phản ánh thực tế rằng bạn có thể muốn chức năng là mục đích cực kỳ chung chung -- không cần thiết phải gắn trực tiếp vào việc xử lý danh sách.

Ví dụ, nếu bạn muốn đọc 1 file, loại bỏ trùng lặp các dòng, bạn có thể làm đơn giản như thế này:

```
with open(somefile,'r') as f:
    for line in dedupe(f):
        ...
```

Đặc điểm chính của một hàm key bắt chước tương tự chức năng trong các hàm xây dựng sẵn như sorted(), min(), max(). Thể hiện điều này, có thể xem recipe 1.8 và 1.13.

1.11. Naming a Slice

Problem:

Chương trình của bạn trở nên một mớ hỗn độn không thể đọc được và bạn muốn clean up nó.

Solution:

Giả sử bạn có một số đoạn code kéo các trường dữ liệu ra khỏi chuỗi bản ghi với các trường cố định:

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '....................100        .......513.25     ..........'
cost = int(record[20:32]) * float(record[40:48])
```

Thay vì làm điều đó, tại sao không đặt tên các slices như thế này?

```
SHARES = slice(20,32)
PRICE  = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

Trong version sau đó, bạn tránh được rất nhiều chỉ số mã hoá bí ẩn, và bạn đang làm nó trở nên rõ ràng hơn.

Discussion:

Như một quy luật chung, viết code với rất nhiều giá trị index hardcoded dẫn đến một mớ hỗn độn khó đọc và bảo trì. Ví dụ, nếu bạn quay lại code mà bạn đã viết 1 năm trước, bạn nhìn vào nó và tự hỏi bạn đang nghĩ gì khi viết về nó. Giải pháp được thể hiện chỉ đơn giản là một cách rõ ràng hơn cho biết thực sự đoạn code của bạn đang làm gì.
Nhìn chung, hàm slice() xây dựng sẵn tạo ra một đối tượng slice mà có thể được sử dụng bất cứ nơi đâu một slice được cho phép.
Ví dụ:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

Nếu bạn có một instance của slice là s, bạn có thể lấy ra nhiều thông tin về nó bằng cách nhìn vào các thuộc tính của nó như s.start, s.stop, s.step. Ví dụ:

```
>>> a = slice(5, 50, 2)
>>> a.start
5
>>> a.stop
50
>>> a.step
2
>>>
```

Thêm vào đó bạn có thể map một slice trên một chuỗi có kích thước cụ thể bằng cách sử dụng phương thức indices(size) của nó. Điều này trả về một tuple (start, stop, step) nơi tất cả các giá trị đã được giới hạn phù hợp để vừa với giới hạn (để tránh IndexError exception khi đánh chỉ mục. Ví dụ:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
d
>>>
```

1.9. Determining the Most Frequently Occurring Items in a Sequence

Problem:

Bạn có một chuỗi các items, bạn muốn xác định các items xuất hiện nhiều nhất trong chuỗi.

Solution:

Lớp collections.Counter được thiết kế cho một vấn đề như vậy. Nó thậm chí còn đi kèm với một phương thức most_common() sẽ cho bạn câu trả lời.

Để minh hoạ điều này, giả sử bạn có danh sách các words và bạn muốn tìm ra các words xuất hiện nhiều nhất trong danh sách ấy.

Đây là cách bạn sẽ làm điều đó:

```
words = [
  'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
  'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
  'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
  'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

Discussion:

Đối tượng Counter có thể nạp bất kỳ chuỗi các items có thể băm. Ở covers bên dưới, một Counter là một dictionary map các items với số lần xuất hiện của nó.
Ví dụ:

```
>>> word_counts['not']
1
```

```
>>> word_counts['eyes']
8
>>>
```

Nếu bạn muốn tăng số lượng theo cách thủ công, đơn giản là chỉ cần sử dụng bổ sung:

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
...     word_counts[word] += 1
...
>>> word_counts['eyes']
9
>>>
```

Hoặc cách khác, có thể sử dụng phương thức update():

```
>>> word_counts.update(morewords)
>>>
```

Một tính năng ít được biết của Counter instances là chúng có thể dễ dàng được kết hợp bằng cách sử dụng nhiều thao tác tính toán.
Ví dụ:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
        "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
        'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
        'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
        'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
        "you're": 1, "don't": 1, 'under': 1})
>>>
```

Không cần nói nhiều, đối tượng Counter là một công cụ có ích rất nhiều cho hầu hết các loại vấn đề nơi bạn cần lập bảng và đếm dữ liệu. Bạn nên sử dụng điều này hơn là các giải pháp thủ công các vấn đề liên quan đến dictionaries.