🡪re模块方法

|  |  |
| --- | --- |
| 方法名 | 描述 |
| re.match | re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回none。 |
| re.search(<regex>, <string>, <flags>) | re.search 扫描整个字符串并返回第一个成功的匹配match对象。  re.match与re.search的区别:  re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。 |
| re.sub | 替换字符串中的匹配项。 |

🡪正则表达式修饰符-可选标志

|  |  |
| --- | --- |
| 修饰符 | 描述 |
| re.I | 使匹配对大小写不敏感 |
| re.L | 做本地化识别匹配 |
| re.M | 多行匹配，影响`和$。  不指定re.M情况下，^匹配整体字符串的开头，$匹配整体字符串的结尾。  指定re.M情况下，^匹配整体字符串的开头，以及每一行的开头。$匹配整体字符串的结尾，以及每一行的结尾。 |
| re.S/re.DOTALL | 当指定时候，使.匹配包括换行在内的所有字符。  当不指定时候，使.匹配除了换行符以外的所有字符。 |
| re.U | 根据Unicode字符集解析字符。这个标志影响\w, \W \b, \B |
| re.X | 允许在正则表达式中加入注释 |

🡪正则表达式特殊字符（元字符）

|  |  |
| --- | --- |
| 特殊字符 | 效果 |
| \w | 匹配字母数字及下划线  等价于[a-zA-Z0-9\_] |
| \W | 匹配非字母数字及下划线以外的所有字符  等价于[^a-zA-Z0-9\_] |
| \s | 匹配任意空白字符，等价于[ \t\n\r\f]。 |
| \S | 匹配任意非空字符 |
| \d | 匹配任意数字，等价于[0-9] |
| \D | 匹配任意非数字，等价于[^0-9] |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 |
| \B | 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。 |
| ^ | 🡪匹配字符串的开头。  🡪如果^出现在字符集[]首位当中，则表示非。  例如不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。 |
| $ | 匹配字符串的末尾。$在字符集[]中使用只代表普通字符$. |
| \* | 数量词，表示匹配0个或者多个 |
| + | 数量词，表示匹配1个或者多个 |
| ? | 🡪匹配0个或者一个  🡪表示量词\*,+,?的非贪婪（懒惰）匹配  🡪创建命名分组 |
| {m, n} | 表示匹配{}指定的数量范围.最少为m次，最多为n次  例如{2,4}匹配最少2次，最多4次 |
| \ | 🡪转义字符，取消元字符的特殊含义  🡪表示使用元字符的特殊含义  🡪在同一个正则表达式中，引用匹配分组 |
| | | 表示匹配多个正则表达式的或关系  例如：  >>> re.search('foo|bar|baz', 'baz')  <\_sre.SRE\_Match object; span=(0, 3), match='baz'>  表示匹配foo或者bar或者baz |
| . | 匹配除了换行符以外的任意字符。  当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。  特殊的，如果被.被[]包裹则, 即是[.]，则不表示匹配除换行以外得所有字符。只匹配单纯的.符号 |
| [...] | 用来表示一组字符集,  量词+, \*, ?在字符集[]当中失去特殊意义，只是简单以字符匹配+, \*, ?  例如：  re.search('ba[artz]', 'foobarqux')# 匹配字母a或者r或者t或者z  re.search('[a-z]', 'FOObar') # 匹配小写字母a到z |
| () | 表示建立匹配分组 |
| <> | 表示建立命名匹配分组 |

🡪

Python字符串类的replace方法，只能使用固定字符串不能指定正则表达式。

Javascript字符串的replace方法，可以使用固定字符串，也可以指定正则表达式。

Python正则表达式模块re.sub以及re.subn可以使用正则表达进行替换。

🡪

**Python使用正则表达式时候，推荐使用自然字符串r.**

因为在Python中使用正则表达式时候，整个表达式首先经过Python解释器解释，然后再被正则表达式解析器解析。

例如：

>>> re.search('\\', s)

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

re.search('\\', s)

File "C:\Python36\lib\re.py", line 182, in search

return \_compile(pattern, flags).search(string)

File "C:\Python36\lib\re.py", line 301, in \_compile

p = sre\_compile.compile(pattern, flags)

File "C:\Python36\lib\sre\_compile.py", line 562, in compile

p = sre\_parse.parse(p, flags)

File "C:\Python36\lib\sre\_parse.py", line 848, in parse

source = Tokenizer(str)

File "C:\Python36\lib\sre\_parse.py", line 231, in \_\_init\_\_

self.\_\_next()

File "C:\Python36\lib\sre\_parse.py", line 245, in \_\_next

self.string, len(self.string) - 1) from None

sre\_constants.error: bad escape (end of pattern) at position 0

为什么”\\”会报错？因为首先”\\“会经过Python解释器解释，Python解释器看到\\会认为第一个\取消第二个\的特殊转移。所以经过Python解析器后，它将正则表达式为”\”. 然后继续将其传到正则表达式的解析器解析，正则表达式的解析器收到”\”，它将报错。因为单独一个”\”没有任何意义。

解决办法：

方法1：要表达匹配单个纯字符”\”时候，写为臃肿的re.search('\\\\', s).则第一步经过Python解析器时候，\\\\变为\\.第二步正则表达式解析器收\\, 则第一个\转移第二个\，所以最终为匹配传字符\.但是这样的方法麻烦，易读性差。

方法2：>>> re.search(r'\\', s)。 使用自然字符串。 使用自然字符串以后，Python将跳过第一步Python解析器的处理。直接将正则字符串交给正则表达式解析器进行处理。从而正则表达式解析器收到\\,第一个\转移第二个\，所以最终为匹配传字符\.

🡪**Python正则表达式分组**

**Python正则表达式分组可以分为匿名分组以及显式命名分组。**

**匿名分组：在正则表达式中使用括号进行分组。匹配中使用\数字引用分组，在匹配后使用.group(数字)获取分组**

**例如：**

>>> m = re.search('(\w+),(\w+),(\w+)', 'foo,quux,baz') # 使用括号进行匿名分组

>>> m.groups()

('foo', 'quux', 'baz')

>>> m.group(1) # 匹配后使用数字获取分组

'foo'

>>> m.group(2)

'quux'

>>> m.group(3)

'baz'

>>> regex = r'(\w+),\1' # 使用括号进行匿名分组， 在匹配中使用\数字引用分组

>>> m = re.search(regex, 'foo,foo')

>>> m

<\_sre.SRE\_Match object; span=(0, 7), match='foo,foo'>

**命名分组：在正则表达式中使用(?P<name><regex>)进行分组。匹配中使用(?P=<name>)引用分组，在匹配后使用.group(分组名)获取分组**

**例如：**

>>> m = re.search('(?P<w1>\w+),(?P<w2>\w+),(?P<w3>\w+)', 'foo,quux,baz') # 使用(?P<name><regex>)进行命名分组

>>> m.group('w1') # 匹配后使用命名获取分组

'foo'

>>> m.group(1) # 同时，使用数字获取分组也还支持

'foo'

>>> m = re.search(r'(?P<word>\w+),(?P=word)', 'foo,foo') # 使用(?P<name><regex>)进行命名分组， 在匹配中使用(?P=<name>)引用分组

>>> m

<\_sre.SRE\_Match object; span=(0, 7), match='foo,foo'>

>>> m.group('word')

'foo'

**根据分组存在情况，进行不同的正则表达式匹配**

**对于匿名分组，如果序号为n的匹配分组存在，则使用<yes-regex>进行匹配。否则使用<no-regrx>进行匹配。  
(?(<n>)<yes-regex>|<no-regex>)**

**对于命名分组，如果名为<name>的匹配分组存在，则使用<yes-regex>进行匹配。否则使用<no-regrx>进行匹配。  
(?(<name>)<yes-regex>|<no-regex>)**

**例如对于匿名分组：**

regex = r'^(###)?foo(?(1)bar|baz)' # 如果(###)能被匹配则存在序号为1的匿名分组，如果存在则要求foo以后跟bar，否则要求foo以后跟baz.

>>> re.search(regex, '###foobar') ###存在，并且foo以后跟bar。匹配。

<\_sre.SRE\_Match object; span=(0, 9), match='###foobar'>

>>> print(re.search(regex, '###foobaz')) ###存在，foo以后并不跟bar。不匹配。

None

>>> print(re.search(regex, 'foobar')) ###不存在，foo以后并不跟baz。不匹配。

None

>>> re.search(regex, 'foobaz') ###不存在，foo以后跟baz。匹配。

<\_sre.SRE\_Match object; span=(0, 6), match='foobaz'>

**例如对于命名分组：**

>>> regex = r'^(?P<ch>\W)?foo(?(ch)(?P=ch)|)$' # 如果foo前面存在\W字符，则创建名为ch的命名分组。如果名为ch的命名分组存在则foo以后要跟该分组所捕获的字符，否则直接结尾。

>>> re.search(regex, 'foo') # foo前面不存在\W字符，没有名为ch的命名分组，foo以后直接结尾。匹配。

<\_sre.SRE\_Match object; span=(0, 3), match='foo'>

>>> re.search(regex, '#foo#') # foo前面存在\W字符，有名为ch的命名分组，foo以后跟该分组所捕获的字符。匹配。

<\_sre.SRE\_Match object; span=(0, 5), match='#foo#'>

>>> re.search(regex, '@foo@') # foo前面存在\W字符，有名为ch的命名分组，foo以后跟该分组所捕获的字符。

<\_sre.SRE\_Match object; span=(0, 5), match='@foo@'>

>>> print(re.search(regex, '#foo')) # foo前面存在\W字符，有名为ch的命名分组，foo以后没有跟该分组所捕获的字符。不匹配

None

>>> print(re.search(regex, 'foo@')) # foo前面存在\W字符，有名为ch的命名分组，foo以后没有直接结束。不匹配

None

>>> print(re.search(regex, '#foo@')) # foo前面存在\W字符，有名为ch的命名分组，foo以后没有跟该分组所捕获的字符。不匹配

None

>>> print(re.search(regex, '@foo#')) # foo前面存在\W字符，有名为ch的命名分组，foo以后没有跟该分组所捕获的字符。不匹配

None

**🡪**