正则表达式

主要内容

- 正则表达式
- 模式对象和相应方法
- 子模式扩展(拓展的内容)
- 替换和分割

- Regular-Expressions.info - Regex Tutorial, Examples and Reference - Regexp Patterns
- regex101: build, test, and debug regex
- Debuggex: Online visual regex tester. JavaScript, Python, and PCRE.

正则表达式(Regular Expression)

- 字符串已经有：
  - in 判断子字符串是否出现: '213-867-5309' in 'My phone number : 213-867-5309.'
  - find(sub [,index,[stop]]) 和index(sub….)判断子字符串sub出现的位置
  - split(sep, maxsplit): 分割字符串
  - replace(old, new [,count]) 用new替代其中的子串old
  - maketrans(x, y, z) translate() 以单字符为单位进行替代或删除
- 如果想要知道字符串里面是否包括合法的电话号码，以及这些号码是什么?
  - 电话号码满足一定的模式，str只支持**固定的字符串**的查找
  - 引入正则表达式:
    - 可以判断某个**模式(pattern)**是否在字符串中出现 matchobj = re.search(pattern)
    - 可以知道**模式在字符串出现的位置以及匹配的内容** matchobj.span() matchobj.group()
    - 可以根据模式来**分割字符串**，可以根据模式来**替换字符串** re.split(...) re.sub(...)

正则表达式

- 正则表达式(regular expression)：由普通字符和具有特殊含义的元字符(metacharacter)组成的模式(pattern)
  - 最简单的正则表达式仅仅包含普通的字符，比如'for'
- re模块(import re)提供了使用正则表达式进行查找、替代和分割的方法

  re.findall(pattern, string, flags=0)
  - 在字符串string中搜索与模式pattern匹配的内容，接下来继续搜索下一个匹配...，找到的匹配组成一个**列表**返回
  - flags给出了搜索时正则表达式匹配的选项

```
>>> import re
>>> text='The quick brown fox jumped for food. The lazy black dog
jumped for food.'
>>> re.findall('for', text)
['for', 'for']
```

正则表达式

具有特殊含义的主要元字符（meta character) 包括 \ . ^ $ * + ? | { } [ ] ( )

字符集	**[ ]**	表示匹配中括号里面的字符集中的任一字符	[abc] [[abc]
	-	在[]内部使用，描述在某个范围的字符，如果出现在第一个或者最后一个，匹配-本身	[a-c] [a-] [-ab] [-[\]] 匹配 – [或]之一
	^	在[]内部的第一个字符为^时，匹配**不在**后面描述的字符集中的字符	[^a-c] [a^c]
	\d \D \s \S \w \W	预定义字符集，分别匹配数字,非数字,空格,非空格,单词,非单词字符	[\d\s] \shelllo\s
	. (dot)	匹配除换行符以外的任意单个字符。如果re.S，也匹配换行符	^\s*#.*$ 匹配注释行
重复	**{n,m}**	表示前面的字符或子模式重复的次数(n-m次)，其中m可以省略	\d{1,3} \d{1} (fo){1,}
	+	表示前面的字符或子模式重复1次以上 {1,}	\d+
	*****	表示前面的字符或子模式重复0次以上 {0,}	\d*
	?	表示前面的字符或子模式重复0次或1次 {0,1}	\d?
边界	**^ \A**	表示位于字符串的**开始位置**，多行模式下还表示行首。\A仅表示开始位置	^The
	$ \Z	表示位于字符串的结束位置或最后位置为换行符时其之前的位置，多行模式下还表示行尾。\Z仅表示结束位置	food$
	\b \B	匹配单词的开始或者结束位置， \B表示在单词的中间位置	\bfood\b \Boo\b
分组	**( )**	描述子模式，表示其作为一个整体，可应用重复符	(fo){1,3}
选择	**|**	用于选择匹配多个可能的模式（注意不一定是单个字符）中的一个	foo.(org|com|net)
转义	****	如果其后跟的字符为数字或特殊序列，则表示特殊含义。如果其后跟的为非数字和非英文字母，匹配第二个字符。如果其后跟英文字母，但是非特殊序列，则抛出异常	\\或*或\d或\1

正则表达式元字符：转义字符 \

- 正则表达式中，元字符\有特殊的含义
  - \后跟着某些(英文字母)字符，表示**预定义字符集或边界匹配**等， \A \b \B \d \D \s \S \w \W \Z
  - 支持与**字符串类似的标准转义序列**，包括\a \t \n \v \f \r \xhh(十六进制的ASCII码对应的字符) \uXXXX \UXXXX (十六进制的Unicode码对应的字符）等，注意字符串中\b表示Backspace字符，但是在正则表达式中\b表示单词的开始或结束位置
  - \后跟英文字母，且不在上面给出的合法序列中，则会**抛出异常(bad escape)**
  - \后跟**数字N**，表示匹配前面**编号为N的子模式(组)匹配的内容**
  - \后跟着**非数字和非英文字母的字符，表示匹配该字符(前面的\去除)**，比如\\ 匹配反斜杠本身

字符串	如果\e不是str支持的合法转义序列时，**保留**	'\e' 等同于r'\e'
正则表达式	如果**\后面跟着非英文字母和数字**，则**去掉**。如果是英文字母但非合法转义序列，抛出异常	\; 匹配 ; \\ 匹配 \

```
>>> import re
>>> re.findall(r'1+1=2', '1+1=2')
[]
>>> re.findall(r'1\+1=2', '1+1=2')
# re.findall(r'1\+1\=2', '1+1=2')
['1+1=2']
```

\后面跟着不识别的字符时，并不保留\。只有正则表达式以及shell采用这样的设计！！

```
>>> re.compile(r'\kabc’ )
…
re.error: bad escape \k at position 0
```

如何传递正则表达式参数?

```
re.findall(pattern, string, flags=0)
```

- pattern参数为字符串(str)类型。正则表达式使用的转义字符\也是str类型使用的转义字符:
  - 首先Python的字符串字面量定义解释pattern中的转义字符\
  - 其次re模块解释字符串pattern包含的正则表达式中的转义字符\

re.findall('....')

- str字面量定义支持转义字符如\b \t \n等，表示某个特殊的字符
- \后面跟着单双引号以及反斜杠，则表示后面的字符本身
- 不支持的转义序列会保留\，如\s表示两个字符\和s

re模块检查pattern中的内容，如果发现字符\，检查后面的字符，并解释

- 假设要匹配空格，正则表达式为\s
  - 可传递r'\s'
  - 可传递'\\s'，字符串字面量定义中前面\\解释为\，后面为s
  - 不建议：可传递pattern='\s'，字符串字面量定义中，不识别\s，两个字符都保留
- 也果我们希望匹配\s两个字符:
  - 正则表达式必须进行转义 → \\s
  - 包含正则表达式的字符串应该这样定义:

'\\\\s'	进行转义，在每个**前面再加上**
r'\\s'	采用**原始字符串**
re.escape(r'\s')	采用re模块的escape方法

```
s = r't\s t t\\s'
print(re.findall('\s',s))
print(re.findall('\\s',s))
# 正则表达式为\s，匹配空格类字符，输出[' ', ' ']
print(re.findall(r'\\s',s))
print(re.findall('\\\\s',s))
print(re.findall(re.escape('\\s'),s))
#正则表达式为\\s，匹配\s，输出['\\s', '\\s']
```

re.escape(pattern)

- 返回字符串，该字符串对于pattern中那些具有特殊含义的字符，会在其前面插入一个反斜杠，这样其作为正则表达式模式时**匹配pattern本身**
- 用于动态构建一个正好匹配pattern本身的正则表达式，避免pattern里面的字符被re解释为特殊的含义

```
>>> print(re.escape(r'\s'))
\\s
>>> print(re.escape(r'http://www.python.org'))
http://www\.python\.org
```

```
def re_escape2():
    text = """regexp = re.compile('((jump)ed for) (food)')
    text='The quick brown fox jumped for food. The lazy black dog jumped for food.'
    print(regexp.findall(text))
    """
    pattern = input("请输入要匹配的内容:")
    esc_pattern = re.escape(pattern)
    print('正则表达式为', esc_pattern)
    matches = re.findall(esc_pattern, text)
    print(matches)
```

re_usage.py

正则表达式元字符：字符集

```
>>> re.findall(r'[\u4E00-\u9FA5]', 'python程序设计')
['程', '序', '设', '计']
```

元字符	说明
[xyz]	匹配位于[]中的任意一个字符。注意除了第一个字符为^表示否定，-在中间出现表示范围，还有\表示转义外，其他元字符失去了其含义，表示字符本身
-	用在[]之内用来表示范围，如果为第一个或者最后一个字符就表示-本身
^	用在[]之内用来表示否定，必须是第一个字符才起作用

字符集：由一对方括号[ ]括起来的字符集合，定义方式如下：

- [xyz]：枚举字符集，匹配括号中字符集中的任意一个字符
  - '[pjc]ython'可以匹配'python'、'jython'、'cython'
- [a-z]：指定范围的字符，匹配其中任意一个字符。
  - '[a-zA-Z0-9]'可以匹配一个任意大小写英文字母或数字
  - 连字符放在最前面或最后面，表示连字符本身，如'[0-]'
- [^xyz]:否定枚举字符集，匹配不在后面给出的字符集中的任意字符
  - ^只有出现在第一个位置才表示否定
  - '[^a-z]' 匹配非小写英文字母的字符，但'[a-z^]' 匹配小写英文字母或者^

```
>>> re.findall(r'[]^[-]', '[^abc0-9]')
['[', '^', '-', ']']
>>> re.findall(r'1\+1=2', '1+1=2')
['1+1=2']
>>> re.findall(r'1[+]1=2', '1+1=2')
['1+1=2']
>>> re.findall(r'1[\+]1=2', '1+1=2')
['1+1=2']
```

[+]等价于\+

正则表达式元字符：预定义字符集

预定义字符集：系统定义了若干预定义的比较常用的字符集

- **缺省的匹配选项隐含了re.U**，表示采用unicode意义上的数字、字母的定义。选项 re.A表示采用ASCII字符集中的数字、字母的定义
- 预定义字符集可以在[ ]内使用，也可以在[ ]外使用

```
>>> re.findall(r'.', 'ab\ncd')
['a', 'b', 'c', 'd']
>>> re.findall(r'.', 'ab\ncd', re.S)
['a', 'b', '\n', 'c', 'd']
```

元字符	说明
\d	匹配数字类字符，对于ASCII字符集(flags=re.A)，相当于[0-9]
\D	与\d含义相反，匹配非数字字符字，相当于[^\d]
\w	匹配能组成单词的字符，对于ASCII字符集，相当于[a-zA-Z0-9_]
\W	与\w含义相反,匹配非单词字符，相当于[^\w]
\s	匹配空格类字符，对于ASCII字符集，相当于[\t\n\r\f\v]
\S	与\s含义相反，匹配非空格类字符，相当于[^\s]
.	匹配除换行符以外的任意单个字符，flags包含re.S(re.DOTALL)时也匹配换行符

```
re.findall(r'[A-Z]\w\w', "You meet Hello T47")
# 匹配英文大写字母开始，接下来紧跟2个单词字符,输出['You', 'Hel', 'T47']
re.findall(r'0[Xx][\da-fA-F][\da-fA-F]', '0x10 0xff')
# 匹配2位十六进制数字，输出['0x10', '0xff']
```

正则表达式元字符：预定义字符集

预定义字符集：系统定义了若干预定义的比较常用的字符集

- 缺省的匹配选项隐含了re.U，表示采用unicode意义上的数字、字母的定义。选项 re.A表示采用ASCII字符集中的数字、字母的定义
- 预定义字符集可以在[ ]内使用，也可以在[ ]外使用

```
>>> re.findall(r'.', 'ab\ncd')
['a', 'b', 'c', 'd']
>>> re.findall(r'.', 'ab\ncd', re.S)
['a', 'b', '\n', 'c', 'd']
```

元字符	说明
\d	匹配数字类字符，对于ASCII字符集(flags=re.A)，相当于[0-9]
\D	与\d含义相反，匹配非数字字符字，相当于[^\d]
\w	匹配能组成单词的字符，对于ASCII字符集，相当于[a-zA-Z0-9_]
\W	与\w含义相反,匹配非单词字符，相当于[^\w]
\s	匹配空格类字符，对于ASCII字符集，相当于[\t\n\r\f\v]
\S	与\s含义相反，匹配非空格类字符，相当于[^\s]
.	匹配除换行符以外的任意单个字符，flags包含re.S(re.DOTALL)时也匹配换行符

```
re.findall(r'[A-Z]\w\w', "You meet Hello T47")
# 匹配英文大写字母开始，接下来紧跟2个单词字符,输出['You', 'Hel', 'T47']
re.findall(r'0[Xx][\da-fA-F][\da-fA-F]', '0x10 0xff')
# 匹配2位十六进制数字，输出['0x10', '0xff']
```

正则表达式元字符：边界匹配符

0宽度边界匹配符：字符串匹配往往涉及从某个位置开始匹配，例如行的开头或结尾、单词边界等，边界匹配符用于**匹配字符串的位置（匹配结果为空字符串）**，**不会消耗**模式中的字符

元字符	说明
^	表示后面匹配的模式出现在字符串的开头，在多行模式下（flags包括re.M）匹配每行的行首，即后面的模式出现在行首。如："^a"匹配"abc"中的"a",不匹配"bat"中的"a"
$	表示匹配的模式位于最后一行行尾(即字符串结尾，或字符串最后位置为换行符时出现在最后那个换行符之前的位置。在多行模式下匹配每行的行尾或字符串结尾 。如："c$"匹配"abc"中的"c",不匹配"acb"中的"c"
\A	匹配字符串开始的位置
\Z	匹配字符串结束的位置
\b	匹配单词头或单词尾。位于单词头指前面为非单词类字符或字符串开始，当前位置为单词字符。单词尾表示当前为单词字符，后面为非单词字符或字符串结尾。 如：r"\bfoo\b"匹配"foo","foo." "(foo)" "bar foo baz"但不匹配"foobar" "foo3" 注意:str中'\b'表示退格字符，所以正则表达式参数或者前面添加转义，或者使用原始字符串r，即"\\bfoo\\b"　或 r"\bfoo\b"
\B	与\b含义相反，表示不在单词的边界位置。如位于单词中间。也可是前面非单词，后面也非单词等 如：'py\B' 匹配 "python" "py3",但不匹配"happy" "py!"　'\B@\w\w\w' 匹配 '@foo'

```
>>> re.findall('^#', '# 1\n# 2\n# 3')
['#']
>>> re.findall('^#', '# 1\n# 2\n# 3', re.M)
['#', '#', '#']
```

正则表达式元字符：重复限定符

X可为单个字符或通过小括号包含的子模式（作为一个整体）

指定重复的次数，默认采用贪婪匹配算法

元字符	说 明
X{m,n}	X重复m到n次,m和n可省略1个 如"o{1,3}" 匹配 "fooooood"中的前3个"o"和后3个"o"
X{m,}	X至少重复m次 如："o{2,} " 匹配 "fooooood"中的所有的"o",不匹配"bob"中的"o"
X{,n}	X重复0到n次。如 0{,2}1匹配 1， 01， 001
X{m}	X重复m次 如："[0-9]{3}" 匹配 "000" ~ "999"
X+	X重复1次或多次，等价于X{1,} 如："zo+" 匹配 "zo" ,"zoo",但不匹配 "z"
X*	X重复0次或多次，等价于X{0,} 如："zo*" 匹配 "zo" ,"zoo", "z"
X?	可选，即X重复0次或1次，等价于X{0,1} 如："colou?r" 匹配"color", "colour"

- **贪婪匹配算法**是指正则表达式引擎**尽可能多 (leftmost or largest)**匹配要重复的前导字符或子模式，只有当这种重复引起整个模式匹配失败的情况下，引擎才会进行回溯（尝试稍短的匹配）

```
>>>re.findall('<.+>','<book><title>Python</title><author>Dong</author></book>')
['<book><title>Python</title><author>Dong</author></book>']
```

期待： ['<book>', '<title>', '</title>', '<author>', '</author>', '</book>']

重复限定符：懒惰匹配算法

如果在重复限定符后面加后缀"?"，表示使用**懒惰匹配算法**

- 懒惰匹配算法是指正则表达式引擎**尽可能少**地进行重复匹配，只有当这种重复引起整个正则表达式**匹配失败**的情况下，引擎会进行回溯(匹配更多的字符)

符号	说明
*?	重复任意次，但尽可能少重复
+?	重复1次或更多次，但尽可能少重复
??	重复0次或1次，但尽可能少重复
{m,n}?	重复m到n次，但尽可能少重复
{m,}?	重复m次以上，但尽可能少重复
{,n}?	重复0到n次，但尽可能少重复

.+?>	尽可能少地匹配一个以上字符，后面为>
[^>]+>	尽可能多地匹配一个以上非>字符，后面为>

```
>>> import re
>>>re.findall('<.+?>','<book><title>Python</title><author>Dong</author></book>')
['<book>', '<title>', '</title>', '<author>', '</author>', '</book>']
>>> re.findall('<[^>]+>','<book><title>Python</title><author>Dong</author></book>')
['<book>', '<title>', '</title>', '<author>', '</author>', '</book>']
```

一些正则表达式的例子

```
>>> import re
>>> text = 'The quick brown fox jumped for food.'
>>> pattern = r'\bf\w+\b'
>>> re.findall(pattern, text)
```

r'\b\w+\b'	匹配所有单词（最后的\b可以省略）	['The', 'quick', 'brown', 'fox', 'jumped', 'for', 'food']
r'\bf\w+\b'	匹配f开头的完整单词（最后的\b可省略）	['fox', 'for', 'food']
r'\b\w+d\b'	匹配d结尾的至少2个字符的单词	['jumped', 'food']
r'\b\w{3}\b'	长度为3的单词	['The', 'fox', 'for']
r'\Bo\w+\b'	出现在非单词开头的o之后的单词剩余部分（最后的\b可省略）	['own', 'ox', 'or', 'ood']

正则表达式元字符：分组符(group)

要匹配的模式中，可以将其分成多个部分，其中的几个特定的部分加上括号，这些加括号的部分'(pattern)'称为分组(group)或子模式

- ()包含的子模式内的内容作为一个整体出现，可以应用前面介绍的重复限定符，表示**子模式重复多次。**比如(pattern){2}表示前面的内容与pattern匹配，之后的内容也与pattern匹配
- 在获得模式所匹配的内容的同时也可以获得各个子模式匹配的内容
- 支持**分组引用(backreference)，**匹配前面相应子模式所匹配的内容

2020/5/29

- 每个group有一个编号: 按照从左到右的**左括号的顺序**从**1开始**进行编号。**编号0**表示整个模式
- group可以嵌套，如'((jump)ed for) (food)'

([0-9]{4})/([0-9]{1,2})/([0-9]{1,2})

年份	月份	第几日
4位数字	1到2位数字	1到2位数字

'(jump)ed for (food)' 匹配字符串'The quick brown fox jumped for food'

group 0: "jumped for food"
group 1: "jump'
group 2: "food"

如果正则表达式为'((jump)ed for) (food)'

group 2
group 1
group 3

group 0: "jumped for food"
group 1: "jumped for"
group 2: "jump'
group 3: "food"

正则表达式元字符：分组符(group)

`re.findall(pattern, string, flags=0)`

- 在字符串string中搜索与模式pattern匹配的内容，接下来继续搜索下一个匹配...，找到的匹配组成一个**列表**返回
- flags给出了搜索时正则表达式匹配的选项
- 如果模式中没有**组(子模式)**，则列表的每个元素为所有与模式匹配的内容
- 如果模式中有**组(子模式)**，则匹配结果中仅包括与子模式匹配的字符串，整个模式匹配的内容(group 0)并没有包含在内。如果有**多个组(子模式)**，则每次匹配的结果以tuple形式组织，即[**(group 1,group2...,groupN)**, (.....), ..., (.....)]
- 如果最后输出既想要子模式也想要模式匹配的内容，则模式定义中应该用group包含所有的内容

```
def re_findall():
    text='The quick brown fox jumped for food. The lazy black dog jumped for food.'
    print(re.findall('jumped for food', text)) #没有group，包含匹配的模式
    print(re.findall('jumped for (food)', text))
    print(re.findall('(((jump)ed for) (food))', text)) #4个group，group 1包含所有内容
```

['jumped for food', 'jumped for food']
['food', 'food']
[('jumped for food', 'jumped for', 'jump', 'food'), ('jumped for food', 'jumped for', 'jump', 'food')]

re_usage.py

正则表达式元字符：分组符(group)

```
>>> re.findall(r'(\s)*(\w+)', 'hello world')
[('', 'hello'), (' ', 'world')]
```

- 分组符"()"：将要匹配的模式进一步分组，也称为子模式
  - 在获得匹配的模式的同时也可以获得各个匹配的子模式
  - ()包含的子模式内的内容作为一个整体出现,可以应用重复限定符，表示**子模式重复多次**
  - 子模式重复多次时，在最后输出的那个分组中的内容是其**最后一次重复**所匹配的内容
  - 子模式重复允许0次，如果**匹配的正好是0次**，则findall中输出的那个分组内容为**空字符串**
  - 子模式重复限定符之后也允许附加一个?，表示采用懒惰匹配 '(red)+'可匹配'red' 'redredred'等

IP地址：202.120.225.10
4个整数以dot分隔，每个整数包括1到3个十进制数字

```
>>> text = '202.120.225.10  8.8.8.8 127.0.0.1'
>>> re.findall(r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}', text)
['202.120.225.10', '8.8.8.8', '127.0.0.1']
>>> re.findall(r'(\d{1,3}\.){3}\d{1,3}', text)
['225.', '8.', '0.']
>>> re.findall(r'((\d{1,3}\.){3}\d{1,3})', text)
[('202.120.225.10', '225.'), ('8.8.8.8', '8.'), ('127.0.0.1',
'0.')]
```

正则表达式	说明
(pattern)?	允许模式重复0次或1次
(pattern)*	允许模式重复0次或多次
(pattern)+	允许模式重复1次或多次
(pattern){m,n}	允许模式重复m~n次
(pattern){m,}	允许模式重复至少m次
(pattern){m}	允许模式正好重复m次

- 一旦引入分组，则findall返回的列表中仅仅包含分组所匹配的内容，因此如果想要包含全部匹配的模式，需要使用分组符将所有要匹配的内容包含起来

正则表达式元字符：选择符

pattern1|pattern2|pattern3

- 选择符"|"，用于选择匹配多个可能的子模式（注意不一定是单个字符）中的一个
- 选择符"|"的优先级最低（最后计算），如需要，可使用( )来限制选择符的作用范围
  (pattern1|pattern2|pattern3)

Python语句	匹配结果
re.findall('red|green|blue', 'pink rred ,red,green, ggreen and blue')	['red', 'red', 'green', 'green', 'blue']
re.findall(r'\bred|green|blue\b', 'pink rred ,red, ggreen and blue')	['red', 'green', 'blue']
re.findall(r'\b(red|green|blue)\b', 'pink rred ,red, ggreen and blue')	['red', 'blue']

```
>>> re.findall(r'(\b(bat|cat)\w+)', 'batch bathroom catch catbird')
[('batch', 'bat'), ('bathroom', 'bat'), ('catch', 'cat'), ('catbird', 'cat')]
>>> re.findall(r'(\b(b|c)at\w+)', 'batch bathroom catch catbird')
[('batch', 'b'), ('bathroom', 'b'), ('catch', 'c'), ('catbird', 'c')]
>>> re.findall(r'\b[bc]at\w+', 'batch bathroom catch catbird')
['batch', 'bathroom', 'catch', 'catbird']
```

主要内容

- 正则表达式
- **模式对象和相应方法**
- 子模式扩展(拓展的内容)
- 替换和分割

模式对象：匹配选项

- Pattern和string都是str对象，选项缺省为re.U
- Pattern和string都是bytes对象，选项缺省为re.L

`re.findall(pattern, string, flags)` 等价 `re.compile(pattern, flags).findall(string)`

- 首先调用re模块的**compile方法**将其编译成一个正则表达式(或模式)对象(类型为re.Pattern)
- 然后调用该模式对象的findall方法
- flags可选，描述匹配时的所采用的选项
- 匹配选项可以采用短格式(如re.I)，也可采用长格式(如re.IGNORECASE)
- 多个选项可以通过运算符 | (按位或) 组合，比如 re.I | re.U

```
>>> import re
>>> regexp = re.compile(r'((jump)ed for) (food)')
>>> regexp
re.compile(r'((jump)ed for) (food)', re.UNICODE)
>>> regexp = re.compile(r'((jump)ed for) (food)', re.I)
>>> regexp
re.compile(r'((jump)ed for) (food)',
re.IGNORECASE|re.UNICODE)
>>> re.compile(r'((jump)ed for) (food)', re.I |
re.DOTALL)
re.compile(r'((jump)ed for) (food)', re.IGNORECASE |
re.DOTALL | re.UNICODE)
```

匹配选项(flags)	说明
re.I re.IGNORECASE	忽略大小写
re.L re.LOCALE 忽略这个选项	\w \W \b \B \s \S与本地字符集有关，re模块用于bytes对象时使用
re.U re.UNICODE	\w \W \b \B \s \S的定义基于Unicode，缺省选项
re.A re.ASCII	\w \W \b \B \s \S的定义基于ASCII
re.S re.DOTALL	元字符.(dot) 也匹配换行符
re.X re.VERBOSE	忽略模式中的空格（这样可以写成多行，也可添加空格），并可以使用#注释，提高可读性
re.M re.MULTILINE	多行匹配模式，^ $可以匹配每行的开始和结尾

模式对象：匹配选项

什么时候需要使用多行匹配模式?

- 调用file.read()得到的是**多行字符串**，且匹配模式时希望通过^ $等来**逐行匹配**时
- 如果file.readline()得到的是一行的内容，则不需要用多行模式

```
def re_flag():
    text = '''# -*- coding: utf-8 -*-
import math
from os import *
print('test')
'''
    match = re.search(r'^.+\*$', text)
    if match:
        print(match.re)
        print(match.group())

    match = re.search(r'^.+\*$', text, re.M)
    if match:
        print(match.re)
        print(match.group())
```

re_group.py

忽略模式中的空格（这样可以写成多行，也可添加空格），并可以使用#注释，提高可读性

```
pattern = r"""
    (?P<first_three>[\d]{3}) # The first three digits…
     -                       # A literal hyphen…
    (?P<last_four>[\d]{4})   # The last four digits…
"""
m = re.search(pattern, '867-5309', re.VERBOSE)
print(m)
```

re模块的方法

`re.findall(pattern, string, flags)` 等价 `re.compile(pattern, flags).search(string)`

- re模块的搜索类的方法**无法传递pos/endpos参数**，即搜索的范围只能为整个字符串

模式对象的方法	re模块的方法
编译正则表达式，返回模式对象	`re.compile(pattern, flags=0)`
`regexp.search(string[, pos[, endpos]])`	`re.search(pattern, string, flags=0)`
`regexp.match(string[, pos[, endpos]])`	`re.match(pattern, string, flags=0)`
`regexp.finditer(string[, pos[, endpos]])`	`re.finditer(pattern, string, flags=0)`
`regexp.findall(string[, pos[, endpos]])`	`re.findall(pattern, string, flags=0)`
`regexp.sub(repl,string[,count=0])`	`re.sub(pattern, repl, string, count=0, flags=0)`
`regexp.subn(repl,string[,count=0])`	`re.subn(pattern, repl, string, count=0, flags=0)`
`regexp.split(string[, maxsplit = 0])`	`re.split(pattern, string, maxsplit=0, flags=0)`
返回字符串，该字符串对于pattern中有特殊含义的正则表达式元字符进行转义	`re.escape(pattern)` 比如`print(re.escape(r'\\'))  # 输出 \\\\`

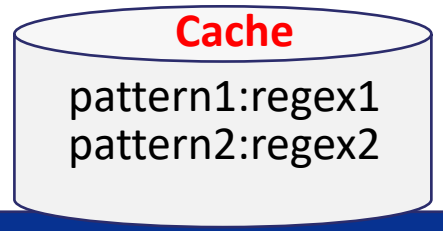
re模块的方法

```
regexp = re.compile('((jump)ed for) (food)')
match = regexp.search('The quick brown fox jumped for food')
print(match.groups())

match = re.search('((jump)ed for) (food)', 'The quick brown fox jumped for food')
print(match.groups())
print(match.re.pattern)  # 正则表达式对象
```

到底采用哪种方法呢?

- 在调用re模块的findall/search…/sub/split等方法时，其内部实现也是首先调用compile创建一个模式对象，然后调用模式对象的方法，而且其内部对于**模式对象进行了缓存**，下次再传递相同的正则表达式时会重用前面compile的模式对象
- 两种方法都可以使用。如果要采用同一个正则表达式调用多次时，可以采用首先compile然后调用模式对象的方法，可节省多次函数调用（不需要每次调用compile…）



search方法

regexp.search(string[, pos[, endpos]])

re.search(pattern, string, flags=0)

- 在字符串string或其指定范围string[pos:endpos)搜索第一个与模式匹配的内容。若找到，返回一个描述匹配信息的**Match对象**，否则返回None

Match对象方法	说明
group([group1,…])	返回匹配的1个或多个捕获子模式所对应的内容。如果不传递参数，表示group 0，即模式对应的内容。 如果传递多个参数，则返回元组，每个元素为相应子模式匹配的内容。比如group(1,2)返回group 1和group 2的匹配内容组成的元组
start(group=0)	返回匹配的相应子模式在原字符串的起始位置的下标，不传递参数时，表示组0，即整个模式在原字符串的起始位置的下标
end(group=0)	返回指定模式或子模式匹配的内容的结束位置(匹配内容不包括该下标，即下一个位置)
span(group=0)	返回用于描述匹配的模式或子模式内容在原字符串的开始和结束位置的下标，二元元组(m.start():m.end())
groups()	返回一个包含匹配的所有捕获的子模式(1,2,…)的内容的元组，注意不包括group 0
groupdict()	返回包含匹配的所有捕获的**命名子模式**内容的字典
属性re	返回进行匹配时采用的模式对象regexp

search方法示例： regexp.search(string[, pos[, endpos]])

```
>>> import re
>>> regexp = re.compile('((jump)ed for) (food)')
>>> text='The quick brown fox jumped for food.
The lazy black dog jumped for food.'
>>> match = regexp.search(text)
>>> match
<_sre.SRE_Match object; span=(20, 35),
match='jumped for food'>
>>> match.group()
'jumped for food'
>>> match.group(1, 2)
('jumped for', 'jump')
>>> match.groups()
('jumped for', 'jump', 'food')
>>> match.start()
20
>>> match.end()
35
>>> match.span()
(20, 35)
```

```
>>> match.start(1)
20
>>> match.end(1)
30
>>> match.span(1)
(20, 30)
>>> match[0], match[1], match[2]
('jumped for food', 'jumped for', 'jump')
>>> match2 = regexp.search(text, 35)
>>> match2
<_sre.SRE_Match object; span=(56, 71),
match='jumped for food'>
>>> match3 = regexp.search(text, 71)
>>> print(match3)
None
```

python3.6允许通过下标访问group

```
pos = 0     # 从头开始搜索（pos=0）
while True:
    match = regexp.search(text, pos)
    if not match:
        break
    pos = match.end()
```

search方法示例

1. 编译模式对象： regexp=re.compile(pattern)
2. 下次搜索开始的位置pos = 0
3. 重复直到调用 regexp.search(text,pos)返回match对象为空（即没有更多的匹配）
   - 从match对象中获得找到的匹配以及相应位置
   - 更新pos = match.end()

re_usage.py

```
import re
regexp = re.compile('((jump)ed for) (food)')
text='The quick brown fox jumped for food. The lazy black dog jumped for food.'
pos = count = 0     # 从头开始搜索（pos=0）
while True:
    match = regexp.search(text, pos)     #从上次匹配的位置之后继续搜索模式
    if not match:      # 找不到，退出循环
        break
    count += 1    # 第几次找到？
    print('{} found "{}" at [{},{}]'.format(count, match.group(), *match.span()))
    for k, v in enumerate(match.groups(), 1):
        print('group %d: %s, span %s' % (k, v, match.span(k)))
    pos = match.end()     # 记录上次匹配的内容之后所对应的位置，即下一次搜索开始的位置
```

finditer方法

- search方法只能找到第一个与模式匹配的match对象
- finditer(string[, pos[, endpos]])：返回一个迭代器，该迭代器会每次返回匹配的下一个match对象

re_usage.py

```
def re_finditer():
    regexp = re.compile('((jump)ed for) (food)')
    text = 'The quick brown fox jumped for food. The lazy black dog jumped for food.'
    matches_iter = regexp.finditer(text)
    for count, match in enumerate(matches_iter, 1):
        print('{} found "{}" at [{},{}]'.format(count, match.group(), *match.span()))
        for k, v in enumerate(match.groups(), 1):
            print('  group %d: %s, span %s' % (k, v, match.span(k)))
```

match和fullmatch方法

- match(string[, pos[, endpos]])与search类似，只是要求模式(pattern)出现在**字符串开头**或指定范围的开头，等价于正则表达式模式为**^pattern**
- fullmatch(string[, pos[, endpos]]) 要求**模式完整匹配**字符串，即模式的最前面为字符串开头或指定范围的开头，最后面为字符串尾部或指定范围的尾部，等价于**^pattern$**

```
>>> import re
>>> text='The quick brown fox jumped for food. The lazy black dog jumped for food.'
>>> regexp = re.compile('((jump)ed for) (food)')
>>> match = regexp.match(text)    # 正则表达式相当于 '^((jump)ed for) (food)'
>>> print(match)
None
>>> regexp2 = re.compile('the', re.I)
>>> match = regexp2.match(text)
>>> match
<_sre.SRE_Match object; span=(0, 3), match='The'>
>>> regexp2.fullmatch('The')    #正则表达式相当于'^the$'
<_sre.SRE_Match object; span=(0, 3), match='The'>
```

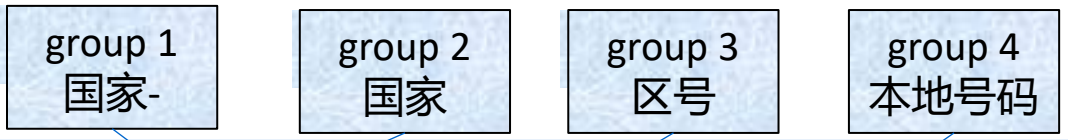
主要内容

- 正则表达式
- 模式对象和相应方法
- **子模式扩展（拓展的内容）**
- 替换和分割

分组(子模式)

电话号码：**国家**-区号-本地号码，其中国家-为可选的。即国家-区号-本地号码 或区号-本地号码

国家为2位，区号为2~3位, 本地号码为7~8位数字



re_group.py

```
import re
pattern1 = r'((\d{2})-)?(\d{2,3})-(\d{7,8})'
text1 = 'Tel: 86-21-65642222,'
text2 = 'Tel: 21-65642222,'
print('-' * 30, 'pattern=%s' % pattern1, '-' * 30)
for text in (text1, text2):
    print('-' * 40)
    match = re.search(pattern1, text)
    print(match)
    print(match.groups())
    print('match: %s, span %s' % (match.group(), match.span()))
    for k, v in enumerate(match.groups(), 1):
        print('group %d: %s, span %s' % (k, v, match.span(k)))
```

```
------------------------------
<_sre.SRE_Match object; span=(5, 19),
match='86-21-65642222'>
('86-', '86', '21', '65642222')
match: 86-21-65642222, span (5, 19)
group 1: 86-, span (5, 8)
group 2: 86, span (5, 7)
group 3: 21, span (8, 10)
group 4: 65642222, span (11, 19)
------------------------------
<_sre.SRE_Match object; span=(5, 16),
match='21-65642222'>
(None, None, '21', '65642222')
match: 21-65642222, span (5, 16)
group 1: None, span (-1, -1)
group 2: None, span (-1, -1)
group 3: 21, span (5, 7)
group 4: 65642222, span (8, 16)
```

matchobj.groups()	返回一个元组，每个元素为各个子模式匹配的内容

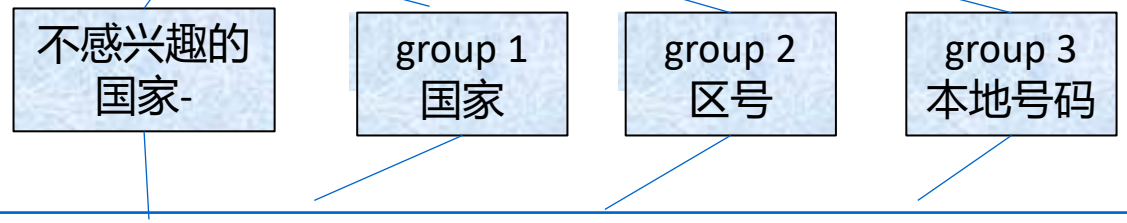
输出的结果中group 1并不需要输出，即不感兴趣

group的**重复次数为0**时，**该group匹配的结果为None**。通过**findall**调用时匹配的结果为**空字符串**

分组(子模式):不捕获分组

- 有的时候可能要进行分组，但用户并不关心某个子模式具体匹配的内容是什么
- 可以通过在**(后面添加?:** 表示对该分组匹配的内容不感兴趣，**不捕获**，即(?:pattern)
- 不感兴趣的分组也**不占用分组编号**

```
pattern1 = r'((\d{2})-)?(\d{2,3})-(\d{7,8})'
```



re_group.py

```
pattern2 = r'(?:(\d{2})-)?(\d{2,3})-(\d{7,8})'
text1 = 'Tel: 86-21-65642222,'
text2 = 'Tel: 21-65642222,'
print('-' * 30, 'pattern=%s' % pattern2, '-' * 30)
for text in (text1, text2):
    match = re.search(pattern2, text)
    print(match)
    print(match.groups())
    print('match: %s, span %s' % (match.group(), match.span()))
    for k, v in enumerate(match.groups(), 1):
        print('group %d: %s, span %s' % (k, v, match.span(k)))
```

```
<_sre.SRE_Match object; span=(5, 19),
match='86-21-65642222'>
('86', '21', '65642222')
match: 86-21-65642222, span (5, 19)
group 1: 86, span (5, 7)
group 2: 21, span (8, 10)
group 3: 65642222, span (11, 19)
<_sre.SRE_Match object; span=(5, 16),
match='21-65642222'>
(None, '21', '65642222')
match: 21-65642222, span (5, 16)
group 1: None, span (-1, -1)
group 2: 21, span (5, 7)
group 3: 65642222, span (8, 16)
```

分组(子模式):命名

- (?:P<name>…)　给分组**命名为name**，但**不捕获该分组**。可通过group(name)来访问该子模式匹配的内容

可以给**分组(子模式)命名**，方便引用，而且更改正则表达式也不会有影响

(?P<name>…)	**给分组命名为name**，该分组**仍然有编号**
matchobj.groups()	返回一个元组，每个元素为各个捕获子模式匹配的内容
matchobj.groupdict()	返回一个字典，其key为那些命名子模式的名字，值为对应子模式匹配的内容
matchobj.group/span/start/end	match对象的那些方法中传递的参数除了原来的分组编号外，还可以**传递分组名(字符串)**

```
pattern3 = r'(?:(?P<country>\d{2})-)?(?P<city>\d{2,3})-(?P<phone>\d{7,8})'
text1 = 'Tel: 86-21-65642222,'
text2 = 'Tel: 21-65642222,'
for text in (text1, text2):
    match = re.search(pattern3, text)
    print(match)
    print(match.groups())
    print('match: %s, span %s' % (match.group(), match.span()))
    for k, v in enumerate(match.groups(), 1):
        print('group %d: %s, span %s' % (k, v, match.span(k)))

    print(match.groupdict())
```

re_group.py

```
<_sre.SRE_Match object; span=(5, 19), match='86-21-65642222'>
('86', '21', '65642222')
match: 86-21-65642222, span (5, 19)
group 1: 86, span (5, 7)
group 2: 21, span (8, 10)
group 3: 65642222, span (11, 19)
{'country': '86', 'city': '21', 'phone': '65642222'}
<_sre.SRE_Match object; span=(5, 16), match='21-65642222'>
(None, '21', '65642222')
match: 21-65642222, span (5, 16)
group 1: None, span (-1, -1)
group 2: 21, span (5, 7)
group 3: 65642222, span (8, 16)
{'country': None, 'city': '21', 'phone': '65642222'}
```

分组(子模式):分组引用

- **分组引用(backreference)**：表示要匹配前面相应子模式所匹配的内容

\N	表示匹配的内容为前面编号为N的分组匹配的内容，注意N不能是后面的分组编号，也不能为0
(?P=name)	表示引用前面组名为name的命名分组匹配的内容

`pattern = r'(\b\w+)\s+\1'`
`match = re.search(pattern,'Paris in the the spring')`
`print('group 0:(%s) group 1:(%s)' % (match.group(), match.group(1)))`
	匹配两个连续出现的单词，之间以多个空格隔开，输出:
	group 0:(the the) group 1:(the)

`pattern = r'((\b\w+)\s+\2)'` `print(re.findall(pattern,'Paris in the the spring'))`	注意findall中使用分组符时，不会返回组0，而是仅仅子模式。在最外层添加了分组符 和上面的代码一样，输出: [('the the', 'the')]

`pattern = r'(?P<word>\b\w+)\s+(?P=word)'`
`match = re.search(pattern,'Paris in the the spring')`
`print('group word:', match.group('word'))`
`print(match.group())`
	采用命名组来实现，匹配两个连续出现的单词

分组(子模式):肯定/否定前瞻

- 在匹配时要检查**当前位置之后是否满足或者不满足某个模式**。在进行前瞻性的检查时并不会消耗任何字符，即**零宽度匹配**
- **不引入捕获组**(capturing group)：不会被捕获，不会占用分组编号

(?=pattern)	零宽度肯定前瞻断言(zero-width positive lookahead assertion)，检查当前位置之后的内容，要求其满足模式pattern，在检查时采用零宽度匹配
(?!pattern)	零宽度否定前瞻断言(zero-width negative lookahead assertion)，检查当前位置之后的内容，要求其与模式pattern不匹配，在检查时采用零宽度匹配

```
# 匹配那些后面跟着逗号的单词
>>> text = 'They were three: Felix, Victor, and Carlos.’
>>> re.findall(r'\w+,', text)
['Felix,', 'Victor,']
>>> pattern = re.compile(r'\w+(?=,)')
>>> pattern.findall(text)
['Felix', 'Victor']
# 匹配那些后面不是跟着空格或.的单词
>>> re.findall(r'\w+\b(?![.\s])', text)
['three', 'Felix', 'Victor']
```

分组(子模式):肯定/否定后顾

- 在匹配时要检查**当前位置之前是否满足或者不满足某个模式**。在进行后顾检查时并不会消耗任何字符，即**零宽度匹配。**python实现限制要求**模式的长度为固定长度**，意味着那些重复次数不固定的重复限定符无法使用，选择符中各个子模式长度也要求一样。**不引入捕获组**

(?<=pattern)	零宽度肯定后顾断言(zero-width positive lookbehind assertion)，检查当前位置之前的内容，要求其满足模式pattern，在检查时采用零宽度匹配
(?<!pattern)	零宽度否定前瞻断言(zero-width negative lookbehind assertion)，检查当前位置之前的内容，要求其与模式pattern不匹配，在检查时采用零宽度匹配

```
>>> text = 'I would rather go out with John Lee than with John Smith or John Baker'
# 匹配前一个单词为John的Smith
>>> re.findall(r'(?<=John\s)Smith', text)
['‘Smith']
# lookbehind要求之前的模式为固定长度
>>> re.findall(r'(?<=(John|Jonathan)\s)Baker', text)
...
re.error: look-behind requires fixed-width pattern
# 匹配前一个单词不是with的John
>>> re.findall(r'(?<!with\s)John', text)
['John']
```

分组(子模式)：匹配选项

- 通过参数传递匹配选项，re.compile(pattern, flags) re.findall(pattern, text, flags)
- 在正则表达式中通过(?flags)等设置匹配选项，**不引入捕获组**

  flags为iLmsux中的一个或者多个字符，分别对应re.IGNORECASE re.LOCALE re.MULTILINE re.DOTALL re.UNICODE re.VERBOSE

(?flags)	必须出现在整个正则表达式的最开始位置，给出了匹配时采用的全局选项。
(?flags:pattern)	flags描述的匹配选项仅仅用于该分组的pattern
(?-flags:pattern)	该分组的pattern匹配时去掉flags描述的匹配选项
(?flags1-flags2:pattern)	该分组的pattern匹配时包含flags1但是不包含flags2描述的匹配选项

```
>>> text = 'Cat SCatTeR CATER cAts CatCh'
>>> re.findall(r'\bCat[a-z]*\b', text) # 以Cat开始，后面为小写字母的单词
['Cat']
>>> re.findall(r'\bCat[a-z]*\b', text, re.I) # 全局大小写无关
['Cat', 'CATER', 'cAts', 'CatCh']
>>> re.findall(r'(?i)\bCat[a-z]*\b', text) # 全局大小写无关
['Cat', 'CATER', 'cAts', 'CatCh']
>>> re.findall(r'\b(?i:Cat)[a-z]*\b', text) # Cat大小写无关
['Cat', 'cAts']
>>> re.findall(r'\b(?-i:Cat)[a-z]*\b', text, re.I) # 全局大小写无关,Cat大小写相关
['Cat', 'CatCh']
```

分组(子模式)扩展语法总结

语法	说明
(?:…)	匹配但不捕获（即对该组不感兴趣）
(?P<groupname>)	为子模式命名，方便对于该组的引用，该分组仍然有编号(捕获组)
(?P=groupname)	在此之前的命名为groupname的子模式匹配的内容
(?=…)	零宽度肯定前瞻断言，检查当前位置之后的内容，要求其满足模式
(?!…)	零宽度否定前瞻断言，检查当前位置之后的内容，要求其不满足模式
(?<=…)	零宽度肯定后顾断言，检查当前位置之前的内容，要求其满足模式
(?<!…)	零宽度否定后顾断言，检查当前位置之前的内容，要求其不满足模式
(?(id/name)yes|no)	如果分组id或name匹配不为空，则要求匹配yes模式，否则要求匹配no模式。 no模式是可选的，即(?(id/name)yes)。比如r'(<)?\w+\b(?(1)>)'
(?#…)	表示后面为注释
(?flags)	必须出现在最开始的为止，指定匹配选项，flag为iLmsux中的一个或者多个字符
(?flags1-flags2:pattern)	给出匹配pattern时采用的选项，即设置flags1的选项，去掉flags2中的选项

主要内容

- 正则表达式
- 模式对象和相应方法
- 子模式扩展(拓展的内容)
- **替换和分割**

替换sub和subn

re.sub(pattern, replace, string, count=0, flags=0) regexp.sub(replace, string, count=0)	将字符串string中与pattern匹配的内容用replace替换，返回新的字符串。count表示最多替换多少次，缺省为0，表示全部替换。replace可以是字符串或函数对象
re.subn(pattern, replace, string, count=0, flags=0) regexp.subn(replace, string, count=0)	与sub类似，只是返回的是一个元组，第1个元素为替换后产生的新字符串，第2个元素为总共替换的次数

replace为字符串时，其中还可以包含分组引用：

- \N或者\g<N>　表示第N个分组匹配的内容
- \g<name>　表示名字为name的分组匹配的内容

```
text = '''Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.'''
```

re_sub.py

```
# 将以字母"b"和"B"开头的单词替换为"*"
pattern = r'\b[bB]\w*\b'
print(re.sub(pattern, '*', text))

# 将第一个以字母"b"和"B"开头的单词替换为"*"
print(re.sub(pattern,'*',text, 1))

# 将以字母"b"和"B"开头的单词前面添加*
pattern = r'\b([bB]\w*)\b'
print(re.sub(pattern, r'*\1', text))
print(re.sub(pattern, r'*\g<1>', text))
```

替换sub和subn

re.sub(pattern, replace, string, count=0, flags=0)

- 参数replace传递的为字符串，表示替换成replace，在replace中可包含分组引用
- 参数replace也可以传递一个**函数对象**，该函数的参数为pattern所匹配的**match对象**。表示替换为调用replace(matchobj)所返回的字符串

re_sub.py

```
def re_sub_func():
    pattern = r'\b([BC]\w*)\b\s+is\s+(better|worse)\s+than\s+(\w+)\b'
    # B或C开头的单词  一个以上空格 is 空格  better或worse 空格  than 空格  单词
    def repl_func(match):
        first = match.group(1)
        last = match.group(3)
        return '%s is %s than %s' %   (last.capitalize(),
                'worse' if match.group(2) == 'better' else 'better',  first.lower())
    text_ = re.sub(pattern, repl_func, text)
    print()
    print(text_)
```

Complex is better than complicated 替换为 Complicated is worse than complex. 或作相反的替换

split

`re.split(pattern, string, maxsplit=0, flags=0)` `regexp.split(string, maxsplit=0)`

- 根据pattern分割字符串string，返回分割后的字符串列表。maxsplit为分割的最大次数，缺省为尽最大可能分割
- 拓展：如果pattern中包含捕获子模式，则作为分隔符的其中所有子模式的内容也在返回的字符串列表中→ 通过(?:...)表示不要捕获该子模式

```
text = 'alpha. beta....gamma delta'
print(re.split(r'[. ]+',text))
print(re.split(r'[. ]+',text, maxsplit=1))
```

分割符为一个以上字符(可以是.或空格)
['alpha', 'beta', 'gamma', 'delta']
['alpha', 'beta....gamma delta']

```
>>> re.split(r'(,+|;+)', '1,,2;;3')
['1', ',,', '2', ';;', '3']
```

```
>>> re.split(r'(?:,+|;+)', '1,,2;;3')
['1', '2', '3']
```

`text = 'aaa  bb  c d e  fff  '`	删除字符串中多余的空格，多种方法
`print(' '.join(text.split()))` `# 'aaa bb c d e fff'`	str的split方法用空格类字符分割字符串，返回['aaa', 'bb', 'c', 'd', 'e', 'fff']，然后合并成一个字符串，中间以空格隔开
`print(' '.join(re.split('\s+',text.strip())))`	首先strip()去掉首尾的空格类字符，然后调用re.split，分隔符模式为一个以上空格类字符
`print(re.sub('\s+', ' ', text.strip()))`	首先去掉首尾的空格类字符，然后调用re.sub将一个以上空格类字符对应的内容替换成一个空格

参考练习

1.验证一个字符串是否为一有效电子邮件格式

cs@fudan.edu.cn

cs&fudan.edu.cn

2.从输入字符串中清除HTML标记

- 如对于字符串'Welcome to Fudan University!'处理后，输出为Welcome to Fudan University!