CHAPTER 1: DATA STRUCTURES AND ALGORITHMS

Python cung cấp các cấu trúc dữ liệu có sẵn hữu ích, như list, set, dictionaries. Nói chung, cách sử dụng những cấu trúc này là khá đơn giản. Tuy nhiên các câu hỏi phổ biến liên quan đến searching, sorting, ordering and filtering thường xuất hiện. Vì vậy trong chap này chúng ta sẽ discuss vào các cấu trúc dữ liệu và giải thuật phổ biến liên quan đến dữ liệu.

1.1. Unpacking a sequence into separate variables

Problem:

Bạn có 1 phần tử hoặc chuỗi phần tử N và bạn muốn unpack vào 1 collection gồm N variables.

Solution:

Bất kì chuỗi nào đều có thể được unpack thành các variables bằng cách sử dụng toán tử gán đơn giản. Yêu cầu duy nhất là số các variables và cấu trúc khớp với chuỗi. Ví dụ:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>
```

Còn nếu có một mismatch về số lượng các elements, bạn sẽ nhận được một lỗi:

```
>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>
```

Discussion:

- Unpacking works với bất kì đối tượng nào mà có thể iterable, không chỉ là tuples hay lists. Điều này có nghĩa là bao gồm cả strings, files, iterators, and generators. Ví dụ:

```
>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
```

```
>>> b
'e'
>>> e
'o'
>>>
```

- Khi unpacking, bạn có thể muốn loại bỏ 1 số các giá trị. Python không có cú pháp đặc biệt cho điều này, nhưng chúng ta có thể dùng một tên biến throwaway cho nó. Ví dụ:

```
>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>
```

- Tuy nhiên hãy đảm bảo tên biến bạn chọn chưa được sử dụng cho một cái gì đó khác rồi.

1.2. Unpacking Elements from Iterables of Arbitrary length

Problem:

Bạn muốn unpack N phần tử từ 1 iterable, nhưng iterables có thể dài hơn N phần tử, gây ra exception là quá nhiều các giá trị để giải nén.

Solution:

Python “star expressions” có thể được sử dụng để giải quyết vấn đề này. Ví dụ như bạn đang chạy một khoá học và vào cuối học kỳ, bạn quyết định bỏ bài tập về nhà đầu tiên và cuối cùng và chỉ làm trung bình phần còn lại của chúng. Nếu như bạn chỉ có 4 bài tập thì điều này đơn giản nhưng ví dụ bạn có 24 bài tập thì sao? Một star expression sẽ làm cho việc này dễ dàng:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

Một trường hợp sử dụng khác, giả sử bạn có các bản ghi người dùng bao gồm tên, địa chỉ email và theo sau đó là một số điện thoại tuỳ ý. Bạn có thể unpack các bản ghi như thế này:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212',
'847-555-1212')
>>> name, email, *phone_numbers = record
>>> name
```

```
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

Lưu ý rằng, biến phone_numbers sẽ luôn là 1 list, bất kể là có bao nhiêu số điện thoại được unpacked (including none). Vì vậy, bất cứ đoạn mã nào sử dụng phone_numbers đều không cần kiểm tra xem nó có phải là list nữa hay không.

Biến được gắn sao cũng có thể là biến đầu tiên trong list. Ví dụ:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

Discussion:

- Chúng ta sẽ mở rộng cho unpacking các iterables chưa biết hoặc có độ dài tuỳ ý. Thông thường những iterables này đã biết một số component hoặc pattern trong cấu trúc của chúng và star unpacking cho phép các developers tận dụng những patterns này để dễ dàng thay vì thực hiện acrobatics để lấy các phần tử trong iterable.

Lưu ý rằng cú pháp star có thể đặc biệt có ích khi iterating qua một chuỗi các tuples có thay đổi độ dài. Ví dụ, có một chuỗi được gắn các tuples:

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

- Star unpacking cũng có ích khi kết hợp với một số loại thao tác xử lí chuỗi, ví dụ như tách.
Ví dụ:

```
>>> line = 'nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

- Đôi khi bạn muốn unpack các giá trị và vứt chúng đi. Bạn sẽ cần sử dụng một star unpacking nó và 1 biến throwaway phổ biến như _
Ví dụ:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_, (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

- Có một điểm tương đồng nhất định giữa star unpacking và các thuộc tính xử lí danh sách. Ví dụ, nếu bạn có 1 list, bạn có thể dễ dàng tách nó vào head và tail như thế này:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

- Chúng ta có thể tưởng tượng được rằng việc viết các hàm thực hiện việc tách như vậy là để thực hiện một số thuật toán đệ quy thông minh. Ví dụ:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
```

```
...
>>> sum(items)
```

Tuy nhiên chúng ta ý thức được rằng đệ quy thực sự không phải là một tính năng mạnh của Python do sự giới hạn đệ quy vốn có. Vì vậy, ví dụ cuối cùng này có thể chẳng là gì ngoài việc tò mò về mặt học thuật trong thực tế.

1.3. Keeping the Last N Items

Problem:

Bạn muốn keep một lịch sử có giới hạn của một vài items cuối nhìn thấy trong suốt quá trình lặp hoặc một số loại xử lý khác.

Solution:

Keeping một lịch sử có giới hạn là một cách sử dụng hoàn hảo đối với một collections.deque. Ví dụ, đoạn code dưới đây thực thi một simple text match trên một chuỗi các dòng và tạo dòng phù hợp cùng với N lines of context khi tìm thấy:

```
from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('-'*20)
```

Discussion:

Khi viết code để search các items, nó thường sử dụng một hàm generator liên quan đến yield, như đã thể hiện trong solution của recipe này. Điều này decouples quá trình tìm kiếm từ code sử dụng các results. Nếu bạn mới dùng generators, có thể xem ở đây: https://www.safaribooksonline.com/library/view/python-cookbook-3rd/9781449357337/ch04.html#generators

Bằng cách sử dụng deque(maxlen=N) để tạo một hàng đợi có kích thước cố định. Khi các items mới được thêm và hàng đợi full, item cũ nhất (có thể hiểu là item được thêm đầu tiên sẽ tự động bị removed. Ví dụ:

```
>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)
```

Mặc dù bạn có thể thực hiện thủ công các thao tác như vậy trên một list (ví dụ: appeding, deleting, etc…), nhưng giải pháp hàng đợi queue thanh lịch hơn và chạy nhanh hơn nhiều. Tổng quát hơn, một hàng đợi có thể được sử dụng mỗi khi bạn cần một cấu trúc hàng đợi đơn giản. Nếu bạn không truyền cho nó một maximum size, bạn sẽ có một hàng đợi không bị chặn, cho phép bạn append và pop ở cả 2 đầu của nó. Ví dụ:

```
>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4
```

Adding và Popping items từ cả 2 đầu của nó có độ phức tạp là O(1). Điều này không giống một list khi chèn hoặc remove items từ list có độ phức tạp là O(n).