# [python中的正则表达式](https://www.cnblogs.com/panwenbin-logs/p/5584864.html)

**引言：**正则表达式为高级的文本模式匹配、抽取、与/或文本形式和替换功能提供基础。在python中，通过标准库中的re模块来支持正则表达式



**'.'**  
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re.MULTILINE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。

**(?#...)**  
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”。字符串'abcdef'可以匹配正则(?<=abc)def，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：(<)?(\w+@\w+(?:\.\w+)+)(?(1)>)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

>>> unistr = u'\u2076\u2084abc'

>>> print unistr

abc

**\D**  
和\d相反，不多说了。

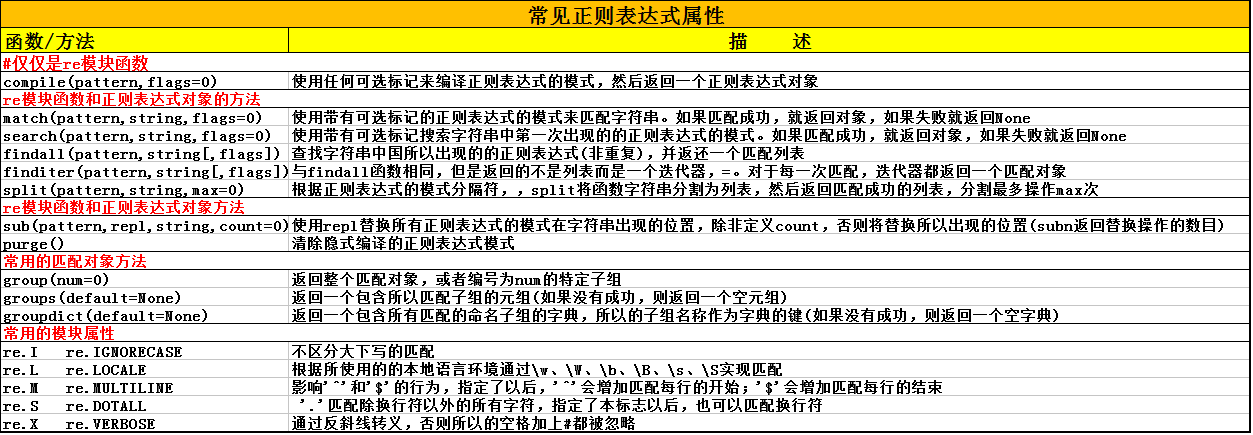
**\s**  
当未指定UNICODE和LOCALE这两个标志位时，匹配任何空白字符，等效于[ \t\n\r\f\v]。如果指定了LOCALE，则还要加LOCALE相关的空白字符；如果指定了UNICODE，还要加上UNICODE空白字符，如较常见的空宽度连接空格（\uFEFF）、零宽度非连接空格(\u200B)等。

**\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。



**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

[复制代码](javascript:void(0);)

>>> 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

[复制代码](javascript:void(0);)

**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：  
代码如下:

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。  
代码如下:

>>> 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<2>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\g<2>0，另外，\g<0>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。  
代码如下:

>>> 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。  
代码如下:

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

>>> pattern.flags

48

**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  
代码如下:

>>> 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：  
代码如下:

>>> 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：  
代码如下:

>>> 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。  
代码如下

[复制代码](javascript:void(0);)

>>> 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')

[复制代码](javascript:void(0);)

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：  
代码如下:

>>> 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：  
代码如下:

>>> 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：**  
代码如下:

>>> 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：  
代码如下:

>>> 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：

[复制代码](javascript:void(0);)

>>> m = re.search('b(c?)', 'cba')

>>> m.start(0)

1

>>> m.end(0)

2

>>> m.start(1)

2

>>> m.end(1)

2

[复制代码](javascript:void(0);)

下面是一个把email地址里的“remove\_this”去掉的例子：  
代码如下:

>>> 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。

=================================例子===========================================

import re

# ret = re.findall(r'\d', r'tom jerry spike') # 返回所有满足匹配条件的结果,放在列表里

# print(ret) #结果 : ['a', 'a']

# ret = re.search('a', 'tom jerry spike')

# if ret:

# print(ret.group())

#search从左到右依次找，找到一个就回来，需要使用group()获取返回值

#如果re.search找不到，就返回None。使用group会报错

# ret = re.match('a', 'om jerry spike')

# print(ret.group())

#match从头开始匹配，匹配上了需要使用group来获取返回值

#匹配不上返回None，使用group会报错

# 'a,b'.split(',') #[a,b]

# ret = re.split('[ac]', 'abcd') # 先按'a'分割得到''和'bcd',在对''和'bcd'分别按'b'分割

# print(ret) # ['', '', 'cd']

# ret = re.sub('\d', 'H', 'tom3jerry4spike4',1) #replace(old,new,count)

# #sub(re,new,str,count)

# print(ret)

# ret = re.subn('\d', 'H', 'tom3jerry4spike4')#将数字替换成'H'，返回元组(替换的结果,替换了多少次)

# print(ret)

# obj = re.compile('\d{3}') #将正则表达式编译成为一个 正则表达式对象，规则要匹配的是3个数字

# ret = obj.search('abc123eeee') #正则表达式对象调用search，参数为待匹配的字符串

# print(ret.group()) #结果 ： 123

#

# re.search('\d{3}','abc123eeee').group()

# re.search('\d{3}','bcd123eeee')

# re.search('\d{3}','efg123eeee')

# re.search('\d{3}','xyz123eeee')

# re.findall() #[]

ret = re.finditer('\d', 'ds3sy4784a') #finditer返回一个存放匹配结果的迭代器

print(ret) # <callable\_iterator object at 0x10195f940>

for i in ret:

print(i.group())

# print(next(ret).group()) #查看第一个结果

# print(next(ret).group()) #查看第二个结果

# print([i.group() for i in ret]) #查看剩余的左右结果

# findall

# search

# match

# split

# sub/subn

# compile

# finditer

express = '1 - 2 \* ( (60-30 +(-40/5) \* (9-3.33 + 198/4\*2998 +10 \* 568/14 )) - (-4\*3)/ (16-3\*2) )'

#正则表达式

#0. 去掉表达式中的所有空格

#1. 从表达式中匹配出所有的（）里面不再有小括号的表达式

#2. 从表达式9-2\*5/3 + 7 /3\*99/4\*2998 +10 \* 568/14中匹配出第一个乘法或者除法

#3. 计算简单的两个数之间的+-\*/

#4. 递归——不用递归更简单 ——> 循环

#5. 博客上的数字匹配