Python程序设计

正则表达式

刘安
苏州大学，计算机科学与技术学院

http://web.suda.edu.cn/anliu/

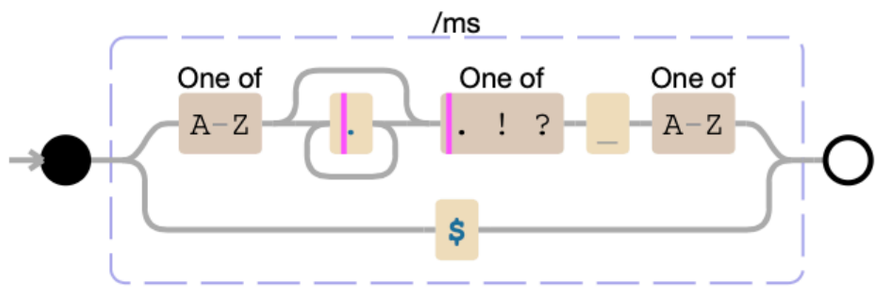
本节涉及到的知识点

- 正则表达式中的量词、预定义和自定义字符集和特别字符
- match, search, findall, finditer, split, sub, subn方法
- 分组、反向引用、命名分组和不捕获的分组
- 贪心和非贪心匹配
- 前瞻和否定前瞻机制
- 推荐网址
  - https://docs.python.org/3/howto/regex.html
  - https://www.debuggex.com

https://www.debuggex.com

Untitled Regex No description

Embed on StackOverflow



Python | View Cheatsheet | Flags: ms

```
1  [A-Z].*?[.!?] [A-Z]|$
```

Result: **Matches** starting at the black triangle slider

```
1  See the U.S.A. today. It is right here,
2  not a world away. Average temp. is 66.5.
```

START POSITION

正则表达式

- 一种描述文本模式的方法
  - 模式cat：匹配cat
  - 模式ca+t（+表示字符a可以出现1次或者多次）：匹配cat, caat, caaat, …

- 正则表达式包含两类字符
  - 简单字符：必须原样匹配，字母和数字通常是简单字符
  - 特别字符：改变其附近字符的含义，通常是标点符号
    - . ^ $ * + ? { } [ ] \ | ( )

量词

- 指定模式的出现次数
  - expr*：表达式expr出现0次或者多次
  - expr+：表达式expr出现1次或者多次
    - expr?：表达式expr出现0次或者1次
- ca*t：ct, cat, caat, caaat, …
- ca+t：cat, caat, caaat, …
- ca?t：ct, cat
- c(ab)*t：ct, cabt, cababt, cabababt, …

量词

- 指定模式的出现次数
  - expr{n}：表达式expr正好出现n次
  - expr{m, n}：表达式expr至少出现m次，最多出现n次
    - expr{0,1}等价于expr?
- expr{m, }：表达式expr至少出现m次
  - expr{0,}等价于expr*，expr{1,}等价于expr+
- expr{ , n}：表达式expr最多出现n次

预定义字符集

- \d：任意一个数字（0~9中的一个）
- \D：任意一个非数字（除了0~9之外的任意字符）
- \s：任意一个空白字符
- \S：任意一个非空白字符
- \w：任意一个单词字符（字母、数字、下划线）
- \W：任意一个非单词字符

预定义字符集

- \b：一个单词边界
  - 如果文本的首字符是\w，那么在其之前有一个\b
  - 如果文本的最后一个字符是\w，那么在其之后有一个\b
  - 文本中的任意两个字符，如果一个是\w，另一个是\W，那么它们之间存在一个\b
- \B：一个非单词边界
  - 如果文本的首字符是\W，那么在其之前有一个\B
  - 如果文本的最后一个字符是\W，那么在其之后有一个\B
  - 任意两个\w之间，以及任意两个\W之间，都有一个\B

Example

- 给定一个字符串'9+99+999=1107-123=984'，找出其中所有的整数以及所有的三位数
- 什么样的模式匹配一个整数？匹配一个三位数？

```
>>> import re
>>> msg = '9+99+999=1107-123=984'
>>> re.findall(r'\d+', msg)
['9', '99', '999', '1107', '123', '984']
>>>
>>> re.findall(r'\d{3}', msg)
['999', '110', '123', '984']
>>>
>>> re.findall(r'\b\d{3}\b', msg)
['999', '123', '984']
```

使用[ ]建立自定义字符集

- [ ]中列出的任一字符都和该模式匹配
  - 字母a、b、c都和模式[abc]匹配
  - [aeiouAEIOU]匹配一个元音字符（这里认为y不是元音）
- 可以单列出所有字符，也可以使用字符-来表达一个范围
  - [a-f]匹配a、b、c、d、e、f中的任意一个
  - [a-zA-Z0-9]匹配任意一个字母或数字
  - -必须出现在两个字符中间
  - X-Y表达的范围根据ASCII码来决定：[X-a]定义了哪些字符？

回顾预定义字符集

- \d：任意一个数字（0~9中的一个），等价于[0~9]
- \D：任意一个非数字，等价于[^0~9]
- \s：任意一个空白字符，等价于[\t\n\r\f\v]
- \S：任意一个非空白字符，等价于[^ \t\n\r\f\v]
- \w：任意一个单词字符，等价于[a-zA-Z0-9_]
- \W：任意一个非单词字符，等价于[^a-zA-Z0-9_]

使用[ ]时的注意事项

- []中的特殊字符一般按照普通字符对待
- 字符-如果在两个字符之间，则表示范围。但如果其在**最前面或者最后面**，比如[a-]，[- +]，还是按照普通字符对待
- 字符^如果在**最前面**，表示不在[]里面的字符都可以被匹配。否则将其看成普通字符（[^5]和[5^]有什么区别？）
- 要在[]中表示普通字符**右方括号]**，需要使用反斜杠\进行转义，比如[[\]]表示匹配[或者]（**注意左方括号[不需要转义**）
- 要在[]中表示反斜杠\，也需要使用反斜杠\进行转义，比如[\\]表示匹配\

Question

- 模式[+*/-]和模式[+-*/]各匹配什么？
- 模式[^+*/-]和模式[+*^/-]各匹配什么？

```
>>> msg = '1+2-3*4/5^6=?'
>>>
>>> re.findall(r'[+*^/-]', msg)
['+', '-', '*', '/', '^']
```

三重引号定义的多行字符串

- 多行字符串也称为块字符串，可以跨越多行
- 字符串中的\n表示换行

```
>>> msg = '''always look
on the bright
side of life.'''
>>>
>>> msg
'always look\non the bright\nside of life.'
>>>
>>> print(msg)
always look
on the bright
side of life.
```

通配字符

- 句点.匹配除了换行（\n）之外的所有字符

```
>>> import re
>>> msg = '''100,101
           10
           2,103, 10 4'''
>>> re.findall(r'1..', msg)
['100', '101', '103', '10 ']
>>>
>>> re.findall(r'1..', '100,101\n10\n2,103, 10 4')
['100', '101', '103', '10 ']
```

特别字符^

- ^expr匹配字符串开始处的expr，在多行模式下，匹配每一行开始处的expr，即紧接着换行\n后面的expr
  - 注意^在字符集[]中的用法
- \Aexpr仅仅匹配字符串开始处的expr（不在多行模式的时候，其与^expr等价）

```
>>> msg = '''from1
from2, from3.'''
>>> re.findall(r'^from\d', msg)
['from1']
>>> re.findall(r'^from\d', msg, re.M)
['from1', 'from2']
```

特别字符$

- expr$匹配字符串结束处的expr，在多行模式下，匹配每一行结束处的expr

- expr\Z仅仅匹配字符串结束处的expr（不在多行模式的时候，其与expr$等价）

```
>>> msg = '''from1 from2
from3'''
>>> re.findall(r'from\d$', msg)
['from3']
>>> re.findall(r'from\d$', msg, re.M)
['from2', 'from3']
```

- 思考：模式$expr匹配什么？模式[$]expr匹配什么？

特别字符|

- A|B匹配A与B当中的任意一个
- |的优先级非常低
  - ax|yb等价于(ax)|(yb)，匹配ax与yb当中的任意一个
  - a(x|y)b匹配axb和ayb当中的任意一个，等价于a[xy]b

```
>>> msg = '3 cats and 5 dogs.'
>>> re.findall(r'cat|dog', msg)
['cat', 'dog']
```

特别字符\进行转义

- \在一个普通字符之前意味着该普通字符不再作为字面量解释，而是具有特殊的含义
  - b是普通字符，匹配b，而\b匹配单词边界
- \在一个特殊字符之前意味着该特殊字符不再具有特殊的含义，而是作为字面量解释
- a*匹配0个或者任意多个a，而a*匹配a*
- []中的特殊字符通常看成普通字符，不需要转义
- \\表示后一个\不再作为特殊字符，用来匹配\

字符串字面量中的转义

- 反斜杠\可以在正则表达式中进行转义
- 字符串字面量中也存在转义情况

```
>>> msg = 'a\nb\tc\x0ad'
>>> print(msg)
a
b	c
d
>>> len(msg)
7
```

\\	反斜杠
\’	单引号
\”	双引号
\b	退格
\n	换行
\r	返回
\t	水平制表
\xhh	十六进制数

raw字符串

- 字母r（大写或小写）出现在字符串的第一个引号之前
  - 用来关闭转义机制

```
>>> path = 'C:\new\text.txt' #windows文件路径
>>> print(path)
C:
ew        ext.txt
>>> path = r'C:\new\text.txt' #raw字符串
>>> print(path)
C:\new\text.txt
```

用raw字符串描述正则表达式

- 定义一个正则表达式，用来匹配\section
  - 正则表达式\\section（正则表达式中\\用来匹配\）
  - 用字符串描述正则表达式\\section时，注意\需要使用\\进行转义表达，所以最终的字符串是\\\\section
- 为了简化表达，建议使用raw字符串，即r'\\section'

```
>>> msg = '\section 1 and \section 2'
>>> re.findall(r'\\section', msg)
['\\section', '\\section']
>>> re.findall('\\\\section', msg)
['\\section', '\\section']
>>> re.findall('\\section', msg)
[]
```

使用compile()编译正则表达式

- 使用raw字符串（假设名为pattern）描述一个正则表达式
- 导入re模块，使用re.compile()将pattern编译成一个正则表达式对象

```
>>> pattern = r'[a-z]+'
>>> p = re.compile(pattern)
>>> p
re.compile('[a-z]+')
>>> p = re.compile(pattern, re.I)
>>> p
re.compile('[a-z]+', re.IGNORECASE)
```

- re.compile(pattern[, flags])：可选的flags参数可以用来修改匹配的工作方式（详情见后）

正则表达式对象支持的方法

- match() - 在字符串开头查找匹配模式的文本，如找到返回一个匹配对象，否则返回None。

- search() - 在字符串中查找第一次匹配模式的文本，如找到返回一个匹配对象，否则返回None

- findall() - 从左向右查找所有匹配模式的文本（匹配的文本不能重叠），以列表的形式返回所有匹配文本

- finditer() - 从左向右查找所有匹配模式的文本（匹配的文本不能重叠），以迭代器形式返回所有匹配对象

- 更多方法见后

match/search返回的匹配对象

- match和search方法在匹配成功时返回一个匹配对象
- 该对象具有下列方法
  - group() - 返回匹配的文本
  - start() - 返回匹配文本在字符串中的开始位置（从0开始）
  - end() - 返回匹配文本在字符串中的结束位置（匹配文本最后一个字符的索引+1）
  - span() - 返回一个元组(start, end)，表示匹配位置信息
  - 更多方法见后

match/search返回的匹配对象

```
>>> p = re.compile(r'[a-z]+')
>>> print(p.match(''))
None
>>> m = p.match('python')
>>> m
<re.Match object; span=(0, 6), match='python'>
>>> m.group()
'python'
>>> m.start(), m.end()
(0, 6)
>>> m.span()
(0, 6)
```

match与search

- match() - 在字符串开头查找匹配模式的文本，如找到返回一个匹配对象，否则返回None。

- search() - 在字符串中查找第一次匹配模式的文本，如找到返回一个匹配对象，否则返回None

```
>>> p = re.compile(r'[a-z]+')
>>> print(p.match('...python'))
None
>>> m = p.search('...python')
>>> print(m)
<re.Match object; span=(3, 9), match='python'>
>>> m.group()
'python'
```

findall与finditer

- findall() - 以列表的形式返回所有匹配文本
- finditer() - 以迭代器形式返回所有匹配对象

```
>>> p = re.compile(r'\d+')
>>> p.findall('The sum of 9 and 99 is 108.')
['9', '99', '108']
>>> it = p.finditer('The sum of 9 and 99 is 108.')
>>> it
<callable_iterator object at 0x7fbd80134910>
>>> for match in it:
        print(match)


<re.Match object; span=(11, 12), match='9'>
<re.Match object; span=(17, 19), match='99'>
<re.Match object; span=(23, 26), match='108'>
```

re模块上的方法

- 在compile编译生成的正则表达式对象上调用各种方法，包括match, search, findall, finditer等

- 也可以直接在re模块上调用上述各种方法，此时，描述正则表达式的字符串作为第一个参数

- 第一种方法效率上相对而言更高（详情自行Google|Baidu)

```
>>> p = re.compile(r'\d+')
>>> p.match('2019-12-12') #编译产生的对象上调用match
<re.Match object; span=(0, 4), match='2019'>
>>>
>>> re.match(r'\d+', '2019-12-12') #re模块上调用match
<re.Match object; span=(0, 4), match='2019'>
```

编译选项

- DOTALL, S - 句点.可以匹配任意字符，包括换行
- IGNORECASE, I - 匹配的时候忽略大小写
- MULTILINE, M - 特殊字符^和&作用在多行字符串中每一行，否则仅仅匹配字符串的开始和末尾

```
>>> msg = '''100,101
10
2,103, 10 4'''
>>>
>>> re.findall(r'1..', msg)
['100', '101', '103', '10 ']
>>>
>>> re.findall(r'1..', msg, re.S)
['100', '101', '10\n', '103', '10 ']
```

编译选项

- DOTALL, S - 句点.可以匹配任意字符，包括换行
- IGNORECASE, I - 匹配的时候忽略大小写
- MULTILINE, M - 特殊字符^和&作用在多行字符串中每一行，否则仅仅匹配字符串的开始和末尾

```
>>> msg = 'abc,ABd,aBe'
>>>
>>> re.findall(r'ab\w', msg)
['abc']
>>>
>>> re.findall(r'ab\w', msg, re.I)
['abc', 'ABd', 'aBe']
```

re模块 vs 正则表达式对象

```
>>> pwd1, pwd2 = 'A123', 'abcd'
>>> expr1, expr2 = r'.*[a-z]', r'.*\d'
>>> p1 = re.compile(expr1, re.I)
>>> p2 = re.compile(expr2)
>>> bool(p1.match(pwd1) and p2.match(pwd1))
True
>>> bool(p1.match(pwd2) and p2.match(pwd2))
False
>>> bool(re.match(expr1, pwd1, re.I) and
         re.match(expr2, pwd1))
True
>>> bool(re.match(expr1, pwd2, re.I) and
         re.match(expr2, pwd2))
False
```

直接在re模块上调用match方法

先通过compile函数编译，然后在生成的正则表达式对象上调用match方法

用()进行分组

- 在模式中可以使用()进行分组
- 作用
  - 指明量词的作用范围
  - 允许反向引用，实现更灵活的匹配和替换（稍后介绍）

```
>>> p = re.compile(r'(ab)*')
>>> print(p.match('abababababa'))
<re.Match object; span=(0, 8), match='abababab'>
```

用()进行分组

- 模式中可以有多个分组，分组还可以嵌套，每个分组都有一个编号，从1开始（不是从0开始），根据左括号的出现顺序确定其编号

```
>>> p = re.compile(r'(a(b)c)(d)')
>>> m = p.match('abcd')
>>> m
<re.Match object; span=(0, 4), match='abcd'>
>>> m.group()
'abcd'
>>> m.group(0) #等价于m.group()
'abcd'
>>> m.group(1), m.group(2), m.group(3)
('abc', 'b', 'd')
```

对分组进行反向引用

- 每个分组都有一个编号k，通过\k可以引用该分组
- 考虑模式\b(\w+)\s+\1\b
  - \b：单词边界
  - (\w+)：一个或者多个单词字符，匹配该模式的文本放在第1个分组中
  - \s+：一个或者多个空白字符
  - \1：反向引用，即第1个分组中的内容

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> m = p.search('Paris in the the spring')
>>> m.group()
'the the'
```

分组情况下匹配对象的方法

- match和search方法在匹配成功时返回一个匹配对象
- 该对象具有方法group(), start(), end(), span()
  - group(k)：返回第k个分组（从1开始编号）
  - group(0)等价于group()，返回整个匹配文本
  - groups()：以元组形式返回每个分组匹配的文本
  - lastindex：分组编号的最大值

分组情况下匹配对象的方法

```
>>> p = re.compile(r'(a+)(b+)(c+)')
>>> m = p.match('abbcceee')
>>> for i in range(m.lastindex + 1):
        print(i, '. ', m.group(i))
```

```
0 .  abbcc
1 .  a
2 .  bb
3 .  cc
>>>
>>> print(m.groups())
('a', 'bb', 'cc')
```

命名分组

- 不仅可以用整数对分组进行编号，还可以对分组进行命名
- (?P<name>expr)：name是分组的名字
- 可以通过匹配对象的group(name)方法获取分组name的文本

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search('(((( Lots of punctuation )))')
>>> m.group('word') #该分组的名字是word
'Lots'
>>> m.group(1) #命名分组也具有整数编号
'Lots'
```

命名分组

- (?P<name>expr)：name是分组的名字
- 可以通过匹配对象的group(name)方法获取分组name的文本
- 此外，匹配对象的groupdict()方法以字典形式返回每个分组的名字及其匹配的文本

```
>>> p = re.compile(r'(?P<first>\w+) (?P<last>\w+)')
>>> m = p.search('John Public')
>>> m.group('first'), m.group('last')
('John', 'Public')
>>> m.groupdict()
{'first': 'John', 'last': 'Public'}
```

反向引用命名分组

- (?P<name>expr)：name是分组的名字
- 引用方法：(?P=name)

```
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> m = p.search('Paris in the the spring')
>>> m.group()
'the the'
>>>
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b',
re.I)
>>> m = p.search('The the dog')
>>> m.group()
'The the'
```

查找数字

- 在一个字符串中找出所有的数字
  - 整数用千分位形式表示，即从个位数起，每3位之间加一个的逗号，比如1000000表示成1,000,000
  - 小数点之前至少有一位数字，比如0.618中0不可省略

- 先考虑整数部分的模式
  - 什么时候会出现逗号：(,\d{3})
  - 上述模式可以出现多少次：(,\d{3})*
  - 上述模式之前应该是什么？\d{1,3}

查找数字

- 在一个字符串中找出所有的数字
  - 整数用千分位形式表示，即从个位数起，每3位之间加一个的逗号，比如1000000表示成1,000,000
  - 小数点之前至少有一位数字，比如0.618中0不可省略
- 先考虑整数部分的模式：`\d{1,3}(,\d{3})*`
- 再考虑小数部分的模式
  - 小数点及后面的数字：`[.]\d+`
  - 上述模式可以出现多少次：`([.]\d+)?`

查找数字

- 在一个字符串中找出所有的数字
  - 整数用千分位形式表示，即从个位数起，每3位之间加一个的逗号，比如1000000表示成1,000,000
  - 小数点之前至少有一位数字，比如0.618中0不可省略
- 先考虑整数部分的模式：\d{1,3}(,\d{3})*
- 再考虑小数部分的模式：([.]\d+)?
- 最后的模式：\d{1,3}(,\d{3})*([.]\d+)?

查找数字

```
>>> p = re.compile(r'\d{1,3}(,\d{3})*([.]\d+)?')
>>> print(p.search('10'))
<re.Match object; span=(0, 2), match='10'>
>>> print(p.search('10.50'))
<re.Match object; span=(0, 5), match='10.50'>
>>> print(p.search('1,000'))
<re.Match object; span=(0, 5), match='1,000'>
>>> print(p.search('1,234,567.89'))
<re.Match object; span=(0, 12), match='1,234,567.89'>
```

查找数字

```
>>> p = re.compile(r'\d{1,3}(,\d{3})*([.]\d+)?')
>>> print(p.search('10'))
<re.Match object; span=(0, 2), match='10'>
>>> print(p.search('10.50'))
<re.Match object; span=(0, 5), match='10.50'>
>>> print(p.search('1,000'))
<re.Match object; span=(0, 5), match='1,000'>
>>> print(p.search('1,234,567.89'))
<re.Match object; span=(0, 12), match='1,234,567.89'>
```

```
>>> p.findall('12.34 cats and 1,234.5 dogs')
[('', '.34'), (',234', '.5')]
```

都是分组惹的祸

- findall方法的详细说明：Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. **If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group**. Empty matches are included in the result.

- 解决方案A：将整个模式放入一个分组，注意返回结果是一个元组列表，对于每一个元组，其第0个元素即为所求

```
>>> p = re.compile(r'\d{1,3}(,\d{3})*([.]\d+)?')
>>> p.findall('12.34 cats and 1,234.5 dogs')
[('', '.34'), (',234', '.5')]
```

都是分组惹的祸

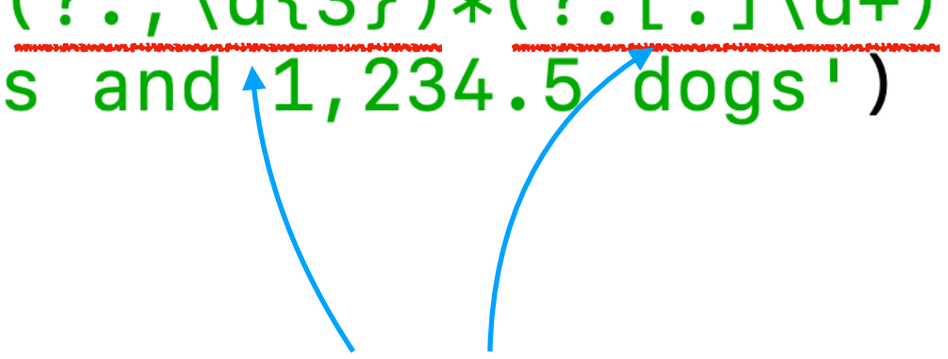
- 解决方案A：将整个模式放入一个分组，注意返回结果是一个元组列表，对于每一个元组，其第0个元素即为所求

```
>>> p = re.compile(r'\d{1,3}(,\d{3})*([.]\d+)?')
>>> p.findall('12.34 cats and 1,234.5 dogs')
[('', '.34'), (',234', '.5')]
>>>
>>> p = re.compile(r'(\d{1,3}(,\d{3})*([.]\d+)?)')
>>> m = p.findall('12.34 cats and 1,234.5 dogs')
>>> m
[('12.34', '', '.34'), ('1,234.5', ',234', '.5')]
>>>
>>> for t in m: print(t[0])

12.34
1,234.5
```

不捕获的分组

- 解决方案B：采用不捕获的分组
- 使用(?:expr)来说明不捕获当前分组，即当前分组仍然要求匹配，但不对该分组进行编号

```
>>> p = re.compile(r'\d{1,3}(?:,\d{3})*(?:[.]\d+)?')
>>> m = p.findall('12.34 cats and 1,234.5 dogs')
>>> m
['12.34', '1,234.5']
>>>
>>> for s in m: print(s)

12.34
1,234.5
```



不捕获的分组

- 使用(?:expr)来说明不捕获当前分组

```
>>> m = re.match(r'([abc])+', 'abc')
>>> m.groups()
('c',)
>>>
>>> m = re.match(r'(?:[abc])+', 'abc')
>>> m.groups()
()
>>>
>>> m = re.match(r'(([abc])+)', 'abc')
>>> m.groups()
('abc', 'c')
>>>
>>> m = re.match(r'((?:[abc])+)', 'abc')
>>> m.groups()
('abc',)
```

如果分组被重复匹配，只能记住最后一次匹配的文本

注意这里是如何记住整个匹配文本的

贪心和非贪心匹配

- 在'<h1>HTML heading</h1>'中找出一对尖括号<>中间的内容
- 因为尖括号不属于特殊字符，所以模式是<.*>

```
>>> p = re.compile(r'<.*>')
>>> m = p.search('<h1>HTML heading</h1>')
>>> m.group()
'<h1>HTML heading</h1>'
```

- <.*>先匹配一个<，然后匹配任意数量的字符（\n除外）
  - 贪心：.*匹配除了最后一个字符之外的所有字符，然后看最后一个字符是否和>匹配
  - 非贪心：.*匹配第一个>之前的所有字符（即h1）

在量词后面使用？指定非贪心匹配

- expr??：非贪心的0次或1次匹配
- expr+?：非贪心的1次或多次匹配
- expr*？：非贪心的0次或多次匹配
- expr{m,n}?：非贪心的最少m次最多n次匹配

```
>>> p = re.compile(r'<.*>') #默认情况下贪心匹配
>>> m = p.search('<h1>HTML heading</h1>')
>>> m.group()
'<h1>HTML heading</h1>'
>>>
>>> p = re.compile(r'<.*?>') #用?指定非贪心匹配
>>> m = p.search('<h1>HTML heading</h1>')
>>> m.group()
'<h1>'
```

统计句子个数

- 给定一个字符串，统计其中的句子个数，其中一个句子以句号、问号或者感叹号结尾
- .*?[.?!] - 这里红色问号表示非贪心匹配句子内容

```
>>> s = '''Here is a single sentence. Here is another sentence, ending with a period. And here is yet another!'''
>>> p = re.compile(r'.*?[.?!]', re.S)
>>> #上述表达式中第一个问号指定非贪心匹配
>>> #re.S使得.可以匹配换行
>>> lst = p.findall(s)
>>> print('There are', len(lst), 'sentences.')
There are 3 sentences.
```

如果不使用非贪心匹配，结果如何?

统计句子个数

- 给定一个字符串，统计其中的句子个数，其中一个句子以句号、问号或者感叹号结尾

- .*?[.?!] - 这里红色问号表示非贪心匹配句子内容

- 上述模式的问题：不是每一个句号都代表句子的结束

```
>>> s = 'The U.S.A. has 310.5 million people.'
>>> p = re.compile(r'.*?[.?!]')
>>> lst = p.findall(s)
>>> print('There are', len(lst), 'sentences.')
There are 5 sentences.
```

- 如何解决：遇到一个句号时，需要考虑其后面的文本模式

前瞻 (look-ahead) 机制

- 遇到一个句号时，需要考虑其后面的文本模式
  - 句号后面有一个空格，然后是一个大写字母（句子首单词的首字母需要大写）
  - 句号是字符串最后一个字符（即其后是字符串结束处）

- 模式A - 修改后的句子模式：[A-Z].*?[.!?]
- 模式B - 句号后文本应该匹配的模式：␣[A-Z]|$（注意[A-Z]之前的␣表示空格）

- 当文本s满足模式A，同时其后文本t又满足模式B时，s才是一个句子

前瞻（look-ahead）机制

- 模式A - 修改后的句子模式：[A-Z].*?[.!?]
- 模式B - 句号后文本应该匹配的模式：␣[A-Z]|$
- 只有文本s满足A，其后文本t又满足B时，s才是一个句子
- 模式C（将AB组合起来）：[A-Z].*?[.!?](␣[A-Z]|$)
  - 模式B匹配的文本t成为模式C的匹配结果，在判断下一个句子时，从文本t后面的文本开始（不是我们期望的！）
  - 模式B只是用来帮助模式A判断文本s是否是一个句子，与B匹配的文本不是句子s的一部分
- 前瞻机制：模式B匹配的文本t不作为当前模式C的匹配结果，即t参与下一次模式C的匹配过程

前瞻 (look-ahead) 机制

- A(?=B)：当前文本s与模式A(?=B)匹配当前仅当s与A匹配并且s后的文本t与B匹配，文本t仍然参与下一次匹配过程

- [A-Z].*?[.!?](?=␣[A-Z]|$)

```
>>> s = 'The U.S.A. has 310.5 million people.'
>>> p = re.compile(r'[A-Z].*?[.!?](? = [A-Z]|$)')
>>> lst = p.findall(s)
>>> print('There are', len(lst), 'sentence.')
There are 1 sentence.
```

前瞻 (look-ahead) 机制

```
>>> s = '''See the U.S.A. today. It's right here,
not a world away. Average temp. is 66.5.'''
>>>
>>> p = re.compile(r'[A-Z].*?[.!?](?= [A-Z]|$)',
                   re.S | re.M) #注意运算符|
>>> lst = p.findall(s)
>>> for i, sentence in enumerate(lst):
        print(i, ':', sentence)


0 : See the U.S.A. today.
1 : It's right here,
not a world away.
2 : Average temp. is 66.5.
```

为什么需要re.S？为什么需要re.M？如果不设置这些选项，在什么情况下会出错？

前瞻 (look-ahead) 机制

- 如果不使用前瞻机制，匹配效果如下

```
>>> s = '''See the U.S.A. today. It's right here,
not a world away. Average temp. is 66.5.'''
>>>
>>> p = re.compile(r'([A-Z].*?[.!?]( [A-Z]|$))',
                   re.S | re.M)
>>> lst = p.findall(s)
>>> lst
[('See the U.S.A. today. I', ' I'), ('Average temp.
is 66.5.', '')]
>>>
>>> for i, sentence in enumerate(lst):
        print(i, ':', sentence)


0 : ('See the U.S.A. today. I', ' I')
1 : ('Average temp. is 66.5.', '')
```

否定(Negative)前瞻机制

- A(?=B)：当前文本s与模式A(?=B)匹配当前仅当s与A匹配并且s后的文本t与B匹配，文本t仍然参与下一次匹配过程

- 否定前瞻机制

- A(?!B)：当前文本s与模式A(?!B)匹配当前仅当s与A匹配并且s后的文本t不与B匹配，文本t仍然参与下一次匹配过程

```
>>> s = 'The magic of abcABC.'
>>> p = re.compile(r'abc(?!abc)', re.I)
>>> lst = p.findall(s)
>>> lst
['ABC']
```

否定(Negative)前瞻机制

- 遇到一个句号，满足下面条件之一即为一个句子（前瞻）
  - 句号后面有一个空格，然后是一个大写字母（句子首单词的首字母需要大写）
  - 句号是字符串最后一个字符（即其后是字符串结束处）
- 遇到一个句号，满足下面条件之一就不是一个句子
  - 句号后面有一个空格，然后是一个非大写字母␣[^A-Z]
  - 句号后面有任一非空格字符[^␣]
  - 否定前瞻模式：(?!␣[^A-Z]|[^␣])

否定(Negative)前瞻机制

```
>>> s = '''See the U.S.A. today. It's right here,
not a world away. Average temp. is 66.5.'''
>>>
>>> p = re.compile(r'[A-Z].*?[.!?](?! [^A-Z]|[^ ])',
                   re.S | re.M)
>>> lst = p.findall(s)
>>> for i, sentence in enumerate(lst):
        print(i, ':', sentence)


0 : See the U.S.A. today.
1 : It's right here,
not a world away.
2 : Average temp. is 66.5.
```

正则表达式对象支持的方法

- match() - 在字符串开头查找匹配模式的文本
- search() - 在字符串中查找第一次匹配模式的文本
- findall()/finditer() - 从左向右查找所有匹配模式的文本
- split() - 用匹配模式的文本作为分隔符对字符串进行分割
- sub() - 将匹配模式的文本用指定字符串进行替换，返回替换后的字符串
- subn() - 同上，但返回值是一个元组，包含替换后的字符串以及进行替换的次数

利用正则表达式对字符串进行分割

- 字符串的split方法用一个固定的分隔符对字符串进行分割，默认情况下用空白字符（空格，换行，制表）作为分隔符
- 正则表达式对象的split方法用匹配模式的文本作为分隔符对字符串进行分割

```
>>> 'abc5ab5a5'.split('5')
['abc', 'ab', 'a', '']
>>>
>>> p = re.compile(r'\d+')
>>> p.split('abc1ab23a456')
['abc', 'ab', 'a', '']
>>>
>>> q = re.compile(r'(\d+)')
>>> q.split('abc1ab23a456')
['abc', '1', 'ab', '23', 'a', '456', '']
```

字符串的split方法

正则表达式对象的split方法
任意一个数字都作为分隔符

如果模式中采用分组，那么
返回结果还包含匹配的分隔符

利用正则表达式对字符串进行替换

- sub(replacement, string [, count=0])：将string中和模式匹配的文本用replacement替换，最多替换count次
- subn() - 同上，但返回值是一个元组，包含替换后的字符串以及进行替换的次数

```
>>> p = re.compile(r'(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes',
          count = 1)
'colour socks and red shoes'
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

在替换过程中引用分组

- 每个分组都有一个编号k，通过\k可以引用该分组
- sub方法的replacement参数中的\k都将引用匹配的分组

```
>>> msg = 'The the cow jumped over over the moon'
>>> p = re.compile(r'\b(\w+)\s+\1\b', re.I)
>>> p.sub(r'\1', msg)
'The cow jumped over the moon'
```

- 对于命名分组(?P<name>expr)，可以使用\g<name>来引用

```
>>> pat = r'\b(?P<word>\w+)\s+(?P=word)\b'
>>> p = re.compile(pat, re.I)
>>> p.sub(r'\g<word>', msg)
'The cow jumped over the moon'
>>> p.sub(r'\g<1>', msg)
'The cow jumped over the moon'
```

\g<1>等价于\1

replacement参数是函数对象

- sub方法的replacement参数如果是一个函数，那么对于每一个和模式匹配的文本，用匹配对象作为参数调用该函数，并用返回结果去替换和模式匹配的文本

```
>>> def hexrepl(m):
        value = int(m.group())
        return hex(value)

>>> msg = 'Call 119 for emergency and 10000 for CTC'
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, msg)
'Call 0x77 for emergency and 0x2710 for CTC'
```