CHAPTER 2: STRINGS AND TEXT

2.7. Specifying a Regular Expression for the Shortest Match

Problem:

Bạn đang muốn match một mẫu văn bản bằng cách sử dụng biểu thức chính quy, nhưng nó đang xác định các kết quả phù hợp dài nhất có thể có của một mẫu. Thay vào đó bạn muốn thay đổi nó để tìm phù hợp ngắn nhất có thể.

Solution:

Vấn đề này thường xảy ra trong các mẫu để match văn bản được đính kèm bên trong một cặp dấu phân cách bắt đầu và kết thúc (ví dụ: chuỗi được trích dẫn). Để minh hoạ, xem xét ví dụ này:

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

Trong ví dụ này, mẫu r'\"(.*)\"' đang được dùng để match văn bản được đính kèm bên trong dấu ngoặc kép. Tuy nhiên, toán tử * trong một biểu thức chính quy là tham lam, vì vậy matching dựa trên tìm kiếm phù hợp dài nhất có thể. Vì vậy, trong ví dụ thứ hai liên quan đến text2, nó khớp không chính xác với 2 chuỗi được trích dẫn.
Để sửa điều này, thêm toán tử ? bên sau toán tử * trong mẫu, như thế này:

```
>>> str_pat = re.compile(r'\"(.*?)\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

Điều này làm cho việc matching không tham lam, chỉ sinh ra khớp ngắn nhất có thể.

Discussion:

Recipe này giải quyết một trong nhiều vấn đề phổ biến đã gặp khi viết biểu thức chính quy liên quan đến kí tự (.). Trong một mẫu, dấu chấm khớp với bất kỳ kí tự nào trừ xuống dòng. Tuy nhiên, nếu bạn gắn dấu chấm với bắt đầu và kết thúc văn bản (chẳng hạn như một trích dẫn), việc matching sẽ tìm khớp dài nhất có thể cho mẫu. Điều này làm cho nhiều lần xuất hiện của văn bản bắt đầu hoặc kết thúc bị bỏ qua hoàn toàn và được bao gồm trong kết quả

của khớp dài nhất. Thêm toán tử ? ngay sau toán tử * hoặc + bắt buộc thuật toán matching tìm kiếm kết quả match ngắn nhất có thể.

2.8. Writing a Regular Expression for Multiline Patterns

Problem:

Bạn muốn match một khối văn bản bằng cách sử dụng biểu thức chính quy nhưng bạn cần match cho mở rộng nhiều dòng.

Solution:

Vấn đề này thường phát sinh trong các mẫu sử dụng dấu chấm (.) để match bất kỳ kí tự nào nhưng quên tính đến thực tế là nó không khớp với các dòng mới. Ví dụ, giả sử bạn đang cố match các nhận xét kiểu C:

```
>>> comment = re.compile(r'/\*(.*?)\*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
...           multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

Để sửa vấn đề này, bạn có thể thêm hỗ trợ cho các dòng mới.
Ví dụ:

```
>>> comment = re.compile(r'/\*((?:.|\n)*?)\*/')
>>> comment.findall(text2)
[' this is a\n           multiline comment ']
>>>
```

Trong mẫu này, (?:.|\n) xác định một nhóm không captures (ví dụ, nó định nghĩa một nhóm cho mục đích matching, nhưng mà nhóm này không được ghi riêng hoặc đánh số).

Discussion:

Hàm re.compile() chấp nhận một cờ là re.DOTALL, có ích tại đây. Nó làm cho dấu chấm (.) trong một biểu thức chính quy khớp với tất cả các kí tự, bao gồm các dòng mới.
Ví dụ:

```
>>> comment = re.compile(r'/\*(.*?)\*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\n            multiline comment ']
```

Bằng cách sử dụng cờ re.DOTALL, nó hoạt động tốt trong các trường hợp đơn giản, nhưng có thể có vấn đề nếu bạn đang làm việc với các mẫu cực kỳ phức tạp hoặc một trộn lẫn các biểu thức chính quy mà được phối hợp cùng nhau cho mục đích mã thông báo, như được mô tả trong recipe 2.18. Nếu được lựa chọn, nó thường tốt hơn để định nghĩa mẫu biểu thức chính quy của bạn vì vậy nó hoạt động đúng mà không cần thêm các cờ bổ sung.

2.9. Normalizing Unicode Text to a Standard Representation

Problem:

Bạn đang làm việc với các chuỗi Unicode, nhưng bạn cần đảm bảo rằng tất cả các chuỗi có cùng diễn tả cơ bản.

Solution:

Trong Unicode, các kí tự nhất định có thể được diễn tả bởi nhiều hơn một chuỗi mã hợp lệ. Để minh hoạ, xem ví dụ dưới đây:

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

Đoạn văn bản ở đây “Spicy Jalapeño” đã được trình bày với 2 định dạng. Đầu tiên sử dụng kí tự “ñ” đầy đủ (U+00F1). Thứ hai sử dụng chữ cái Latin “n" theo sau là “~” (U+0303).
Có nhiều biểu diễn là một vấn đề đối với các chương trình so sánh các chuỗi. Để sửa điều này, bạn nên bình thường hoá đoạn văn bản thành một biểu diễn tiêu chuẩn bằng cách sử dụng module unicodedata:

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
```

```
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\xf1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

Tham số đầu tiên của normalize() xác định cách bạn muốn chuỗi được bình thường hoá. NFC có nghĩa là các ký tự phải được tạo thành hoàn toàn (tức là, sử dụng một điểm mã duy nhất nếu có thể). NFD có nghĩa là các ký tự phải được phân tách hoàn toàn với việc sử dụng các ký tự kết hợp.
Python cũng hỗ trợ các định dạng bình thường hoá NFKC và NFKD, thêm các tính năng tương thích bổ sung để xử lý một số loại ký tự nhất định.
Ví dụ:

```
>>> s = '\ufb01'   # A single character
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'

# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

Discussion:
Việc bình thường hoá là một phần quan trọng của bất kỳ đoạn mã nào cần để đảm bảo rằng nó xử lý văn bản Unicode một cách lành mạnh và nhất quán. Điều này đặc biệt đúng khi xử lý các chuỗi nhận được như là phần nhập vào của người dùng nơi bạn có ít kiểm soát mã hoá.
Việc bình thường hoá cũng có thể là một phần quan trọng của việc vệ sinh và lọc văn bản. Ví dụ, giả sử bạn muốn xoá tất cả các dấu phụ từ một số văn bản (có thể cho mục đích tìm kiếm hoặc matching):

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

Ví dụ cuối cùng này hiển thị khía cạnh quan trọng khác của module unicodedata - cụ thể là, các hàm hữu ích cho việc kiểm tra các ký tự đối với các lớp ký tự. Hàm combining() kiểm tra một ký tự để xem nó có phải là một ký tự kết hợp hay không. Có các hàm khác trong module để tìm kiếm các danh mục, kiểm tra các số,...
Unicode rõ ràng là một chủ đề lớn. Để biết thêm thông tin tham khảo chi tiết về việc bình thường hoá, xem Unicode's page on the subject (http://www.unicode.org/faq/normalization.html). Ned Batchelder cũng có một bài trình bày tuyệt vời về các vấn đề xử lý Unicode trong Python tại trang web của anh ta (https://nedbatchelder.com/text/unipain.html).