2.18 字符串令牌解析¶

问题¶

你有一个字符串，想从左至右将其解析为一个令牌流。

解决方案¶

假如你有下面这样一个文本字符串：

```
text = 'foo = 23 + 42 * 10'
```

为了令牌化字符串，你不仅需要匹配模式，还得指定模式的类型。 比如，你可能想将字符串像下面这样转换为序列对：

```
tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', '10')]
```

为了执行这样的切分，第一步就是像下面这样利用命名捕获组的正则表达式来定义所有可能的令牌，包括空格：

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

在上面的模式中， `?P<TOKENNAME>` 用于给一个模式命名，供后面使用。

下一步，为了令牌化，使用模式对象很少被人知道的 `scanner()` 方法。 这个方法会创建一个 `scanner` 对象， 在这个对象上不断的调用 `match()` 方法会一步步的扫描目标文本，每步一个匹配。 下面是演示一个 `scanner` 对象如何工作的交互式例子：

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

实际使用这种技术的时候，可以很容易的像下面这样将上述代码打包到一个生成器中：

```
def generate_tokens(pat, text):
    Token = namedtuple('Token', ['type', 'value'])
    scanner = pat.scanner(text)
```

```
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)
# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

如果你想过滤令牌流，你可以定义更多的生成器函数或者使用一个生成器表达式。 比如，下面演示怎样过滤所有的空白令牌：

```
tokens = (tok for tok in generate_tokens(master_pat, text)
          if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

讨论¶

通常来讲令牌化是很多高级文本解析与处理的第一步。 为了使用上面的扫描方法，你需要记住这里一些重要的几点。第一点就是你必须确认你使用正则表达式指定了所有输入中可能出现的文本序列。 如果有任何不可匹配的文本出现了，扫描就会直接停止。这也是为什么上面例子中必须指定空白字符令牌的原因。

令牌的顺序也是有影响的。 `re` 模块会按照指定好的顺序去做匹配。 因此，如果一个模式恰好是另一个更长模式的子字符串，那么你需要确定长模式写在前面。比如：

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

第二个模式是错的，因为它会将文本<=匹配为令牌LT紧跟着EQ，而不是单独的令牌LE，这个并不是我们想要的结果。

最后，你需要留意下子字符串形式的模式。比如，假设你有如下两个模式：

```
PRINT = r'(?P<PRINT>print)'
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'

master_pat = re.compile('|'.join([PRINT, NAME]))

for tok in generate_tokens(master_pat, 'printer'):
    print(tok)

# Outputs :
# Token(type='PRINT', value='print')
# Token(type='NAME', value='er')
```

关于更高阶的令牌化技术，你可能需要查看 PyParsing 或者 PLY 包。 一个调用PLY的例子在下一节会有演示。