正则表达式

为什么要学正则表达式

实际上爬虫一共就四个主要步骤：

1. 明确目标 (要知道你准备在哪个范围或者网站去搜索)
2. 爬 (将所有的网站的内容全部爬下来)
3. 取 (去掉对我们没用处的数据)
4. 处理数据（按照我们想要的方式存储和使用）

我们在昨天的案例里实际上省略了第3步，也就是"取"的步骤。因为我们down下了的数据是全部的网页，这些数据很庞大并且很混乱，大部分的东西使我们不关心的，因此我们需要将之按我们的需要过滤和匹配出来。

那么对于文本的过滤或者规则的匹配，最强大的就是正则表达式，是Python爬虫世界里必不可少的神兵利器。

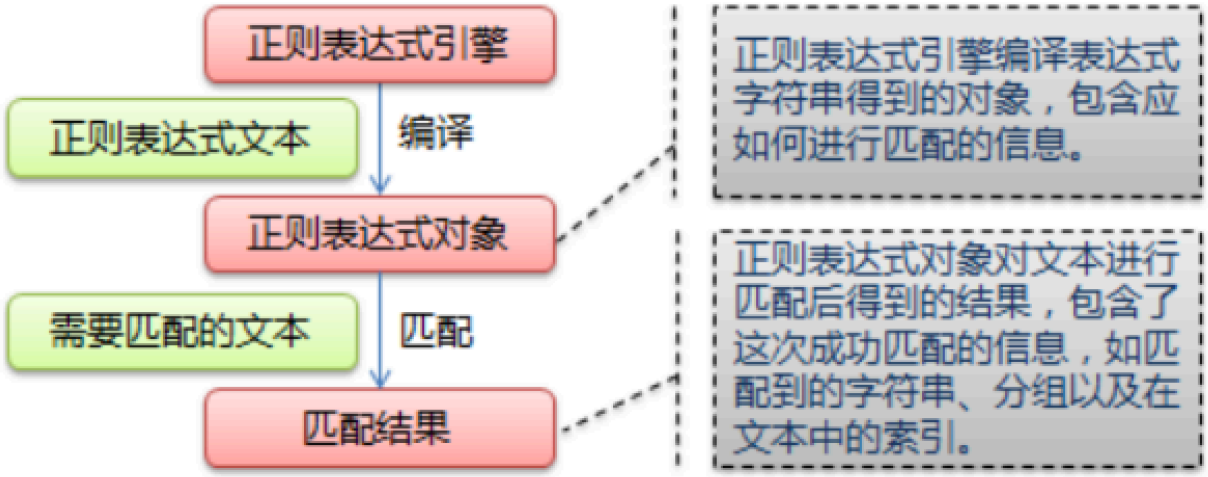
什么是正则表达式(Regular Expression)

> 正则表达式，又称规则表达式，通常被用来检索、替换那些符合某个模式(规则)的文本。
>
> 正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个"规则字符串"，这个"规则字符串"用来表达对字符串的一种过滤逻辑。

给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

> - 给定的字符串是否符合正则表达式的过滤逻辑（"匹配"）；

> - 通过正则表达式，从文本字符串中获取我们想要的特定部分（"过滤"）。



常用正则表达式规则

模式	描述
.	通配符，匹配任一字符(1. 任何字符都可以匹配; 2. 只匹配一个)
\d	匹配一个数字(digit)
\D	匹配一个非数字
\w	匹配一个单词(word) 外延: 数字 or 字母 or 下划线
\W	非\w
\s	匹配一个空白(space)字符。外延: \n, \t, ' '
\S	非\s
*	量词，0或者无穷多个 字符个数>= 0
+	量词，1或者无穷多个 字符个数>=1
?	非贪婪

完整全部规则:

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符：[^\s]	a\Sc	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c
数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	
边界匹配（不消耗待匹配字符串中的字符）			
^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc
逻辑、分组			
|	|代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果|没有被包括在()中，则它的范围是整个正则表达式。	abc|def	abc def
(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(abc){2} a(123|456)c	abcabc a456c
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc
\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5

(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5
特殊构造（不作为分组）			
(?:...)	(...)的不分组版本，用于使用'|'或后接数量词。	(?:abc){2}	abcabc
(?iLmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的内容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc2 abcabc

Python 的 re 模块

在 Python 中，我们可以使用内置的 re 模块来使用正则表达式。

有一点需要特别注意的是，正则表达式使用 对特殊字符进行转义，所以如果我们要使用原始字符串，只需加一个 r 前缀，示例：

```
In [1]: s = 'hello\tpython\nJunGe'

In [2]: print(s)
hello python
JunGe

In [3]: t = r'hello\tpython\nJunGe'

In [4]: print(t)
hello\tpython\nJunGe
```

re 模块的一般使用步骤如下：

1. 使用 `compile()` 函数将正则表达式的字符串形式编译为一个 `Pattern` 对象

2. 通过 `Pattern` 对象提供的一系列方法对文本进行匹配查找，获得匹配结果，一个 Match 对象。
3. 最后使用 `Match` 对象提供的属性和方法获得信息，根据需要进行其他的操作

compile 函数

compile 函数用于编译正则表达式，生成一个 Pattern 对象，它的一般使用形式如下：

```
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
```

在上面，我们已将一个正则表达式编译成 Pattern 对象，接下来，我们就可以利用 pattern 的一系列方法对文本进行匹配查找了。

Pattern 对象的一些常用方法主要有：

> - match 方法：从起始位置开始查找，一次匹配
> - search 方法：从任何位置开始查找，一次匹配
> - findall 方法：全部匹配，返回列表
> - finditer 方法：全部匹配，返回迭代器
> - split 方法：分割字符串，返回列表
> - sub 方法：替换

match 方法

match 方法用于查找字符串的头部（也可以指定起始位置），它是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果。它的一般使用形式如下：

```
match(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。因此，当你不指定 pos 和 endpos 时，match 方法默认匹配字符串的头部。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

```
>>> import re
>>> pattern = re.compile(r'\d+')  # 用于匹配至少一个数字

>>> m = pattern.match('one12twothree34four')  # 查找头部，没有匹配
>>> print(m)
None

>>> m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配
>>> print(m)
None

>>> m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配
>>> print(m)                                        # 返回一个 Match 对象
<_sre.SRE_Match object at 0x10a42aac0>

>>> m.group(0)   # 可省略 0
'12'
>>> m.start(0)   # 可省略 0
3
>>> m.end(0)     # 可省略 0
5
>>> m.span(0)    # 可省略 0
(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- `group([group1, …])`方法用于获得一个或多个分组匹配的字符串，当

要获得整个匹配的子串时，可直接使用 group() 或 group(0)；

- `start([group])`方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
- `end([group])`方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
- `span([group])`方法返回 (start(group), end(group))。

再看看一个例子：

```
>>> import re
>>> pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)  # re.I 表示忽略大小写
>>> m = pattern.match('Hello World Wide Web')

>>> print(m)     # 匹配成功，返回一个 Match 对象
<_sre.SRE_Match object at 0x10bea83e8>

>>> m.group(0)  # 返回匹配成功的整个子串
'Hello World'

>>> m.span(0)   # 返回匹配成功的整个子串的索引
(0, 11)

>>> m.group(1)  # 返回第一个分组匹配成功的子串
'Hello'

>>> m.span(1)   # 返回第一个分组匹配成功的子串的索引
(0, 5)

>>> m.group(2)  # 返回第二个分组匹配成功的子串
'World'

>>> m.span(2)   # 返回第二个分组匹配成功的子串
(6, 11)

>>> m.groups()  # 等价于 (m.group(1), m.group(2), ...)
('Hello', 'World')
```

```
28
29 >>> m.group(3)   # 不存在第三个分组
30 Traceback (most recent call last):
31   File "<stdin>", line 1, in <module>
32 IndexError: no such group
```

search 方法

search 方法用于查找字符串的任何位置，它也是一次匹配，只要找到了一个匹配的结果就返回，而不是查找所有匹配的结果，它的一般使用形式如下：

```
1 search(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

当匹配成功时，返回一个 Match 对象，如果没有匹配上，则返回 None。

让我们看看例子：

```
1  >>> import re
2  >>> pattern = re.compile('\d+')
3  >>> m = pattern.search('one12twothree34four')  # 这里如果使用
   match 方法则不匹配
4  >>> m
5  <_sre.SRE_Match object at 0x10cc03ac0>
6  >>> m.group()
7  '12'
8  >>> m = pattern.search('one12twothree34four', 10, 30)  # 指定
   字符串区间
9  >>> m
10 <_sre.SRE_Match object at 0x10cc03b28>
11 >>> m.group()
12 '34'
13 >>> m.span()
14 (13, 15)
```

再来看一个例子：

```
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'\d+')
# 使用 search() 查找匹配的子串，不存在匹配的子串时将返回 None
# 这里使用 match() 无法成功匹配
m = pattern.search('hello 123456 789')
if m:
    # 使用 Match 获得分组信息
    print 'matching string:',m.group()
    # 起始位置和结束位置
    print 'position:',m.span()
```

执行结果：

```
matching string: 123456
position: (6, 12)
```

findall 方法

上面的 match 和 search 方法都是一次匹配，只要找到了一个匹配的结果就返回。然而，在大多数时候，我们需要搜索整个字符串，获得所有匹配的结果。

findall 方法的使用形式如下：

```
findall(string[, pos[, endpos]])
```

其中，string 是待匹配的字符串，pos 和 endpos 是可选参数，指定字符串的起始和终点位置，默认值分别是 0 和 len (字符串长度)。

findall 以列表形式返回全部能匹配的子串，如果没有匹配，则返回一个空列表。

看看例子：

```python
import re
pattern = re.compile(r'\d+')   # 查找数字

result1 = pattern.findall('hello 123456 789')
result2 = pattern.findall('one1two2three3four4', 0, 10)

print(result1)
print(result2)
```

执行结果：

```
['123456', '789']
['1', '2']
```

再先看一个栗子：

```python
import re

#re模块提供一个方法叫compile模块，提供我们输入一个匹配的规则
#然后返回一个pattern实例，我们根据这个规则去匹配字符串
pattern = re.compile(r'\d+\.\d*')

#通过partten.findall()方法就能够全部匹配到我们得到的字符串
result = pattern.findall("123.141593, 'JunGe', 232312, 3.15")

#findall 以 列表形式 返回全部能匹配的子串给result
for item in result:
    print(item)
```

运行结果：

```
123.141593
3.15
```

finditer 方法

finditer 方法的行为跟 findall 的行为类似，也是搜索整个字符串，获得所有匹配的结果。但它返回一个顺序访问每一个匹配结果（Match 对象）的迭代器。

看看例子：

```
import re
pattern = re.compile(r'\d+')

result_iter1 = pattern.finditer('hello 123456 789')
result_iter2 = pattern.finditer('one1two2three3four4', 0,
10)

print(type(result_iter1))
print(type(result_iter2))

print('result1...')
for m1 in result_iter1:   # m1 是 Match 对象
    print('matching string: {}, position:
{}'.format(m1.group(), m1.span()))

print 'result2...'
for m2 in result_iter2:
    print('matching string: {}, position:
{}'.format(m2.group(), m2.span()))
```

执行结果：

```
<type 'callable-iterator'>
<type 'callable-iterator'>
result1...
matching string: 123456, position: (6, 12)
matching string: 789, position: (13, 16)
result2...
matching string: 1, position: (3, 4)
matching string: 2, position: (7, 8)
```

split 方法

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

```
split(string[, maxsplit])
```

其中，maxsplit 用于指定最大分割次数，不指定将全部分割。

看看例子：

```
import re
p = re.compile(r'[\s\,\;]+')
print(p.split('a,b;; c   d'))
```

执行结果：

```
['a', 'b', 'c', 'd']
```

sub 方法

sub 方法用于替换。它的使用形式如下：

```
sub(repl, string[, count])
```

其中，repl 可以是字符串也可以是一个函数：

- 如果 repl 是字符串，则会使用 repl 去替换字符串每一个匹配的子串，并返回替换后的字符串，另外，repl 还可以使用 id 的形式来引用分组，但不能使用编号 0；
- 如果 repl 是函数，这个方法应当只接受一个参数（Match 对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。
- count 用于指定最多替换次数，不指定时全部替换。

看看例子：

```python
import re
p = re.compile(r'(\w+) (\w+)') # \w = [A-Za-z0-9]
s = 'hello 123, hello 456'

print(p.sub(r'hello world', s))  # 使用 'hello world' 替换 'hello 123' 和 'hello 456'
print(p.sub(r'\2 \1', s))        # 引用分组

def func(m):
    return 'hi' + ' ' + m.group(2)

print(p.sub(func, s))
print(p.sub(func, s, 1))         # 最多替换一次
```

执行结果：

```
hello world, hello world
123 hello, 456 hello
hi 123, hi 456
hi 123, hello 456
```

匹配中文

在某些情况下，我们想匹配文本中的汉字，有一点需要注意的是，中文的 unicode 编码范围 主要在 [u4e00-u9fa5]，这里说主要是因为这个范围并不完整，比如没有包括全角（中文）标点，不过，在大部分情况下，应该是够用的。

假设现在想把字符串 title = u'你好，hello，世界' 中的中文提取出来，可以这么做：

```
import re

title = u'你好，hello，世界'
pattern = re.compile(ur'[\u4e00-\u9fa5]+')
result = pattern.findall(title)

print(result)
```

注意到，我们在正则表达式前面加上了两个前缀 ur，其中 r 表示使用原始字符串，u 表示是 unicode 字符串。

执行结果:

```
[u'\u4f60\u597d', u'\u4e16\u754c']
```

注意：贪婪模式与非贪婪模式

1. 贪婪模式：在整个表达式匹配成功的前提下，尽可能多的匹配 (*)；
2. 非贪婪模式：在整个表达式匹配成功的前提下，尽可能少的匹配 (?)；
3. **Python里数量词默认是贪婪的。**

示例一： 源字符串：abbbc

- 使用贪婪的数量词的正则表达式 ab* ， 匹配结果： abbb。

> * 决定了尽可能多匹配 b，所以a后面所有的 b 都出现了。

- 使用非贪婪的数量词的正则表达式ab*?，匹配结果： a。

> 即使前面有 *，但是 ? 决定了尽可能少匹配 b，所以没有 b。

示例二： 源字符串

`aa<div>test1</div>bb<div>test2</div>cc`

- 使用贪婪的数量词的正则表达式：`<div>.*</div>`
- 匹配结果：`<div>test1</div>bb<div>test2</div>`

> 这里采用的是贪婪模式。在匹配到第一个“`</div>`”时已经可以使整个表达式匹配成功，但是由于采用的是贪婪模式，所以仍然要向右尝试匹配，查看是否还有更长的可以成功匹配的子串。匹配到第二个“

”后，向右再没有可以成功匹配的子串，匹配结束，匹配结果为“`<div>test1</div>bb<div>test2</div>`”

- 使用非贪婪的数量词的正则表达式：`<div>.*?</div>`
- 匹配结果：`<div>test1</div>`

> 正则表达式二采用的是非贪婪模式，在匹配到第一个“”时使整个表达式匹配成功，由于采用的是非贪婪模式，所以结束匹配，不再向右尝试，匹配结果为“`<div>test1</div>`”。