第8章 正则表达式

第8章 正则表达式

- 正则表达式是字符串处理的有力工具，正则表达式使用预定义的模式去匹配一类具有共同特征的字符串，可以快速、准确地完成复杂的查找、替换、分隔等处理要求，比字符串自身提供的方法提供了更强大的处理功能。
- 例如，使用字符串对象的split()方法只能指定一种分隔符，而使用正则表达式可以很方便地指定多种分隔符；使用字符串对象的split()并指定分隔符时，很难处理分隔符连续多次出现的情况，而正则表达式让这一切都变成非常轻松。
- 正则表达式在文本编辑与处理、网页爬虫之类的场合中有重要应用。

8.1 正则表达式语法

- 正则表达式由元字符及其不同组合来构成，通过巧妙地构造正则表达式可以匹配任意字符串，并完成查找、替换、分隔等复杂的字符串处理任务。

8.1.1 正则表达式基本语法

元字符	功能说明
.	默认匹配除换行符以外的任意单个字符，单行模式下也可以匹配换行符。表示普通圆点时需要在前面加反斜线，在方括号中时除外
*	匹配位于*之前的字符或子模式的0次或多次出现
+	匹配位于+之前的字符或子模式的1次或多次出现
-	在[]之内用来表示范围，其他位置作为普通减号字符
|	匹配位于|之前或之后的模式
^	1）匹配行首，匹配以^后面的字符开头的字符串；2）在[]中第一个字符时表示不匹配方括号中的字符
$	匹配行尾，匹配以$之前的字符结束的字符串。以^开头且以$结尾的模式表示匹配整个字符串
?	1）匹配位于?之前的字符或子模式的0次或1次出现；2）当此字符紧随任何表示重复次数的限定符（*、+、?、{n}、{n,}、{n,m}）之后时，匹配模式是“非贪心的”。“非贪心的”模式匹配搜索到的、尽可能短的字符串，而默认的“贪心的”模式匹配搜索到的、尽可能长的字符串
\	非原始字符串中尝试对单个反斜线后面的字符或数字进行转义
\num	此处的num是一个正整数，表示子模式编号，编号从1开始，0表示整个正则表达式。这个用法与转义字符冲突，需要使用原始字符串或者两个反斜线
\f	换页符
\n	换行符

8.1.1 正则表达式基本语法

元字符	功能说明
\r	匹配一个回车符
\b	匹配单词头或单词尾，这个符号的含义与转义字符冲突，需要使用原始字符串
\B	与\b含义相反
\d	匹配单个任意数字，相当于[0-9]
\D	与\d含义相反，等效于[^0-9]
\s	匹配单个任意空白字符，包括空格、制表符、换页符、换行符、回车符，与 [\f\n\r\t\v]等效
\S	与\s含义相反
\w	匹配任何汉字、字母、数字以及下划线
\W	与\w含义相反\w含义相反
()	将位于()内的内容作为一个整体来对待，表示普通括号时需要在前面加反斜线
{m,n}	{}前的字符或子模式重复至少m次，至多n次，逗号前后不能有空格，有{m}、{m,}、{,n}、{m,n}等变形用法
[]	表示范围，匹配位于[]中的任意一个字符，方括号中每个字符都表示字面意思。表示普通括号时需要在前面加反斜线
[^xyz]	反向字符集，匹配除x、y、z之外的任何字符
[a-z]	字符范围，匹配指定范围内的任何字符
[^a-z]	反向范围字符，匹配除小写英文字母之外的任何字符

8.1.1 正则表达式基本语法

- 在字符串前加上字符r或R之后表示原始字符串，字符串中任意字符都不再进行转义。原始字符串可以减少用户的输入，主要用于正则表达式和文件路径字符串的情况，但如果字符串以一个斜线“\”结束的话，则需要多写一个斜线，即以“\\”结束。
- 如果以“\”开头的元字符与转义字符形式相同但含义不同相同，则需要使用“\\”，或者使用原始字符串。

8.1.2 正则表达式扩展语法

- 正则表达式使用圆括号“()”表示一个子模式，圆括号内的内容作为一个整体对待，例如'(red)+'可以匹配'redred'、'redredred'等一个或多个重复'red'的情况。
- 子模式扩展语法用来对子模式进行限制和修饰，实现更加复杂的字符串处理功能。

8.1.2 正则表达式扩展语法

语法	功能说明
(?P<groupname>)	为当前子模式命名
(?iLmsux)	设置匹配标志，可以是几个字母的组合，每个字母含义与编译标志相同
(?:...)	匹配但不捕获该子模式匹配到的内容，使用re.findall()函数搜索时不返回该子模式匹配到的内容
(?P=groupname)	表示在此之前的命名为groupname的子模式匹配到的内容在当前位置又出现一次
(?#...)	表示注释
(?<=...)	用于正则表达式之前，表示如果<=后的内容在字符串中出现则匹配，但不返回<=之后的内容
(?=...)	用于正则表达式之后，表示如果=后的内容在字符串中出现则匹配，但不返回=之后的内容
(?<!...)	用于正则表达式之前，表示如果<!后的内容在字符串中不出现则匹配，但不返回<!之后的内容
(?!...)	用于正则表达式之后，表示如果!后的内容在字符串中不出现则匹配，但不返回!之后的内容

8.1.3 正则表达式集锦

- ✓ 最简单的正则表达式是普通字符串，可以匹配自身
- ✓ '[pjc]ython'可以匹配'python'、'jython'、'cython'
- ✓ '[a-zA-Z0-9]'可以匹配一个任意大小写字母或数字
- ✓ '[^abc]'可以匹配一个除'a'、'b'、'c'之外的任意字符
- ✓ 'python|perl'或'p(ython|erl)'都可以匹配'python'或'perl'
- ✓ r'(https://)?(www\.)?python\.org'可以匹配'https://www.python.org'、'https://python.org'、'www.python.org'和'python.org'
- ✓ '^http'只能匹配所有以'http'开头的字符串
- ✓ (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

8.1.3 正则表达式集锦

- ✓ '(a|b)*c'：匹配多个（包含0个）a或b，后面紧跟一个字母c。
- ✓ 'ab{1,}'：等价于'ab+'，匹配以字母a开头后面带1个至多个字母b的字符串。
- ✓ '^[a-zA-Z]{1}[a-zA-Z0-9._]{4,19}$'：匹配长度为5-20的字符串，必须以字母开头并且可带字母、数字、“_”、“.”的字符串。
- ✓ '^\w{6,20}$'：匹配长度为6-20的字符串，可以包含汉字、字母、数字、下划线。
- ✓ '^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$'：检查给定字符串是否为合法IP地址格式。
- ✓ '^(13[4-9]\d{8})|(15[01289]\d{8})$'：检查给定字符串是否为移动手机号码。
- ✓ '^[a-zA-Z]+$'：检查给定字符串是否只包含英文字母大小写。
- ✓ '^\w+@(\w+\.)+\w+$'：检查给定字符串是否为合法电子邮件地址。
- ✓ r'(\w)(?!.*\1)'：查找字符串中每个字符的最后一次出现。
- ✓ r'(\w)(?=.*\1)'：查找字符串中所有重复出现的字符。

8.1.3 正则表达式集锦

- ✓ '^-?\d+(\.\d{1,2})?$'：检查给定字符串是否为最多带有2位小数的正数或负数。
- ✓ '[\u4e00-\u9fa5]'：匹配常用汉字。
- ✓ '^\d{17}X|\d{18}|\d{15}$'：检查给定字符串是否身份证号格式。
- ✓ '\d{4}-\d{1,2}-\d{1,2}'：匹配指定格式的日期，例如2023-1-31。
- ✓ '^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[,._]).{8,}$'：检查给定字符串是否为强密码，必须同时包含英语字母大写字母、英文小写字母、数字或特殊符号（如英文逗号、英文句号、下划线），并且长度必须至少8位。
- ✓ "(?!.*[\'\"\/;=%?]).+"：如果给定字符串中包含'、"、/、;、=、%、?则匹配失败。
- ✓ '(.)\\1+'：匹配任意字符的一次或多次重复。
- ✓ r'((?P<f>\b\w+\b)\s+(?P=f))'：匹配连续出现两次的单词。
- ✓ '((?P<f>.)(?P=f)(?P<g>.)(?P=g))'：匹配AABB形式的成语或字母组合。

8.1.3 正则表达式集锦

- 使用时要注意的是，正则表达式只是进行形式上的检查，并不保证内容一定正确，除非增加更多约束条件，但那样可能会使得正则表达式非常复杂。
- 例如上面的例子中，正则表达式'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$'可以检查字符串是否为IP地址，字符串'888.888.888.888'这样的也能通过检查，但实际上并不是有效的IP地址。
- 同样的道理，正则表达式'^\d{17}X|\d{18}|\d{15}$'只负责检查字符串是否为18位或15位数字，并不保证一定是合法的身份证号，也没有考虑最后一位是字母的身份证号。

8.2 使用正则表达式模块re的函数处理字符串

- Python标准库re模块提供了正则表达式操作所需要的功能。

方法	功能说明
compile(pattern[, flags])	编译正则表达式模式，创建模式对象
escape(string)	将字符串中所有特殊正则表达式字符转义
findall(pattern, string[, flags])	返回包含字符串中所有与给定模式匹配的项的列表
finditer(pattern, string, flags=0)	返回包含所有匹配项的迭代对象，其中每个匹配项都是Match对象
fullmatch(pattern, string, flags=0)	尝试把模式作用于整个字符串，返回Match对象或None
match(pattern, string[, flags])	从字符串开始处进行匹配，返回Match对象或None
purge()	清空正则表达式缓存
search(pattern, string[, flags])	在整个字符串中查找模式匹配项，返回Match对象或None
split(pattern, string[, maxsplit=0])	根据模式匹配项分隔字符串
sub(pat, repl, string[, count=0])	将字符串中所有与pat匹配的项用repl替换或处理，返回新字符串，repl可以是字符串，也可以是返回字符串的可调用对象（作用于每个匹配的Match对象）
subn(pat, repl, string[, count=0])	将字符串中所有pat的匹配项用repl替换，返回包含新字符串和替换次数的二元元组，repl可以是字符串或返回字符串的可调用对象（作用于每个匹配的Match对象）

8.2 使用正则表达式模块re的函数处理字符串

- 其中函数参数“flags”的值可以是下面几个的不同组合（使用“|”进行组合）：

✓re.I（忽略大小写）

✓re.L（支持本地字符集的字符）

✓re.M（多行匹配模式）

✓re.S（使元字符“.”匹配任意字符，包括换行符）

✓re.U（匹配Unicode字符）

✓re.X（忽略模式中的空格，并可以使用#注释）

8.2 使用正则表达式模块re的函数处理字符串

```
>>> import re                                    # 导入re模块
>>> text = 'alpha. beta....gamma delta'         # 测试用的字符串
>>> re.split('[\. ]+', text)                    # 使用指定字符作为分隔符进行分隔
                                                # 反括号中的每个字符都表示字面值
                                                # 在方括号中圆点前可以不加反斜线
['alpha', 'beta', 'gamma', 'delta']
>>> re.split('[\. ]+', text, maxsplit=2) # 最多分隔2次
['alpha', 'beta', 'gamma delta']
>>> re.split('[\. ]+', text, maxsplit=1) # 最多分隔1次
['alpha', 'beta....gamma delta']
>>> pat = '[a-zA-Z]+'
>>> re.findall(pat, text)                       # 查找连续的字母组合
['alpha', 'beta', 'gamma', 'delta']
```

8.2 使用正则表达式模块re的函数处理字符串

```
>>> pat = '{name}'
>>> text = 'Dear {name}...'
>>> re.sub(pat, 'Mr.Dong', text)          # 字符串替换
'Dear Mr.Dong...'
>>> s = 'a s d'
>>> re.sub('a|s|d', 'good', s)           # 字符串替换
'good good good'
>>> s = "It's a very good good idea"
>>> re.sub(r'(\b\w+) \1', r'\1', s)      # 处理连续的重复单词
"It's a very good idea"
>>> re.sub(r'((\w+) )\1', r'\2', s)      # 这个处理结果不太理想
"It's a very goodidea"
>>> re.sub('a', lambda x: x.group(0).upper(), 'aaa abc abde')
                                         # repl可以为可调用对象
'AAA Abc Abde'
```

8.2 使用正则表达式模块re的函数处理字符串

```
>>> re.sub('[a-z]', lambda x: x.group(0).upper(), 'aaa abc abde')
'AAA ABC ABDE'
>>> re.sub('[a-zA-z]', lambda x: chr(ord(x.group(0))^32), 'aaa aBc abde')
'AAA AbC ABDE'
>>> re.sub('[a-zA-z]', lambda x: x.group(0).swapcase(), 'aaa aBc abde')
                                                    # 英文字母大小写互换
'AAA AbC ABDE'
>>> re.subn('a', 'dfg', 'aaa abc abde')      # 返回新字符串和替换次数
('dfgdfgdfg dfgbc dfgbde', 5)
>>> re.sub('a', 'dfg', 'aaa abc abde')
'dfgdfgdfg dfgbc dfgbde'
```

8.2 使用正则表达式模块re的函数处理字符串

```
>>> print(re.match('done|quit', 'done'))          # 匹配成功，返回match对象
<re.Match object; span=(0, 4), match='done'>
>>> print(re.match('done|quit', 'done!'))         # 匹配成功
<re.Match object; span=(0, 4), match='done'>
>>> print(re.search('done|quit', 'd!one!done'))   # 匹配成功
<re.Match object; span=(6, 10), match='done'>
>>> print(re.match('done|quit', 'doe!'))          # 匹配不成功，返回空值None
None
>>> print(re.match('done|quit', 'd!one!'))        # 匹配不成功
None
```

8.2 使用正则表达式模块re的函数处理字符串

- 下面的代码使用不同的方法删除字符串中多余的空格，如果遇到连续多个空格则只保留一个，同时删除字符串两侧的所有空白字符。

```
>>> import re
>>> s = 'aaa      bb      c d e   fff    '
>>> ' '.join(s.split())                          # 直接使用字符串对象的方法
'aaa bb c d e fff'
>>> ' '.join(re.split('[\s]+', s.strip())) # 同时使用re中的函数和字符串对象的方法
'aaa bb c d e fff'
>>> ' '.join(re.split('\s+', s.strip()))   # 与上一行代码等价
'aaa bb c d e fff'
>>> re.split('\s+', s)                           # 注意最后的空字符串
['aaa', 'bb', 'c', 'd', 'e', 'fff', '']
>>> re.sub('\s+', ' ', s.strip())           # 直接使用re模块的字符串替换方法
'aaa bb c d e fff'
```

8.2 使用正则表达式模块re的函数处理字符串

- 下面的代码使用几种不同的方法来删除字符串中指定内容：

```
>>> email = 'dongfuguo2005@remove_this126.com'
>>> m = re.search('remove_this', email)     # 使用search()方法返回的Match对象
>>> email[:m.start()] + email[m.end():]     # 字符串切片
'dongfuguo2005@126.com'
>>> re.sub('remove_this', '', email)        # 直接使用re模块的sub()方法
'dongfuguo2005@126.com'
>>> email.replace('remove_this', '')        # 直接使用字符串替换方法
'dongfuguo2005@126.com'
```

8.2 使用正则表达式模块re的函数处理字符串

- 下面的代码使用以“\”开头的元字符来实现字符串的特定搜索。

```
>>> import re
>>> example = 'Beautiful is better than ugly.'
>>> re.findall('\\bb.+?\\b', example)    # 以字母b开头的单词，问号表示非贪心模式
                                         # \\b是为了避免和转义字符\b冲突
['better']
>>> re.findall('\\bb.+\\b', example)     # 贪心模式，且此处圆点可以匹配空格
['better than ugly']
>>> re.findall('\\bb\w*\\b', example)    # \w不能匹配空格
['better']
>>> re.findall('\\Bh.+?\\b', example)    # 不以h开头且含有h字母的单词剩余部分
['han']
```

8.2 使用正则表达式模块re的函数处理字符串

```
>>> re.findall('\\b\w.+?\\b', example)                    # 所有单词
['Beautiful', 'is', 'better', 'than', 'ugly']
>>> re.findall(r'\b\w.+?\b', example)                     # 使用原始字符串
['Beautiful', 'is', 'better', 'than', 'ugly']
>>> re.findall('\w+', example)                            # 等价写法
['Beautiful', 'is', 'better', 'than', 'ugly']
>>> re.split('\s', example)                               # 使用任何空白字符分隔字符串
['Beautiful', 'is', 'better', 'than', 'ugly.']
>>> re.findall('\d+\.\d+\.\d+', 'Python 2.7.18') # 查找并返回x.x.x形式的数字
['2.7.18']
>>> re.findall('\d+\.\d+\.\d+', 'Python 2.7.18,Python 3.9.0')
['2.7.18', '3.9.0']
```

8.2 使用正则表达式模块re的函数处理字符串

```
>>> s = '<html><head>This is head.</head><body>This is body.</body></html>'
>>> pattern = r'<html><head>(.+)</head><body>(.+)</body></html>'
>>> result = re.search(pattern, s)
>>> result.group(0)                          # 整个正则表达式匹配到的内容为模式0
'<html><head>This is head.</head><body>This is body.</body></html>'
>>> result.group(1)                          # 第一个子模式
'This is head.'
>>> result.group(2)                          # 第二个子模式
'This is body.'
>>> re.findall(pattern, s)                   # 正则表达式中含有子模式
                                             # 此时findall()只返回子模式匹配的内容
[('This is head.', 'This is body.')]
```

8.3 使用正则表达式对象处理字符串

- 首先使用re模块的compile()函数将正则表达式编译生成正则表达式对象，然后再使用正则表达式对象提供的方法进行字符串处理。
- 使用编译后的正则表达式对象可以提高字符串处理速度，也提供了更强大的文本处理功能。

8.3 使用正则表达式对象处理字符串

- match()、search()、findall()

✓ match(string[, pos[, endpos]])方法在字符串开头或指定范围的开始位置进行搜索，模式必须出现在字符串开头或指定范围的开始位置；

✓ search(string[, pos[, endpos]])方法在整个字符串或指定范围中进行搜索；

✓ findall(string[, pos[, endpos]])方法在字符串指定范围中查找所有符合正则表达式的字符串并以列表形式返回。

8.3 使用正则表达式对象处理字符串

```
>>> import re
>>> example = 'ShanDong Institute of Business and Technology'
>>> pattern = re.compile(r'\bB\w+\b')          # 查找以B开头的单词
>>> pattern.findall(example)                   # 使用正则表达式对象的findall()方法
['Business']
>>> pattern = re.compile(r'\w+g\b')            # 查找以字母g结尾的单词
>>> pattern.findall(example)
['ShanDong']
>>> pattern = re.compile(r'\b[a-zA-Z]{3}\b')   # 查找3个字母长的单词
>>> pattern.findall(example)
['and']
```

8.3 使用正则表达式对象处理字符串

```
>>> pattern.match(example)                    # 从字符串开头开始匹配，失败返回空值
>>> pattern.search(example)                   # 在整个字符串中搜索，成功
<_sre.SRE_Match object; span=(31, 34), match='and'>
>>> pattern = re.compile(r'\b\w*a\w*\b') # 查找所有含有字母a的单词
>>> pattern.findall(example)
['ShanDong', 'and']
>>> text = 'He was carefully disguised but captured quickly by police.'
>>> re.findall(r'\w+ly', text)                # 查找所有以字母组合ly结尾的单词
['carefully', 'quickly']
```

8.3 使用正则表达式对象处理字符串

■ sub()、subn()

✓ 正则表达式对象的sub(repl, string[, count = 0])和subn(repl, string[, count = 0])方法用来实现字符串替换功能，其中参数repl可以为字符串或返回字符串的可调用对象。

```
>>> example = '''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.'''
```

8.3 使用正则表达式对象处理字符串

```
>>> pattern = re.compile(r'\bb\w*\b', re.I) # 匹配以b或B开头的单词
                                            # re.I表示忽略大小写
>>> print(pattern.sub('*', example))        # 将符合条件的单词替换为*
* is * than ugly.
Explicit is * than implicit.
Simple is * than complex.
Complex is * than complicated.
Flat is * than nested.
Sparse is * than dense.
Readability counts.
```

8.3 使用正则表达式对象处理字符串

```
>>> print(pattern.sub(lambda x: x.group(0).upper(), example))
                                                    # 把所有匹配项都改为大写


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.
```

8.3 使用正则表达式对象处理字符串

```
>>> print(pattern.sub('*', example, 1))        # 只替换1次

* 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.
```

8.3 使用正则表达式对象处理字符串

```
>>> pattern = re.compile(r'\bb\w*\b')    # 匹配以字母b开头的单词，区分大小写
>>> print(pattern.sub('*', example, 1)) # 将符合条件的单词替换为*
                                         # 只替换1次
Beautiful is * 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.
```

8.3 使用正则表达式对象处理字符串

- 正则表达式对象的split(string[, maxsplit = 0])方法用来实现字符串分隔。

```
>>> example = r'one,two,three.four/five\six?seven[eight]nine|ten'
>>> pattern = re.compile(r'[,./\\?[\]\|]')      # 指定多个可能的分隔符
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
>>> example = r'one1two2three3four4five5six6seven7eight8nine9ten'
>>> pattern = re.compile(r'\d+')                 # 使用数字作为分隔符
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
>>> example = r'one two    three  four,five.six.seven,eight,nine9ten'
>>> pattern = re.compile(r'[\s,.\d]+')           # 允许分隔符重复
>>> pattern.split(example)
['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

8.4 Match对象

- 正则表达式对象的match()方法和search()方法以及re模块的同名函数匹配成功后返回Match对象。Match对象的主要方法有：

✓ group()：返回匹配的一个或多个子模式内容

✓ groups()：返回一个包含匹配的所有子模式内容的元组

✓ groupdict()：返回包含匹配的所有命名子模式内容的字典

✓ start()：返回指定子模式内容的起始位置

✓ end()：返回指定子模式内容的结束位置的下一个位置

✓ span()：返回一个包含指定子模式内容起始位置和结束位置下一个位置的元组。

8.4 Match对象

```
>>> m = re.match(r'(\w+) (\w+)', 'Isaac Newton, physicist')
>>> m.group(0)                          # 返回整个模式匹配到的内容
'Isaac Newton'
>>> m.group(1)                          # 返回第1个子模式匹配到的内容
'Isaac'
>>> m.group(2)                          # 返回第2个子模式匹配到的内容
'Newton'
>>> m.group(1, 2)                       # 返回指定的多个子模式匹配到的内容
('Isaac', 'Newton')
```

8.4 Match对象

```
>>> m = re.match(r'(?P<first_name>\w+) (?P<last_name>\w+)',
                 'Malcolm Reynolds')
>>> m.group('first_name')      # 使用命名的子模式
'Malcolm'
>>> m.group('last_name')
'Reynolds'
>>> m = re.match(r'(\d+)\.(\d+)', '24.1632')
>>> m.groups()                 # 返回所有匹配的子模式（不包括第0个）
('24', '1632')
>>> m = re.match(r'(?P<first_name>\w+) (?P<last_name>\w+)',
                 'Malcolm Reynolds')
>>> m.groupdict()              # 以字典形式返回匹配的结果
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

8.4 Match对象

```
>>> exampleString = '''There should be one-- and preferably only one --
obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than right now.'''
>>> pattern = re.compile(r'(?<=\w\s)never(?=\s\w)')
                                        # 查找不在句子开头和结尾的

never
>>> matchResult = pattern.search(exampleString)
>>> matchResult.span()
(172, 177)
```

8.4 Match对象

```
>>> pattern = re.compile(r'\b(?i)n\w+\b')    # 查找以n或N字母开头的所有单词
>>> index = 0
>>> while True:
    matchResult = pattern.search(exampleString, index)
    if not matchResult:
        break
    print(matchResult.group(0), ':', matchResult.span(0))
    index = matchResult.end(0)

not : (92, 95)
Now : (137, 140)
never : (156, 161)
never : (172, 177)
now : (205, 208)
```

8.4 Match对象

```
>>> pattern = re.compile(r'(?<!not\s)be\b')
                                          # 查找前面没有单词not的单词be
>>> index = 0
>>> while True:
    matchResult = pattern.search(exampleString, index)
    if not matchResult:
        break
    print(matchResult.group(0), ':', matchResult.span(0))
    index = matchResult.end(0)

be : (13, 15)
>>> exampleString[13:20]                  # 验证一下结果是否正确
'be one-'
```

8.4 Match对象

```
>>> pattern = re.compile(r'(\b\w*(?P<f>\w+)(?P=f)\w*\b)')     # 有连续相同字母的单词
>>> index = 0
>>> while True:
    matchResult = pattern.search(exampleString, index)
    if not matchResult:
        break
    print(matchResult.group(0), ':', matchResult.group(2))
    index = matchResult.end(0) + 1
unless : s
better : t
better : t
>>> s = 'aabc abcd abbcd abccd abcdd'
>>> pattern.findall(s)
[('aabc', 'a'), ('abbcd', 'b'), ('abccd', 'c'), ('abcdd', 'd')]
```

8.5 精彩案例赏析

- **例8-1** 使用正则表达式提取字符串中的电话号码。code\例8-1.py

```
import re

# 在三引号内每行后面加反斜线，表示不换行
telNumber = '''Suppose my Phone No. is 0535-1234567,\
yours is 010-12345678,\
his is 025-87654321.'''

pattern = r'\d{3,4}-\d{7,8}'
print(re.findall(pattern, telNumber))
```

8.5 精彩案例赏析

- **例8-2** 使用正则表达式批量检查网页文件是否被嵌入iframe框架。
  code\例8-2.py

```
import os
import re

def detectIframe(fn):
    with open(fn, encoding='utf8') as fp:
        content = fp.read()
    # 正则表达式
    m = re.findall(r'<iframe\s+src=.*?></iframe>', content)
    if m:
        # 返回文件名和被嵌入的框架
        return {fn:m}
    return False
```

8.5 精彩案例赏析

```
# 遍历当前文件夹中所有html和htm文件并检查是否被嵌入框架
for fn in (f for f in os.listdir('.') if f.endswith(('.html','.htm'))):
    r = detectIframe(fn)
    if not r:
        continue
    # 输出检查结果
    for k, v in r.items():
        print(k)
        for vv in v:
            print('\t', vv)
```

8.5 精彩案例赏析

- **例8-3** 在手机上安装“学习强国”APP，然后搜索一篇文章“气壮山河的凯歌 永载史册的丰碑——写在中国人民志愿军抗美援朝出国作战70周年之际”，复制其中的文字并保存到当前文件夹中以文章标题为名的记事本文件中。然后编写程序，使用正则表达式提取其中的关键日期和事件。code\例8-3.py

```
from re import findall

fn = r'气壮山河的凯歌 永载史册的丰碑—写在中国人民志愿军抗美援朝出国作战70周年之际.txt'

with open(fn, encoding='utf8') as fp:
    content = fp.read()

pattern = r'((((\d{4}年)?\d{1,2}月\d{1,2}日)|(\d{4}年)).+?。”?)'
result = findall(pattern, content)
for item in result:
    if '）' not in item[0]:
        print(item[0])
```