链滴

Python-CookBook：37、文本分词

作者：zhaolixiang
原文链接：https://ld246.com/article/1566887427163
来源网站：链滴

问题

我们有一个字符串，想从左到右将它解析为标记流（stream of tokens）。

解决方案

假设有如下的字符串文本：

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

要对字符串做分词处理，需要做的不仅仅只是匹配模式。我们还需要有某种方法来识别出模式的类型例如，我们可能想将字符串转换为如下的序列对：

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

要完成这样的分词处理，第一步是定义出所有可能的标记，包括空格。这可以通过正则表达式中的命捕获组来实现，示例如下：

```
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]))
```

在这些正则表达式模式中，形如?P<TOKENNAME>这样的约定是用来将名称分配给该模式的。这个们稍后会用到。

接下来我们使用模式对象的scanner()方法来完成分词操作。该方法会创建一个扫描对象，在给定的文中重复调用match()，一次匹配一个模式。下面这个交互式示例展示了扫描对象是如何工作的：

```
text = 'foo = 23 + 42 * 10'
import re
#tokens = [('NAME', 'foo'), ('EQ','='), ('NUM', '23'), ('PLUS','+'),
#          ('NUM', '42'), ('TIMES', '*'), ('NUM', 10')]
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]))
scanner=master_pat.scanner('foo = 42')
sm=scanner.match()
print(sm,sm.lastgroup,sm.group())

sm=scanner.match()
print(sm,sm.lastgroup,sm.group())
```

```
sm=scanner.match()
print(sm,sm.lastgroup,sm.group())

sm=scanner.match()
print(sm,sm.lastgroup,sm.group())

sm=scanner.match()
print(sm,sm.lastgroup,sm.group())
```

输出：

```
<re.Match object; span=(0, 3), match='foo'> NAME foo
<re.Match object; span=(3, 4), match=' '> WS
<re.Match object; span=(4, 5), match='='> EQ =
<re.Match object; span=(5, 6), match=' '> WS
<re.Match object; span=(6, 8), match='42'> NUM 42
```

要利用这项技术并将其转化为代码，我们可以做些清理工作然后轻松地将其包含在一个生成器函数中示例如下：

```
from collections import namedtuple
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]))

Token=namedtuple('Token',['type','value'])
def generate_tokens(pat,text):
    scanner=pat.scanner(text)
    for m in iter(scanner.match,None):
        yield Token(m.lastgroup,m.group())

for tok in generate_tokens(master_pat,'foo = 42'):
    print(tok)
```

输出：

```
Token(type='NAME', value='foo')
Token(type='WS', value=' ')
Token(type='EQ', value='=')
Token(type='WS', value=' ')
Token(type='NUM', value='42')
```

如果想以某种方式对标记流做过滤处理，要么定义更多的生成器函数，要么就用生成器表达式。例如下面的代码告诉我们如何过滤掉所有的空格标记。

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

```
    for tok in tokens:
        print(tok)
```

讨论

对于更加高级的文本解析，第一步往往是分词处理。要使用上面展示的扫描技术，有几个重要的细节要牢记于心。第一，对于每个可能出现在输入文本中的文本序列，都要确保有一个对应的正则表达式式可以将其识别出来。如果发现有任何不能匹配的文本，扫描过程就会停止。这就是为什么有必要在面的示例中指定空格标记（WS）。

这些标记在正则表达式（即re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))）中的顺序样也很重要。当进行匹配时，re模块会按照指定的顺序来对模式做匹配。因此，如果碰巧某个模式是一个较长模式的子串时，就必须确保较长的那个模式要先做匹配。示例如下：

```
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2个模式是错误的（注释掉的那一行），因为这样会把文本'<='匹配为LT（'<'）紧跟着EQ（'='）而没有匹配为单独的标记LE（'<='），这与我们的本意不符。

最后也最重要的是，对于有可能形成子串的模式要多加小心。例如，假设有如下两种模式：

```
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')
```

对于更加高级的分词处理，我们应该去看看像PyParsing或PLY这样的包。有关PLY的例子将在下一节讲解。