# python的正则表达式 re

延伸阅读：python的 内建函数 和 subprocess。此文是本系列的第三篇文章了，和之前一样，内容出自官方文档，但是会有自己的理解，并非单纯的翻译。所以，如果我理解有误，欢迎指正，谢谢。

本模块提供了和 Perl里的正则表达式类似的功能，不关是正则表达式本身还是被搜索的字符串，都可以是 Unicode字符，这点不用担心， python会处理地和 Ascii字符一样漂亮。

正则表达式使用反斜杆（ \）来转义特殊字符，使其可以匹配字符本身，而不是指定其他特殊的含义。这可能会和 python字面意义上的字符串转义相冲突，这也许有些令人费解。比如，要匹配一个反斜杆本身，你也许要用 '\\\\'来做为正则表达式的字符串，因为正则表达式要是 \\，而字符串里，每个反斜杆都要写成 \\。

你也可以在字符串前加上 r这个前缀来避免部分疑惑，因为 r开头的 python字符串是 raw 字符串，所以里面的所有字符都不会被转义，比如 r'\n'这个字符串就是一个反斜杆加上一字母 n，而 '\n'我们知道这是个换行符。因此，上面的 '\\\\'你也可以写成 r'\\'；这样，应该就好理解很多了。可以看下面这段：

```
>>> import re
>>> s = '\x5c' #0x5c就是反斜杆
>>> print s
\
>>> re.match('\\\\', s) #这样可以匹配
<_sre.SRE_Match object at 0xb6949e20>
>>> re.match(r'\\', s) #这样也可以
<_sre.SRE_Match object at 0x80ce2c0>
>>> re.match('\\', s) #但是这样不行
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.6/re.py", line 137, in match
    return _compile(pattern, flags).match(string)
  File "/usr/lib/python2.6/re.py", line 245, in _compile
    raise error, v # invalid expression
sre_constants.error: bogus escape (end of line)
>>>
```

另外值得一提的是， re模块的方法，大多也就是 RegexObject对象的方法，两者的区别在于执行效率。这个在最后再展开吧。

### 正则表达式语法

正则表达式（RE）指定一个与之匹配的字符集合；本模块所提供的函数，将可以用来检查所给的字符串是否与指定的正则表达式匹配。

正则表达式可以被连接，从而形成新的正则表达式；例如 A和 B都是正则表达式，那么 AB也是正则表达式。一般地，如果字符串 *p*与 A匹配， *q*与 B匹配的话，那么字符串 *pq*也会与 AB匹配，但 A或者 B里含有边界限定条件或者命名组操作的情况除外。也就是说，复杂的正则表达式可以用简单的连接而成。

正则表达式可以包含特殊字符和普通字符，大部分字符比如 'A'， 'a'和 '0'都是普通字符，如果做为正则表达式，它们将匹配它们本身。由于正则表达式可以连接，所以连接多个普通字符而成的正则表达式 last也将匹配 'last'。（后面将用不带引号的表示正则表达式，带引号的表示字符串）

下面就来介绍正则表达式的特殊字符：

'.'

点号，在普通模式，它匹配除换行符外的任意一个字符；如果指定了 **DOTALL**标记，匹配包括换行符以内的任意一个字符。

'^'

尖尖号，匹配一个字符串的开始，在 **MULTILINE**模式下，也将匹配任意一个新行的开始。

'$'

美元符号，匹配一个字符串的结尾或者字符串最后面的换行符，在 **MULTILINE**模式下，也匹配任意一行的行尾。也就是说，普通模式下， foo $去搜索 'foo1\nfoo2\n只会找到 ’ foo2'；但是在 **MULTILINE**模式，还能找到 ‘ foo1'；而且就用一个 $去搜索 'foo\n'的话，会找到两个空的匹配：一个是最后的换行符，一个是字符串的结尾，演示：

```
>>> re.findall('(foo$)', 'foo1\nfoo2\n')
['foo2']
>>> re.findall('(foo$)', 'foo1\nfoo2\n', re.MULTILINE)
['foo1', 'foo2']
>>> re.findall('($)', 'foo\n')
['', '']
```

'*'

星号，指定将前面的 RE重复 0次或者任意多次，而且总是试图尽量多次地匹配。

'+'

加号，指定将前面的 RE重复 1次或者任意多次，而且总是试图尽量多次地匹配。

'?'

问号，指定将前面的 RE重复 0次或者 1次，如果有的话，也尽量匹配 1次。

*?, +?, ??

从前面的描述可以看到 '*'， '+'和 '?'都是*贪婪的*，但这也许并不是我们说要的，所以，可以在后面加个问号，将策略改为*非贪婪*，只匹配尽量少的 RE。示例，体会两者的区别：

```
>>> re.findall('<(.*)>', '<h1>title</h1>')
['h1>title</h1']
>>> re.findall('<(.*?)>', '<h1>title</h1>')
['h1', '/h1']
```

{m}

m 是一个数字，指定将前面的 RE重复 m 次。

{m,n}

m 和 n都是数字，指定将前面的 RE重复 m 到 n次，例如 a{3,5}匹配 3到 5个连续的 a。注意，如果省略 m，将匹配 0到 n个前面的 RE；如果省略 n，将匹配 n到无穷多个前面的 RE；当然中间的逗号是不能省略的，不然就变成前面那种形式了。

{m,n}?

前面说的 {m,n}，也是贪婪的，a{3,5}如果有 5个以上连续 a的话，会匹配 5个，这个也可以通过加问号改变。a{3,5}?如果可能的话，将只匹配 3个 a。

'\'

反斜杆，转义 '*'， '?'等特殊字符，或者指定一个特殊序列（下面会详述）

由于之前所述的原因，强烈建议用 raw字符串来表述正则。

[]

方括号，用于指定一个字符的集合。可以单独列出字符，也可以用 '-'连接起止字符以表示一个范围。特殊字符在中括号里将失效，比如 [akm$]就表示字符 'a'， 'k'， 'm'，或 '$'，在这里 $也变身为普通字符了。 [a-z]匹配任意一个小写字母， [a-zA-Z0-9]匹配任意一个字母或数字。如果你要匹配 ']'或 '-'本身，你需要加反斜杆转义，或者是将其置于中括号的最前面，比如 []]可以匹配 ']'

你还可以对一个字符集合*取反*，以匹配任意不在这个字符集合里的字符，取反操作用一个 '^'放在集合的最前面表示，放在其他地方的 '^'将不会起特殊作用。例如 [^5]将匹配任意不是 '5'的字符；[^^]将匹配任意不是 '^'的字符。

注意：在中括号里，+、*、(、)这类字符将会失去特殊含义，仅作为普通字符。反向引用也不能在中括号内使用。

'|'

管道符号，A和B是任意的RE，那么A|B就是匹配A或者B的一个新的RE。任意个数的RE都可以像这样用管道符号间隔连接起来。这种形式可以被用于**组**中（后面将详述）。对于目标字符串，被'|'分割的RE将自左至右一一被测试，一旦有一个测试成功，后面的将不再被测试，即使后面的RE可能可以匹配更长的串，换句话说，'|'操作符是非贪婪的。要匹配字面意义上的'|'，可以用反斜杆转义：\|，或是包含在反括号内：[|]。

(...)

匹配圆括号里的RE匹配的内容，并指定**组**的开始和结束位置。组里面的内容可以被提取，也可以采用\number这样的特殊序列，被用于后续的匹配。要匹配字面意义上的'('和')'，可以用反斜杆转义：\(、\)，或是包含在反括号内：[(]、[)]。

(?...)

这是一个表达式的扩展符号。'?'后的第一个字母决定了整个表达式的语法和含义，除了(?P...)以外，表达式不会产生一个新的组。下面介绍几个目前已被支持的扩展：

(?iLmsux)

'i'、'L'、'm'、's'、'u'、'x'里的一个或多个字母。表达式不匹配任何字符，但是指定相应的标志：**re.I**（忽略大小写）、**re.L**（依赖locale）、**re.M**（多行模式）、**re.S**（匹配所有字符）、**re.U**（依赖Unicode）、**re.X**（详细模式）。关于各个模式的区别，下面会有专门的一节来介绍的。使用这个语法可以代替在re.compile()的时候或者调用的时候指定*flag*参数。

例如，上面举过的例子，可以改写成这样（和指定了reMULTILINE是一样的效果）：

```
>>> re.findall('(?m)(foo.$)', 'foo1\nfoo2\n')
['foo1', 'foo2']
```

另外，还要注意(?x)标志如果有的话，要放在最前面。

(?:...)

匹配内部的RE所匹配的内容，但是不建立**组**。

(?P<name>...)

和普通的圆括号类似，但是子串匹配到的内容将可以用命名的*name*参数来提取。组的*name*必须是有效的python标识符，而且在本表达式内不重名。命名了的组和普通组一样，也用数字来提取，也就是说名字只是个额外的属性。

演示一下：

```
>>> m = re.match('(?P<var>[a-zA-Z_]\w*)', 'abc=123')
```

```
>>> m.group('var')
'abc'
>>> m.group(1)
'abc'
```

(?P=name)

匹配之前以 *name*命名的组里的内容。

演示一下：

```
>>> re.match('<(?P<tagname>\w*)>.*</(?P=tagname)>', '<h1>xxx</h2>') # 这个不匹配
>>> re.match('<(?P<tagname>\w*)>.*</(?P=tagname)>', '<h1>xxx</h1>') # 这个匹配
<_sre.SRE_Match object at 0xb69588e0>
```

(?#...)

注释，圆括号里的内容会被忽略。

(?=...)

如果 ...匹配接下来的字符，才算匹配，但是并不会消耗任何被匹配的字符。例如 Isaac (?=Asimov)只会匹配后面跟着 'Asimov'的 'Isaac '；这个叫做“前瞻断言”。

(?!...)

和上面的相反，只匹配接下来的字符串**不**匹配 ...的串，这叫做“反前瞻断言”。

(?<=...)

只有当当前位置之前的字符串匹配 ...，整个匹配才有效，这叫“后顾断言”。(?<=abc)def会找到 'abcdef'，因为会后向查找 3个字符，看是否为 abc。所以内置的子 RE，需要是固定长度的，比如可以是 abc、a|b，但不能是 a*、a{3,4}。注意这种 RE永远不会匹配到字符串的开头。举个例子，找到连字符（'-'）后的单词：

```
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

(?<!...)

同理，这个叫做“反后顾断言”，子 RE需要固定长度的，含义是前面的字符串不匹配 ...整个才算匹配。

(?(id/name)yes-pattern|no-pattern)

如有由 *id*或者 *name*指定的组存在的话，将会匹配 yes-pattern，否则将会匹配 no-pattern，通常情况下 no-pattern也可以省略。例如：(可以匹配 '<user@ host.com>'和 'user@ host.com'；但是不会匹配 '<user@ host.com'。

下面列出以 '\'开头的特殊序列。如果某个字符没有在下面列出，那么 RE的结果会只匹配那个字母本身，比如， \$只匹配字面意义上的 '$'。

\number

匹配 number所指的组相同的字符串。组的序号从 1开始。例如：(.+) \1可以匹配 'the the'和 '55 55'，但不匹配 'the end'。这种序列在一个正则表达式里最多可以有 99个，如果 *number*以 0开头，或是有 3位以上的数字，就会被当做八进制表示的字符了。同时，这个也不能用于方括号内。

\A

只匹配字符串的开始。

\b

匹配单词边界（包括开始和结束），这里的“ 单词 ”，是指连续的字母、数字和下划线组成的字符串。注意， \b的定义是 \w和 \W 的交界，所以精确的定义有赖于 UNICODE和 LOCALE这两个标志位。

\B

和 \b相反， \B匹配非单词边界。也依赖于 UNICODE和 LOCALE这两个标志位。

\d

未指定 UNICODE标志时，匹配数字，等效于：[0-9]。指定了 UNICODE标志时，还会匹配其他 Unicode库里描述为字符串的符号。便于理解，举个例子（好不容易找的例子啊，呵呵）：

```
# \u2076和 u2084分别是上标的 6和下标的 4，属于 unicode的 DIGIT
>>> unistr = u'\u2076\u2084abc'
>>> print unistr
⁶₄abc
>>> print re.findall('\d+',unistr,re.U)[0]
⁶₄
```

\D

和 \d相反，不多说了。

\s

当未指定 UNICODE和 LOCALE这两个标志位时，匹配任何空白字符，等效于 [ \t\n\r\f\v]。如果指定了 LOCALE，则还要加 LOCALE相关的空白字符；如果指定了 UNICODE，还要加上 UNICODE空白字符，如较常见的空宽度连接空格（\uFEFF）、零宽度非连接空格 (\u200B)等。

\S

和 \s相反，也不多说。

\w

当未指定 UNICODE和 LOCALE这两个标志位时，等效于 [a-zA-Z0-9_]。当指定了 LOCALE时，为 [0-9_]加上当前 LOCAL指定的字母。当指定了 UNICODE时，为 [0-9_]加上 UNICODE库里的所有字母。

\W

和 \w相反，不多说。

\Z

只匹配字符串的结尾。

**匹配之于搜索**

python提供了两种基于正则表达式的操作：匹配（match）从字符串的开始检查字符串是否个正则匹配。而搜索（search）检查字符串任意位置是否有匹配的子串（perl默认就是如此）。

注意，即使 search的正则以 '^'开头，match和 search也还是有许多不同的。

```
>>> re.match("c", "abcdef") #不匹配
>>> re.search("c", "abcdef") #匹配
<_sre.SRE_Match object at ...>
```

**模块的属性和方法**

re.**compile**(*pattern*[, *flags*])

把一个正则表达式 *pattern*编译成正则对象，以便可以用正则对象的 **match**和 **search**方法。

得到的正则对象的行为（也就是模式）可以用 *flags*来指定，值可以由几个下面的值 OR得到。

以下两段内容在语法上是等效的：

```
prog = re.compile(pattern)
result = prog.match(string)
```

```
result = re.match(pattern, string)
```

区别是，用了 **re.compile**以后，正则对象会得到保留，这样在需要多次运用这个正则对象的时候，效率会有较大的提升。再用上面用过的例子来演示一下，用相同的正则匹配相同的字符串，执行 100万次，就体现出 compile的效率了（数据来自我那 1.86G CPU的神舟本本）：

```
>>> timeit.timeit(
...     setup='''import re; reg =
re.compile('<(?P<tagname>\w*)>.*</(?P=tagname)>')''',
...     stmt='''reg.match('<h1>xxx</h1>')''',
...     number=1000000)
1.2062149047851562
>>> timeit.timeit(
...     setup='''import re''',
...     stmt='''re.match('<(?P<tagname>\w*)>.*</(?P=tagname)>',
'<h1>xxx</h1>')''',
...     number=1000000)
4.4380838871002197
```

re.**I**

re.**IGNORECASE**

让正则表达式忽略大小写，这样一来，[A-Z]也可以匹配小写字母了。此特性和 locale无关。

re.**L**

re.**LOCALE**

让 \w、\W、\b、\B、\s和 \S依赖当前的 locale。

re.**M**

re.**MULTILINE**

影响 '^和 '$的行为，指定了以后，'^会增加匹配每行的开始（也就是换行符后的位置）；'$会增加匹配每行的结束（也就是换行符前的位置）。

re.**S**

re.**DOTALL**

影响 '.的行为，平时 '.匹配除换行符以外的所有字符，指定了本标志以后，也可以匹配换行符。

re.U

re.UNICODE

让 \w、 \W、 \b、 \B、 \d、 \D、 \s和 \S依赖 Unicode库。

re.X

re.VERBOSE

运用这个标志，你可以写出可读性更好的正则表达式：除了在方括号内的和被反斜杠转义的以外的所有空白字符，都将被忽略，而且每行中，一个正常的井号后的所有字符也被忽略，这样就可以方便地在正则表达式内部写注释了。也就是说，下面两个正则表达式是等效的：

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.**search**(*pattern*, *string*[, *flags*])

扫描 *string*，看是否有个位置可以匹配正则表达式 *pattern*，如果找到了，就返回一个**MatchObject**的实例，否则返回 **None**，注意这和找到长度为 0的子串含义是不一样的。搜索过程受 *flags*的影响。

re.**match**(*pattern*, *string*[, *flags*])

如果字符串 *string*的开头和正则表达式 *pattern*匹配的话，返回一个相应的**MatchObject**的实例，否则返回 **None**

注意：要在字符串的任意位置搜索的话，需要使用上面的 **search()**。

re.**split**(*pattern*, *string*[,*maxsplit=0*])

用匹配 *pattern*的子串来分割 *string*，如果 *pattern*里使用了圆括号，那么被 *pattern*匹配到的串也将作为返回值列表的一部分。如果 *maxsplit*不为 0，则最多被分割为 *maxsplit*个子串，剩余部分将整个地被返回。

```
>>> re.split('\W +', 'Words,words,words.')
['Words', 'words', 'words', '']
>>> re.split('(\W +)', 'Words,words,words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W +', 'Words,words,words.', 1)
['Words', 'words,words.']
```

如果正则有圆括号，并且可以匹配到字符串的开始位置的时候，返回值的第一项，会多出一个空字符串。匹配到字符结尾也是同样的道理：

```
>>> re.split('(\W+)', '...words,words...')
['', '...', 'words', ', ', 'words', '...', '']
```

注意， *split*不会被零长度的正则所分割，例如：

```
>>> re.split('x*', 'foo')
['foo']
>>> re.split("(?m)^$", "foo\n\nbar\n")
['foo\n\nbar\n']
```

re.**findall**(*pattern*, *string*[, *flags*])

以列表的形式返回 *string*里匹配 *pattern*的不重叠的子串。 *string*会被从左到右依次扫描，返回的列表也是从左到右一次匹配到的。如果 *pattern*里含有**组**的话，那么会返回匹配到的组的列表；如果 *pattern*里有多个组，那么各组会先组成一个元组，然后返回值将是一个元组的列表。

由于这个函数不会涉及到 **MatchObject**之类的概念，所以，对新手来说，应该是最好理解也最容易使用的一个函数了。下面就此来举几个简单的例子：

```
#简单的 findall
>>> re.findall('\w+', 'hello,world!')
['hello', 'world']
#这个返回的就是元组的列表
>>> re.findall('(\d+)\.(\d+)\.(\d+)\.(\d+)', My IP is 192.168.0.2,and your is 192.168.0.3.')
[('192', '168', '0', '2'), ('192', '168', '0', '3')]
```

re.**finditer**(*pattern*, *string*[, *flags*])

和上面的 **findall()**类似，但返回的是 **MatchObject**的实例的迭代器。

还是例子说明问题：

```
>>> for m in re.finditer('\w+', 'hello,world!'):
...print m.group()
...
hello
world
```

re.**sub**(*pattern*, *repl*, *string*[, *count*])

替换，将 *string*里，匹配 *pattern*的部分，用 *repl*替换掉，最多替换 *count*次（剩余的匹配将不做处理），然后返回替换后的字符串。如果 *string*里没有可以匹配 *pattern*的串，将被原封不动地返回。 *repl*可以是一个字符串，也可以是一个函数（也可以参考我以前的例子）。如果 *repl*是个字符串，则其中的反斜杆会被处理过，比如 \n会被转成换行符，反斜杆加数字会被替换成相应的组，比如 \6表示 *pattern*匹配到的第 6个组的内容。

例子：

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl*是个函数，每次 *pattern*被匹配到的时候，都会被调用一次，传入一个匹配到的 **MatchObject**对象，需要返回一个字符串，在匹配到的位置，就填入返回的字符串。

例子：

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
```

零长度的匹配也会被替换，比如：

```
>>> re.sub('x*', '-', 'abcxxd')
'-a-b-c-d-'
```

特殊地，在替换字符串里，如果有 \g这样的写法，将匹配正则的命名组（前面介绍过的，(?P...)这样定义出来的东西）。 \g这样的写法，也是数字的组，也就是说， \g一般和 \2是等效的，但是万一你要在 \2后面紧接着写上字面意义的 0，你就不能写成 \20了（因为这代表第 20个组），这时候必须写成 \g0，另外， \g代表匹配到的整个子串。

例子：

```
>>> re.sub('-(\d+)-', '-\g<1>0\g<0>', 'a-11-b-22-c')
'a-110-11-b-220-22-c'
```

re.**subn**(*pattern*, *repl*, *string*[, *count*])

跟上面的 sub()函数一样，只是它返回的是一个元组（新字符串，匹配到的次数），还是用例子说话：

```
>>> re.subn('-(\d+)-', '-\g<1>0\g<0>', 'a-11-b-22-c')
('a-110-11-b-220-22-c', 2)
```

re.**escape**(*string*)

把 *string*中，除了字母和数字以外的字符，都加上反斜杆。

```
>>> print re.escape('abc123_@ #$')
abc123\_\@ \#\$
```

*exception* re.**error**

如果字符串不能被成功编译成正则表达式或者正则表达式在匹配过程中出错了，都会抛出此异常。但是如果正则表达式没有匹配到任何文本，是不会抛出这个异常的。

**正则对象**

正则对象由 re.**compile()**返回。它有如下的属性和方法。

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

作用和模块的 **match()**函数类似，区别就是后面两个参数。

*pos*是开始搜索的位置，默认为 0。 *endpos*是搜索的结束位置，如果 *endpos*比 *pos* 还小的话，结果肯定是空的。也就是说只有 *pos*到 *endpos*-1位置的字符串将会被搜索。

例子：

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")    #开始位置不是 o，所以不匹配
>>> pattern.match("dog",1)  #第二个字符是 o，所以匹配
<_sre.SRE_Match object at ...>
```

**search**(*string*[,*pos*[,*endpos*]])

作用和模块的 **search()**函数类似， *pos*和 *endpos*参数和上面的 **match()**函数类似。

**split**(*string*[,*maxsplit*=0])

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

**finditer**(*string*[,*pos*[,*endpos*]])

**sub**(*repl*, *string*[, *count*=*0*])

**subn**(*repl*, *string*[, *count*=*0*])

这几个函数，都和模块的相应函数一致。

**flags**

编译本 RE时，指定的标志位，如果未指定任何标志位，则为 0。

```
>>> pattern = re.compile("o", re.S|re.U)
>>> pattern.flags
```

**groups**

RE所含有的组的个数。

**groupindex**

一个字典，定义了命名组的名字和序号之间的关系。

例子：

这个正则有 3个组，如果匹配到，第一个叫区号，最后一个叫分机号，中间的那个未命名

```
>>> pattern = re.compile("(?P<quhao>\d+)-(\d+)-(?P<fenjihao>\d+)")
>>> pattern.groups
3
>>> pattern.groupindex
{'fenjihao':3, 'quhao':1}
```

**pattern**

建立本 RE的原始字符串，相当于源代码了，呵呵。

还是上面这个正则，可以看到，会原样返回：

```
>>> print pattern.pattern
(?P<quhao>\d+)-(\d+)-(?P<fenjihao>\d+)
```

**Match对象**

re**MatchObject**被用于布尔判断的时候，始终返回 True，所以你用 if语句来判断某个 **match()**是否成功是安全的。

它有以下方法和属性：

**expand**(*template*)

用 *template*做为模板，将 **MatchObject**展开，就像 **sub()**里的行为一样，看例子：

```
>>> m = re.match('a=(\d+)', 'a=100')
>>> m.expand('above a is \g<1>')
'above a is 100'
>>> m.expand(r'above a is \1')
'above a is 100'
```

**group**([*group1*, ...])

返回一个或多个子组。如果参数为一个，就返回一个子串；如果参数有多个，就返回多个子串注册的元组。如果不传任何参数，效果和传入一个 0一样，将返回整个匹配。如果某

个 *groupN*未匹配到，相应位置会返回 **None**。如果某个 *groupN*是负数或者大于 group 的总数，则会抛出 **IndexError**异常。

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       #整个匹配
'Isaac Newton'
>>> m.group(1)       #第一个子串
'Isaac'
>>> m.group(2)       #第二个子串
'Newton'
>>> m.group(1,2)     #多个子串组成的元组
('Isaac', 'Newton')
```

如果有其中有用 (?P...)这种语法命名过的子串的话，相应的 *groupN*也可以是名字字符串。例如：

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", 'Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

如果某个组被匹配到多次，那么只有最后一次的数据，可以被提取到：

```
>>> m = re.match(r"(..)+", "a1b2c3")  #匹配到 3次
>>> m.group(1)                        #返回的是最后一次
'c3'
```

**groups**([*default*])

返回一个由所有匹配到的子串组成的元组。 *default*参数，用于给那些没有匹配到的组做默认值，它的默认值是 **None**

例如：

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

*default*的作用：

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      #第二个默认是 None
```

```
('24',None)
>>> m.groups('0')  #现在默认是 0了
('24', '0')
```

**groupdict**([*default*])

返回一个包含所有命名组的名字和子串的字典， *default*参数，用于给那些没有匹配到的组做默认值，它的默认值是 **None**，例如：

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

**start**([**group**])

**end**([*group*])

返回的是：被组 *group*匹配到的子串在原字符串中的位置。如果不指定 *group*或 *group*指定为 0，则代表整个匹配。如果 *group*未匹配到，则返回 -1。

对于指定的 m 和 g，m.group(g)和 m.string[m.start(g):m.end(g)]等效。

注意：如果 *group*匹配到空字符串，*m.start(group)*和 *m.end(group)*将相等。

例如：

```
>>> m = re.search('b(c?)', 'cba')
>>> m.start(0)
1
>>> m.end(0)
2
>>> m.start(1)
2
>>> m.end(1)
2
```

下面是一个把 email地址里的“ remove_this”去掉的例子：

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

**span**([*group*])

返回一个元组： (m.start(group),m.end(group))

**pos**

就是传给 RE对象的 **search()**或 **match()**方法的参数 *pos*，代表 RE开始搜索字符串的位置。

**endpos**

就是传给 RE对象的 **search()**或 **match()**方法的参数 *endpos*，代表 RE搜索字符串的结束位置。

**lastindex**

最后一次匹配到的组的数字序号，如果没有匹配到，将得到 **None**。

例如：(a)b、((a)(b))和 ((ab))正则去匹配 'ab的话，得到的 **lastindex**为 1。而用 (a)(b)去匹配 'ab的话，得到的 **lastindex**为 2。

**lastgroup**

最后一次匹配到的组的名字，如果没有匹配到或者最后的组没有名字，将得到 **None**。

**re**

得到本 Match对象的正则表达式对象，也就是执行 **search()**或 **match()**的对象。

**string**

传给 **search()**或 **match()**的字符串。