1．  Python正则式的基本用法

1.1基本规则

1.2重复

1.2.1最小匹配与精确匹配

1.3前向界定与后向界定

1.4组的基本知识

2．  re模块的基本函数

2.1使用compile加速

2.2 match和search

2.3 finditer

2.4 字符串的修改与替换

3．  更深入的了解re的组与对象

3.1编译后的Pattern对象

3.2 组与Match对象

3.2.1组的名字与序号

3.2.2 Match对象的方法

4．  更多的资料

初学Python，对Python的文字处理能力有很深的印象，除了str对象自带的一些方法外，就是正则表达式这个强大的模块了。但是对于初学者来说，要用好这个功能还是有点难度，我花了好长时间才摸出了点门道。由于我记性不好，很容易就忘事，所以还是写下来比较好一些，同时也可以加深印象，整理思路。

由于我是初学，所以肯定会有些错误，还望高手不吝赐教，指出我的错误。

**1 Python正则式的基本用法**

Python的正则表达式的模块是 ‘re’,它的基本语法规则就是指定一个字符序列，比如你要在一个字符串s=’123abc456’ 中查找字符串 ’abc’,只要这样写：

>>> import re

>>> s='123abc456eabc789'

>>> re.findall(r’abc’,s)

结果就是：

['abc', 'abc']

这里用到的函数 ”findall(rule , target [,flag] )” 是个比较直观的函数，就是在目标字符串中查找符合规则的字符串。第一个参数是规则，第二个参数是目标字符串，后面还可以跟一个规则选项（选项功能将在compile函数的说明中详细说明）。返回结果结果是一个**列表，**中间存放的是符合规则的字符串。如果没有符合规则的字符串被找到，就返回一个**空**列表。

**为什么要用r’ ..‘字符串（raw字符串）？ 由于正则式的规则也是由一个字符串定义的，而在正则式中大量使用转义字符’\’，如果不用raw字符串，则在需要写一个’\’的地方，你必须得写成’\\’,那么在要从目标字符串中匹配一个’\’的时候，你就得写上4个’\’成为’\\\\’！这当然很麻烦，也不直观，所以一般都使用r’’来定义规则字符串。当然，某些情况下，可能不用raw字符串比较好。**

以上是个最简单的例子。当然实际中这么简单的用法几乎没有意义。为了实现复杂的规则查找，re规定了若干语法规则。它们分为这么几类：

功能字符 ：    ‘.’ ‘\*’ ‘+’ ‘|’ ‘?’ ‘^’ ‘$’ ‘\’ 等，它们有特殊的功能含义。特别是’/’字符，它是转义引导符号，跟在它后面的字符一般有特殊的含义。

规则分界符： ‘[‘ ‘]’ ‘（’ ‘）’ ‘{‘ ‘}’ 等，也就是几种括号了。

预定义转义字符集： “\d”  “\w” “\s” 等等，它们是以字符’\’开头，后面接一个特定字符的形式，用来指示一个预定义好的含义。

其它特殊功能字符： ’#’ ‘!’ ‘:’ ‘-‘等，它们只在特定的情况下表示特殊的含义，比如(?# …)就表示一个注释，里面的内容会被忽略。

下面来一个一个的说明这些规则的含义，不过说明的顺序并不是按照上面的顺序来的，而是我认为由浅入深，由基本到复杂的顺序来编排的。同时为了直观，在说明的过程中尽量多举些例子以方便理解。

**1.1 基本规则**

**‘[‘  ‘]’ 字符集合设定符**

首先说明一下字符集合设定的方法。由一对方括号括起来的字符，表明一个字符集合，能够匹配包含在其中的任意一个字符。比如 [abc123]，表明字符’a’ ‘b’ ‘c’ ‘1’ ‘2’ ‘3’都符合它的要求。可以被匹配。

在’[‘ ‘]’中还可以通过 ’-‘ 减号来指定一个字符集合的范围，比如可以用[a-zA-Z]来指定所以英文字母的大小写，因为英文字母是按照从小到大的顺序来排的。你不可以把大小的顺序颠倒了，比如写成[z-a]就不对了。

如果在’[‘ ‘]’里面的开头写一个 ‘^’ 号，则表示取非，即在括号里的字符都不匹配。如[^a-zA-Z]表明不匹配所有英文字母。但是如果 ‘^’不在开头，则它就不再是表示取非，而表示其本身，如[a-z^A-Z]表明匹配所有的英文字母和字符’^’。

**‘|’    或规则**

将两个规则并列起来，以‘|’连接，表示只要满足其中之一就可以匹配。比如

[a-zA-Z]|[0-9] 表示满足数字或字母就可以匹配，这个规则等价于 [a-zA-Z0-9]

**注意**：关于’|’要注意两点：

第一，           它在’[‘ ‘]’之中不再表示或，而表示他本身的字符。如果要在’[‘ ‘]’外面表示一个’|’字符，必须用反斜杠引导，即 ’/|’ ;

第二，           它的有效范围是它两边的整条规则，比如‘dog|cat’匹配的是‘dog’和’cat’，而不是’g’和’c’。如果想限定它的有效范围，必需使用一个无捕获组 ‘(?: )’包起来。比如要匹配 ‘I have a dog’或’I have a cat’，需要写成r’I have a (?:dog|cat)’ ，而不能写成 r’I have a dog|cat’

例

>>> s = ‘I have a dog , I have a cat’

>>> re.findall( r’I have a (?:dog|cat)’ , s )

['I have a dog', 'I have a cat']                #正如我们所要的

下面再看看不用无捕获组会是什么后果：

>>> re.findall( r’I have a dog|cat’ , s )

['I have a dog', 'cat']                                   #它将’I have a dog’ 和’cat’当成两个规则了

至于无捕获组的使用，后面将仔细说明。这里先跳过。

**‘.’    匹配所有字符**

匹配除换行符’\n’外的所有字符。如果使用了’S’选项，匹配包括’\n’的所有字符。

       例：

       >>> s=’123 \n456 \n789’

       >>> findall(r‘.+’,s)

       ['123', '456', '789']

       >>> re.findall(r‘.+’ , s , re.S)

       ['123\n456\n789']

**‘^’和’$’ 匹配字符串开头和结尾**

注意’^’不能在‘[ ]’中，否则含意就发生变化，具体请看上面的’[‘ ‘]’说明。 在多行模式下，它们可以匹配每一行的行首和行尾。具体请看后面compile函数说明的’M’选项部分

**‘\d’ 匹配数字**

这是一个以’\’开头的转义字符，’\d’表示匹配一个数字，即等价于[0-9]

**‘\D’ 匹配非数字**

这个是上面的反集，即匹配一个非数字的字符，等价于[^0-9]。注意它们的大小写。下面我们还将看到Python的正则规则中很多转义字符的大小写形式，代表互补的关系。这样很好记。

**‘\w’ 匹配字母和数字**

匹配所有的英文字母和数字，即等价于[a-zA-Z0-9]。

**‘\W’ 匹配非英文字母和数字**

即’\w’的补集，等价于[^a-zA-Z0-9]。

**‘\s’ 匹配间隔符**

即匹配空格符、制表符、回车符等表示分隔意义的字符，它等价于[ \t\r\n\f\v]。（注意最前面有个空格)

**‘\S’ 匹配非间隔符**

即间隔符的补集，等价于[^ \t\r\n\f\v]

**‘\A’ 匹配字符串开头**

匹配字符串的开头。它和’^’的区别是，’\A’只匹配整个字符串的开头，即使在’M’模式下，它也不会匹配其它行的行首。

**‘\Z’ 匹配字符串结尾**

匹配字符串的结尾。它和’$’的区别是，’\Z’只匹配整个字符串的结尾，即使在’M’模式下，它也不会匹配其它各行的行尾。

例：

>>> s= '12 34\n56 78\n90'

>>> re.findall( r'^\d+' , s , re.M )          #匹配位于行首的数字

['12', '56', '90']

>>> re.findall( r’\A\d+’, s , re.M )        #匹配位于字符串开头的数字

['12']

>>> re.findall( r'\d+$' , s , re.M )          #匹配位于行尾的数字

['34', '78', '90']

>>> re.findall( r’\d+\Z’ , s , re.M )        #匹配位于字符串尾的数字

['90']

**‘\b’ 匹配单词边界**

它匹配一个单词的边界，比如空格等，不过它是一个‘0’长度字符，它匹配完的字符串不会包括那个分界的字符。而如果用’\s’来匹配的话，则匹配出的字符串中会包含那个分界符。

例：

>>> s =  'abc abcde bc bcd'

>>> re.findall( r’\bbc\b’ , s )         #匹配一个单独的单词 ‘bc’ ，而当它是其它单词的一部分的时候不匹配

['bc']                                           ＃只找到了那个单独的’bc’

>>> re.findall( r’\sbc\s’ , s )          ＃匹配一个单独的单词 ‘bc’

[' bc ']                                         #只找到那个单独的’bc’，不过注意前后有两个空格，可能有点看不清楚

**‘\B’ 匹配非边界**

和’\b’相反，它只匹配非边界的字符。它同样是个0长度字符。

接上例：

>>> re.findall( r’\Bbc\w+’ , s )     #匹配包含’bc’但不以’bc’为开头的单词

['bcde']                                       #成功匹配了’abcde’中的’bcde’，而没有匹配’bcd’

**‘(?:)’ 无捕获组**

当你要将一部分规则作为一个整体对它进行某些操作，比如指定其重复次数时，你需要将这部分规则用’(?:’ ‘)’把它包围起来，而不能仅仅只用一对括号，那样将得到绝对出人意料的结果。

例：匹配字符串中重复的’ab’

>>> s=’ababab abbabb aabaab’

>>> re.findall( r’\b(?:ab)+\b’ , s )

['ababab']

如果仅使用一对括号，看看会是什么结果：

>>> re.findall( r’b\(ab)+\b’ , s )

['ab']

这是因为如果只使用一对括号，那么这就成为了一个组(group)。组的使用比较复杂，将在后面详细讲解。

**‘(?# )’ 注释**

Python允许你在正则表达式中写入注释，在’(?#’ ‘)’之间的内容将被忽略。

**(?iLmsux) 编译选项指定**

Python的正则式可以指定一些选项，这个选项可以写在findall或compile的参数中，也可以写在正则式里，成为正则式的一部分。这在某些情况下会便利一些。具体的选项含义请看后面的compile函数的说明。

此处编译选项’i’ 等价于IGNORECASE ，L 等价于 LOCAL ，m 等价于 MULTILINE ，s 等价于 DOTALL ，u 等价于 UNICODE ， x 等价于 VERBOSE 。

请注意它们的大小写。在使用时可以只指定一部分，比如只指定忽略大小写，可写为 ‘(?i)’，要同时忽略大小写并使用多行模式，可以写为 ‘(?im)’。

另外要注意选项的有效范围是整条规则，即写在规则的任何地方，选项都会对全部整条正则式有效。

**1.2 重复**

正则式需要匹配不定长的字符串，那就一定需要表示重复的指示符。Python的正则式表示重复的功能很丰富灵活。重复规则的一般的形式是在一条字符规则后面紧跟一个表示重复次数的规则，已表明需要重复前面的规则一定的次数。重复规则有：

**‘\*’   0或多次匹配**

表示匹配前面的规则0次或多次。

**‘+’   1次或多次匹配**

表示匹配前面的规则至少1次，可以多次匹配

例：匹配以下字符串中的前一部分是字母，后一部分是数字或没有的变量名字

>>> s = ‘ aaa bbb111 cc22cc 33dd ‘

>>> re.findall( r’\b[a-z]+\d\*\b’ , s )             #必须至少1个字母开头，以连续数字结尾或没有数字

['aaa', 'bbb111']

注意上例中规则前后加了表示单词边界的’\b’指示符，如果不加的话结果就会变成：

>>> re.findall( r’[a-z]+\d\*’ , s )

['aaa', 'bbb111', 'cc22', 'cc', 'dd']    #把单词给拆开了

大多数情况下这不是我们期望的结果。

**‘?’   0或1次匹配**

只匹配前面的规则0次或1次。

例，匹配一个数字，这个数字可以是一个整数，也可以是一个科学计数法记录的数字，比如123和10e3都是正确的数字。

>>> s = ‘ 123 10e3 20e4e4 30ee5 ‘

>>> re.findall( r’ \b\d+[eE]?\d\*\b’ , s )

['123', '10e3']

它正确匹配了123和10e3,正是我们期望的。注意前后的’\b’的使用，否则将得到不期望的结果。

**1.2.1 精确匹配和最小匹配**

Python正则式还可以精确指定匹配的次数。指定的方式是

**‘{m}’      精确匹配m次**

**‘{m,n}’   匹配最少m次，最多n次。(n>m)**

如果你只想指定一个最少次数或只指定一个最多次数，你可以把另外一个参数空起来。比如你想指定最少3次，可以写成 {3,} （注意那个逗号），同样如果只想指定最大为5次，可以写成{，5}，也可以写成{0,5}。

例 寻找下面字符串中

a：3位数

b: 2位数到4位数

c: 5位数以上的数

d: 4位数以下的数

>>> s= ‘ 1 22 333 4444 55555 666666 ‘

>>> re.findall( r’\b\d{3}\b’ , s )            # a：3位数

['333']

>>> re.findall( r’\b\d{2,4}\b’ , s )         # b: 2位数到4位数

['22', '333', '4444']

>>> re.findall( r’\b\d{5,}\b’, s )           # c: 5位数以上的数

['55555', '666666']

>>> re.findall( r’\b\d{1,4}\b’ , s )         # 4位数以下的数

['1', '22', '333', '4444']

**‘\*?’ ‘+?’ ‘??’ 最小匹配**

‘\*’ ‘+’ ‘?’通常都是尽可能多的匹配字符。有时候我们希望它尽可能少的匹配。比如一个c语言的注释 ‘/\* part 1 \*/ /\* part 2 \*/’，如果使用最大规则：

>>> s =r ‘/\* part 1 \*/ code /\* part 2 \*/’

>>> re.findall( r’/\\*.\*\\*/’ , s )

[‘/\* part 1 \*/ code /\* part 2 \*/’]

结果把整个字符串都包括进去了。如果把规则改写成

>>> re.findall( r’/\\*.\*?\\*/’ , s )                    #在\*后面加上?，表示尽可能少的匹配

['/\* part 1 \*/', '/\* part 2 \*/']

结果正确的匹配出了注释里的内容

**1.3   前向界定与后向界定**

有时候需要匹配一个跟在特定内容后面的或者在特定内容前面的字符串，Python提供一个简便的前向界定和后向界定功能，或者叫前导指定和跟从指定功能。它们是：

**‘(?<=…)’ 前向界定**

括号中’…’代表你希望匹配的字符串的前面应该出现的字符串。

**‘(?=…)’  后向界定**

括号中的’…’代表你希望匹配的字符串后面应该出现的字符串。

例： 你希望找出c语言的注释中的内容，它们是包含在’/\*’和’\*/’之间，不过你并不希望匹配的结果把’/\*’和’\*/’也包括进来，那么你可以这样用：

>>> s=r’/\* comment 1 \*/  code  /\* comment 2 \*/’

>>> re.findall( r’(?<=/\\*).+?(?=\\*/)’ , s )

[' comment 1 ', ' comment 2 ']

注意这里我们仍然使用了最小匹配，以避免把整个字符串给匹配进去了。

要注意的是，前向界定括号中的表达式必须是常值，也即你不可以在前向界定的括号里写正则式。比如你如果在下面的字符串中想找到被字母夹在中间的数字，你不可以用前向界定：

例：

>>> s = ‘aaa111aaa , bbb222 , 333ccc ‘

>>> re.findall( r’(?<=[a-z]+)\d+(?=[a-z]+)' , s )          # 错误的用法

它会给出一个错误信息：

error: look-behind requires fixed-width pattern

不过如果你只要找出后面接着有字母的数字，你可以在后向界定写正则式：

>>> re.findall( r’\d+(?=[a-z]+)’, s )

['111', '333']

如果你一定要匹配包夹在字母中间的数字，你可以使用组（group）的方式

>>> re.findall (r'[a-z]+(\d+)[a-z]+' , s )

['111']

组的使用将在后面详细讲解。

除了前向界定前向界定和后向界定外，还有前向非界定和后向非界定，它的写法为：

**‘(?<!...)’前向非界定**

只有当你希望的字符串前面不是’…’的内容时才匹配

**‘(?!...)’后向非界定**

只有当你希望的字符串后面不跟着’…’内容时才匹配。

接上例，希望匹配后面不跟着字母的数字

>>> re.findall( r’\d+(?!\w+)’ , s )

['222']

注意这里我们使用了\w而不是像上面那样用[a-z]，因为如果这样写的话，结果会是：

>>> re.findall( r’\d+(?![a-z]+)’ , s )

['11', '222', '33']

这和我们期望的似乎有点不一样。它的原因，是因为’111’和’222’中的前两个数字也是满足这个要求的。因此可看出，正则式的使用还是要相当小心的，因为我开始就是这样写的，看到结果后才明白过来。不过Python试验起来很方便，这也是脚本语言的一大优点，可以一步一步的试验，快速得到结果，而不用经过烦琐的编译、链接过程。也因此学习Python就要多试，跌跌撞撞的走过来，虽然曲折，却也很有乐趣。

**1.4 组的基本知识**

上面我们已经看过了Python的正则式的很多基本用法。不过如果仅仅是上面那些规则的话，还是有很多情况下会非常麻烦，比如上面在讲前向界定和后向界定时，取夹在字母中间的数字的例子。用前面讲过的规则都很难达到目的，但是用了组以后就很简单了。

**‘(‘’)’       无命名组**

最基本的组是由一对圆括号括起来的正则式。比如上面匹配包夹在字母中间的数字的例子中使用的(\d+)，我们再回顾一下这个例子：

>>> s = ‘aaa111aaa , bbb222 , 333ccc ‘

>>> re.findall (r'[a-z]+(\d+)[a-z]+' , s )

['111']

**可以看到findall函数只返回了包含在’()’中的内容，而虽然前面和后面的内容都匹配成功了，却并不包含在结果中。**

除了最基本的形式外，我们还可以给组起个名字，它的形式是

**‘(?P<name>…)’ 命名组**

‘(?P’代表这是一个Python的语法扩展’<…>’里面是你给这个组起的名字，比如你可以给一个全部由数字组成的组叫做’num’，它的形式就是’(?P<num>\d+)’。起了名字之后，我们就可以在后面的正则式中通过名字调用这个组，它的形式是

**‘(?P=name)’ 调用已匹配的命名组**

要注意，再次调用的这个组是已被匹配的组，也就是说它里面的内容是和前面命名组里的内容是一样的。

我们可以看更多的例子：请注意下面这个字符串各子串的特点。

>>> s='aaa111aaa,bbb222,333ccc,444ddd444,555eee666,fff777ggg'

我们看看下面的正则式会返回什么样的结果：

>>> re.findall( r'([a-z]+)\d+([a-z]+)' , s )             # 找出中间夹有数字的字母

[('aaa', 'aaa'), ('fff', 'ggg')]

>>> re.findall( r '(?P<g1>[a-z]+)\d+(?P=g1)' , s ) #找出被中间夹有数字的前后同样的字母

['aaa']

>>> re.findall( r'[a-z]+(/d+)([a-z]+)' , s )             #找出前面有字母引导，中间是数字，后面是字母的字符串中的中间的数字和后面的字母

[('111', 'aaa'), ('777', 'ggg')]

我们可以通过命名组的名字在后面调用已匹配的命名组，不过名字也不是必需的。

**‘\number’             通过序号调用已匹配的组**

正则式中的每个组都有一个序号，序号是按组从左到右，从1开始的数字，你可以通过下面的形式来调用已匹配的组

比如上面找出被中间夹有数字的前后同样的字母的例子，也可以写成：

>>> re.findall( r’([a-z]+)\d+\1’ , s )

['aaa']

结果是一样的。

我们再看一个例子

>>> s='111aaa222aaa111 , 333bbb444bb33'

>>> re.findall( r'(\d+)([a-z]+)(\d+)(\2)(\1)' , s )           #找出完全对称的 数字－字母－数字－字母－数字 中的数字和字母

[('111', 'aaa', '222', 'aaa', '111')]

Python2.4以后的re模块，还加入了一个新的条件匹配功能

**‘(?(*id/name*)yes-pattern|no-pattern)’ 判断指定组是否已匹配，执行相应的规则**

这个规则的含义是，如果id/name指定的组在前面匹配成功了，则执行yes-pattern的正则式，否则执行no-pattern的正则式。

举个例子，比如要匹配一些形如 usr@mail 的邮箱地址，不过有的写成< usr@mail >即用一对<>括起来，有点则没有，要匹配这两种情况，可以这样写

>>> s='<usr1@mail1>  usr2@maill2'

>>> re.findall( r'(<)?\s\*(\[w+@\w+)\s\*(?(1)>)'](mailto:w+@w+%29/s*%28?%281%29%3E%29%27) , s )

[('<', 'usr1@mail1'), ('', 'usr2@maill2')]

不过如果目标字符串如下

>>> s='<usr1@mail1>  usr2@maill2 <usr3@mail3   usr4@mail4>  < usr5@mail5 '

而你想得到要么由一对<>包围起来的一个邮件地址，要么得到一个没有被<>包围起来的地址，但不想得到一对<>中间包围的多个地址或不完整的<>中的地址，那么使用这个式子并不能得到你想要的结果

>>> re.findall( r'(<)?\s\*(\[w+@\w+)\s\*(?(1)>)'](mailto:w+@w+%29/s*%28?%281%29%3E%29%27) , s )

[('<', 'usr1@mail1'), ('', 'usr2@maill2'), ('', 'usr3@mail3'), ('', 'usr4@mail4'), ('', 'usr5@mail5')]

它仍然找到了所有的邮件地址。

想要实现这个功能，单纯的使用findall有点吃力，需要使用其它的一些函数，比如match或search函数，再配合一些控制功能。这部分的内容将在下面详细讲解。

小结：以上基本上讲述了Python正则式的语法规则。虽然大部分语法规则看上去都很简单，可是稍不注意，仍然会得到与期望大相径庭的结果，所以要写好正则式，需要仔细的体会正则式规则的含义后不同规则之间细微的差别。

详细的了解了规则后，再配合后面就要介绍的功能函数，就能最大的发挥正则式的威力了。

<FONT style="FONT-SIZE: 10.5pt" face=""">

**2 re模块的基本函数**

在上面的说明中，我们已经对re模块的基本函数 ‘findall’很熟悉了。当然如果光有findall的话，很多功能是不能实现的。下面开始介绍一下re模块其它的常用基本函数。灵活搭配使用这些函数，才能充分发挥Python正则式的强大功能。

首先还是说下老熟人findall函数吧

**findall(rule , target [,flag] )**

在目标字符串中查找符合规则的字符串。

第一个参数是规则，第二个参数是目标字符串，后面还可以跟一个规则选项（选项功能将在compile函数的说明中详细说明）。

返回结果结果是一个**列表，**中间存放的是符合规则的字符串。如果没有符合规则的字符串被找到，就返回一个**空**列表。

**2.1 使用compile加速**

**compile( rule [,flag] )**

将正则规则编译成一个Pattern对象，以供接下来使用。

第一个参数是规则式，第二个参数是规则选项。

返回一个Pattern对象

直接使用findall ( rule , target )的方式来匹配字符串，一次两次没什么，如果是多次使用的话，由于正则引擎每次都要把规则解释一遍，而规则的解释又是相当费时间的，所以这样的效率就很低了。如果要多次使用同一规则来进行匹配的话，可以使用re.compile函数来将规则预编译，使用编译过返回的Regular Expression Object或叫做Pattern对象来进行查找。

例

>>> s='111,222,aaa,bbb,ccc333,444ddd'

>>> rule=r’\b\d+\b’

>>> compiled\_rule=re.compile(rule)

>>> compiled\_rule.findall(s)

['111', '222']

可见使用compile过的规则使用和未编译的使用很相似。compile函数还可以指定一些规则标志，来指定一些特殊选项。多个选项之间用 ’**|**’（位或）连接起来。

**I**      **IGNORECASE** 忽略大小写区别。

**L   LOCAL** 字符集本地化。这个功能是为了支持多语言版本的字符集使用环境的，比如在转义符\w，在英文环境下，它代表[a-zA-Z0-9]，即所以英文字符和数字。如果在一个法语环境下使用，缺省设置下，不能匹配"é" 或 "ç"。加上这L选项和就可以匹配了。不过这个对于中文环境似乎没有什么用，它仍然不能匹配中文字符。

**M    MULTILINE** 多行匹配。在这个模式下’^’(代表字符串开头)和’$’(代表字符串结尾)将能够匹配多行的情况，成为行首和行尾标记。比如

>>> s=’123 456\n789 012\n345 678’

>>> rc=re.compile(r’^\d+’)    #匹配一个位于开头的数字，没有使用M选项

>>> rc.findall(s)

['123']             #结果只能找到位于第一个行首的’123’

>>> rcm=re.compile(r’^\d+’,re.M)       #使用 M 选项

>>> rcm.findall(s)

['123', '789', '345']  #找到了三个行首的数字

同样，对于’$’来说，没有使用M选项，它将匹配最后一个行尾的数字，即’678’，加上以后，就能匹配三个行尾的数字456 012和678了.

>>> rc=re.compile(r’\d+$’)

>>> rcm=re.compile(r’\d+$’,re.M)

>>> rc.findall(s)

['678']

>>> rcm.findall(s)

['456', '012', '678']

**S     DOTALL** ‘.’号将匹配所有的字符。缺省情况下’.’匹配除换行符’\n’外的所有字符，使用这一选项以后，’.’就能匹配包括’\n’的任何字符了。

**U**   **UNICODE       \**w, \W, \b, \B, \d, \D, \s 和 \S都将使用Unicode。

**X     VERBOSE**     这个选项忽略规则表达式中的空白，并允许使用’#’来引导一个注释。这样可以让你把规则写得更美观些。比如你可以把规则

>>> rc = re.compile(r"\d+|[a-zA-Z]+")       #匹配一个数字或者单词

使用X选项写成：

>>> rc = re.compile(r"""  # start a rule

\d+                   # number

| [a-zA-Z]+           # word

""", re.VERBOSE)

在这个模式下，如果你想匹配一个空格，你必须用'\ '的形式（'\'后面跟一个空格）

**2.2 match与search**

**match( rule , targetString [,flag] )**

**search( rule , targetString [,flag] )**

（注：re的match 与search函数同compile过的Pattern对象的match与search函数的参数是不一样的。Pattern对象的match与search函数更为强大，是真正最常用的函数）

按照规则在目标字符串中进行匹配。

第一个参数是正则规则，第二个是目标字符串，第三个是选项（同compile函数的选项）

返回：若成功返回一个Match对象，失败无返回

findall虽然很直观，但是在进行更复杂的操作时，就有些力不从心了。此时更多的使用的是match和search函数。他们的参数和findall是一样的，都是：

match( rule , targetString [,flag] )

search( rule , targetString [,flag] )

不过它们的返回不是一个简单的字符串列表，而是一个MatchObject （如果匹配成功的话）.。通过操作这个matchObject，我们可以得到更多的信息。

需要注意的是，如果匹配不成功，它们则返回一个NoneType。所以在对匹配完的结果进行操作之前，你必需先判断一下是否匹配成功了，比如：

>>> m=re.match( rule , target )

>>> if m:                       #必需先判断是否成功

        doSomethin

这两个函数唯一的区别是：match从字符串的开头开始匹配，如果开头位置没有匹配成功，就算失败了；而search会跳过开头，继续向后寻找是否有匹配的字符串。针对不同的需要，可以灵活使用这两个函数。

关于match返回的MatchObject如果使用的问题，是Python正则式的精髓所在，它与组的使用密切相关。我将在下一部分详细讲解，这里只举个最简单的例子：

例：

>>> s= 'Tom:9527 , Sharry:0003'

>>> m=re.match( r'(?P<name>\w+):(?P<num>\d+)' , s )

>>> m.group()

'Tom:9527'

>>> m.groups()

('Tom', '9527')

>>> m.group(‘name’)

'Tom'

>>> m.group(‘num’)

'9527'

**2.3 finditer**

**finditer( rule , target [,flag] )**

参数同findall

返回一个迭代器

finditer函数和findall函数的区别是，findall返回所有匹配的字符串，并存为一个列表，而finditer则并不直接返回这些字符串，而是返回一个迭代器。关于迭代器，解释起来有点复杂，还是看看例子把：

>>> s=’111 222 333 444’

>>> for i in re.finditer(r’\d+’ , s ):

        print i.group(),i.span()          #打印每次得到的字符串和起始结束位置

结果是

111 (0, 3)

222 (4, 7)

333 (8, 11)

444 (12, 15)

简单的说吧，就是finditer返回了一个可调用的对象，使用 for i in finditer()的形式，可以一个一个的得到匹配返回的 Match对象。这在对每次返回的对象进行比较复杂的操作时比较有用。

**2.4 字符串的替换和修改**

re模块还提供了对字符串的替换和修改函数，他们比字符串对象提供的函数功能要强大一些。这几个函数是

**sub ( rule , replace , target [,count] )**

**subn(rule , replace , target [,count] )**

在目标字符串中规格规则查找匹配的字符串，再把它们替换成指定的字符串。你可以指定一个最多替换次数，否则将替换所有的匹配到的字符串。

第一个参数是正则规则，第二个参数是指定的用来替换的字符串，第三个参数是目标字符串，第四个参数是最多替换次数。

这两个函数的唯一区别是返回值。

sub返回一个被替换的字符串

sub返回一个元组，第一个元素是被替换的字符串，第二个元素是一个数字，表明产生了多少次替换。

例，将下面字符串中的’dog’全部替换成’cat’

>>> s=’ I have a dog , you have a dog , he have a dog ‘

>>> re.sub( r’dog’ , ‘cat’ , s )

' I have a cat , you have a cat , he have a cat '

如果我们只想替换前面两个，则

>>> re.sub( r’dog’ , ‘cat’ , s , 2 )

' I have a cat , you have a cat , he have a dog '

或者我们想知道发生了多少次替换，则可以使用subn

>>> re.subn( r’dog’ , ‘cat’ , s )

(' I have a cat , you have a cat , he have a cat ', 3)

**split( rule , target [,maxsplit] )**

切片函数。使用指定的正则规则在目标字符串中查找匹配的字符串，用它们作为分界，把字符串切片。

第一个参数是正则规则，第二个参数是目标字符串，第三个参数是最多切片次数

返回一个被切完的子字符串的列表

这个函数和str对象提供的split函数很相似。举个例子，我们想把上例中的字符串被’,’分割开，同时要去掉逗号前后的空格

>>> s=’ I have a dog   ,   you have a dog  ,  he have a dog ‘

>>> re.split( ‘\s\*,\s\*’ , s )

[' I have a dog', 'you have a dog', 'he have a dog ']

结果很好。如果使用str对象的split函数，则由于我们不知道’,’两边会有多少个空格，而不得不对结果再进行一次处理。

**escape( string )**

这是个功能比较古怪的函数，它的作用是将字符串中的non-alphanumerics字符（我已不知道该怎么翻译比较好了）用反义字符的形式显示出来。有时候你可能希望在正则式中匹配一个字符串，不过里面含有很多re使用的符号，你要一个一个的修改写法实在有点麻烦，你可以使用这个函数,

例 在目标字符串s中匹配’(\*+?)’这个子字符串

>>> s= ‘111 222 (\*+?) 333’

>>> rule= re.escape( r’(\*+?)’ )

>>> print rule

\\*\+\?

>>> re.findall( rule , s )

['(\*+?)']

<FONT style="FONT-SIZE: 10.5pt" face=""">

3     **更深入的了解re的组与对象**

前面对Python正则式的组进行了一些简单的介绍，由于还没有介绍到match对象，而组又是和match对象密切相关的，所以必须将它们结合起来介绍才能充分地说明它们的用途。

不过再详细介绍它们之前，我觉得有必要先介绍一下将规则编译后的生成的patter对象

**3.1编译后的Pattern对象**

将一个正则式，使用compile函数编译，不仅是为了提高匹配的速度，同时还能使用一些附加的功能。编译后的结果生成一个Pattern对象，这个对象里面有很多函数，他们看起来和re模块的函数非常象，它同样有findall , match , search ,finditer , sub , subn , split 这些函数，只不过它们的参数有些小小的不同。一般说来，re模块函数的第一个参数，即正则规则不再需要了，应为规则就包含在Pattern对象中了，编译选项也不再需要了，因为已经被编译过了。因此re模块中函数的这两个参数的位置，就被后面的参数取代了。

findall , match , search 和finditer这几个函数的参数是一样的，除了少了规则和选项两个参数外，它们又加入了另外两个参数，它们是：查找开始位置和查找结束位置，也就是说，现在你可以指定查找的区间，除去你不感兴趣的区间。它们现在的参数形式是：

**findall ( targetString [, startPos [,endPos] ] )**

**finditer ( targetString [, startPos [,endPos] ] )**

**match ( targetString [, startPos [,endPos] ] )**

**search ( targetString [, startPos [,endPos] ] )**

这些函数的使用和re模块的同名函数使用完全一样。所以就不多介绍了。

除了和re模块的函数同样的函数外，Pattern对象还多了些东西，它们是：

**flags       查询编译时的选项**

**pattern 查询编译时的规则**

**groupindex 规则里的组**

这几个不是函数，而是一个值。它们提供你一些规则的信息。比如下面这个例子

>>> p=re.compile( r'(?P<word>\b[a-z]+\b)|(?P<num>\b\d+\b)|(?P<id>\b[a-z\_]+\w\*\b)' , re.I )

>>> p.flags

2

>>> p.pattern

'(?P<word>\\b[a-z]+\\b)|(?P<num>\\b\\d+\\b)|(?P<id>\\b[a-z\_]+\\w\*\\b)'

>>> p.groupindex

{'num': 2, 'word': 1, 'id': 3}

我们来分析一下这个例子：这个正则式是匹配单词、或数字、或一个由字母或’\_’开头，后面接字母或数字的一个ID。我们给这三种情况的规则都包入了一个命名组，分别命名为’word’ , ‘num’ 和 ‘id’。我们规定大小写不敏感，所以使用了编译选项 ‘I’。

编译以后返回的对象为p，通过p.flag我们可以查看编译时的选项，不过它显示的不是’I’，而是一个数值2 。其实re.I是一个整数，2就是它的值。我们可以查看一下：

>>> re.I

2

>>> re.L

4

>>> re.M

8

…

每个选项都是一个数值。

通过p.pattern可以查看被编译的规则是什么。使用print的话会更好看一些

>>> print p.pattern

(?P<word>\b[a-z]+\b)|(?P<num>\b\d+\b)|(?P<id>\b[a-z\_]+\w\*\b)

看，和我们输入的一样。

接下来的p.groupindex则是一个字典，它包含了规则中的所有命名组。字典的key是名字，values是组的序号。由于字典是以名字作为key，所以一个无命名的组不会出现在这里。

**3.2 组与Match对象**

组与Match对象是Python正则式的重点。只有掌握了组和Match对象的使用，才算是真正学会了Python正则式。

**3.2.1 组的名字与序号**

正则式中的每个组都有一个序号，它是按定义时从左到右的顺序从1开始编号的。其实，re的正则式还有一个0号组，它就是整个正则式本身。

我们来看个例子

>>> p=re.compile( r’(?P<name>[a-z]+)\s+(?P<age>\d+)\s+(?P<tel>\d+).\*’ , re.I )

>>> p.groupindex

{'age': 2, 'tel': 3, 'name': 1}

>>> s=’Tom 24 88888888  <=’

>>> m=p.search(s)

>>> m.groups()                           # 看看匹配的各组的情况

('Tom', '24', '8888888')

>>> m.group(‘name’)                   # 使用组名获取匹配的字符串

‘Tom’

>>> m.group( 1 )                         # 使用组序号获取匹配的字符串，同使用组名的效果一样

>>> m.group(0)                           # 0 组里面是什么呢？

'Tom 24 88888888  <='

原来0组就是整个正则式,包括没有被包围到组里面的内容。当获取0组的时候，你可以不写这个参数。m.group(0)和m.group()的效果是一样的：

>>> m.group()

'Tom 24 88888888  <='

接下来看看更多的Match对象的方法，看看我们能做些什么。

**3.2.2 Match对象的方法**

**group([index|id])** 获取匹配的组，缺省返回组0,也就是全部值

**groups()**               返回全部的组

**groupdict()**           返回以组名为key，匹配的内容为values的字典

接上例：

>>> m.groupindex()

{'age': '24', 'tel': '88888888', 'name': 'Tom'}

**start( [group] )** 获取匹配的组的开始位置

**end( [group] )** 获取匹配的组的结束位置

**span( [group] )** 获取匹配的组的（开始，结束）位置

**expand( template )** 根据一个模版用找到的内容替换模版里的相应位置

这个功能比较有趣，它根据一个模版来用匹配到的内容替换模版中的相应位置，组成一个新的字符串返回。它使用\g<index|name>或 \index 来指示一个组。

接上例

>>> m.expand(r'name is \g<1> , age is \g<age> , tel is \3')

'name is Tom , age is 24 , tel is 88888888'

除了以上这些函数外，Match对象还有些属性

**pos**         搜索开始的位置参数

**endpos**  搜索结束的位置参数

这两个是使用findall或match等函数时，传入的参数。在上面这个例子里，我们没有指定开始和结束位置，那么缺省的开始位置就是0,结束位置就是最后。

>>> m.pos

0

>>> m.endpos

19

**lastindex** 最后匹配的组的序号

>>> m.lastindex

3

**lastgroup** 最后匹配的组名

>>> m.lastgroup

'tel'

**re**    产生这个匹配的Pattern对象，可以认为是个逆引用

>>> m.re.pattern

'(?P<name>[a-z]+)\\s+(?P<age>\\d+)\\s+(?P<tel>\\d+).\*'

得到了产生这个匹配的规则

**string** 匹配的目标字符串

>>> m.string

'Tom 24 88888888  <='

**4 更多的资料**

以上基本上是把Python正则式的全面的介绍了一遍了。基本上是涵盖了Python帮助中有关正则式的全部内容。本来是想再多举点例子的，不过一来有点累了，二来觉得例子也举得够多的了，大家还是要靠自己多用多试，才能真正体会到Python 正则式的精髓。

这篇文章只能算是个学习笔记。我本身也是个初学者，错漏的地方难免，所以最可靠的，还是Python自带的帮助。以及网络上寻找的一些资料。

写这篇文章的时候我参考了以下的资料

《Python正则表达式操作指南》

<http://wiki.ubuntu.org.cn/index.php?title=Python%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F%E6%93%8D%E4%BD%9C%E6%8C%87%E5%8D%97&variant=zh-cn>