Python正则表达式

正则表达式（regular expression）发源于与计算机密切相关的两个领域：计算理论和形式语言。其主要功能是从字符串中通过特定的模式（pattern）搜索想要的内容。

给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

1. 给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）；
2. 可以通过正则表达式，从字符串中获取我们想要的特定部分。

正则表达式的特点

1. 灵活性、逻辑性和功能性非常的强
2. 可以迅速地用极简单的方式达到对字符串的复杂控制
3. 对于刚接触的人来说，比较晦涩难懂

正则表达式的语法

正则表达式由一些**普通字符**和一些**元字符**组成。普通字符就是我们平时常见的字符串、数字之类的，当然也包括一些常见的符号，等等。而元字符则可以理解为正则表达式引擎的保留字符，就像很多计算机语言中的保留字符一样，它们在正则引擎中有特殊的意义。

字符组

字符组就是一组字符，在正则表达式中，其表示“在同一个位置可能出现的各种字符”，其写法是在一对方括号`[`和`]`之间列出所有可能出现的字符，简单的字符组比如`[ab]`、`[314]`、`[#.?]`。

```
# 用正则表达式判断数字字符
import re


is_string = lambda astring: re.search('[0123456789]', astring) != None
is_string('1234')
```

`re.search()`是Python提供的正则表达式操作函数，表示“进行正则表示匹配”；`astring`是需要判断的字符串，而`[0123456789]`则是以字符串形式给出的正则表达式，它是一个字符组，表示这里可以是0、1、2、…、8、9中的任意一个字符。只要`astring`包含其中任何一个字符，就会得到一个**MatchObject**对象，否则，返回None。 字符组中的字符排列顺序并不影响字符组的功能，出现重复字符也不会影响，所以`[0123456789]`和`[9876543210]`、`[998877654332210]`完全等价。

正则表达式提供了**-范围表示法（range）**，它更直观，能进一步简化字符组。其形式为`[x-y]`，表示**x到y整个范围内的字符**。这样`[0123456789]`就可以表示为`[0-9]`，`[abcdefghijklmnopqrstuvwxyz]`就可以表示为`[a-z]`。一般情况下，字符组的范围表示法都表示一类字符（数字字符或者字母字符等）。

字符组中可以同时并列多个**-范围表示法（range）**，字符组`[0-9a-zA-Z]`可以匹配数字、大写字母或小写字母；字符组`[0-9a-fA-F]`可以匹配数字、大小写形式的a~f，它可以用来验证十六进制字符。

元字符与转义

字符组的开方括号`[`、闭方括号`]`和之前出现的`^`、`$`都算元字符。在匹配中，它们都有特殊含义。有时候只需要表示普通字符，就必须做特殊处理。

字符组中的`-`，如果其紧邻这字符组中的开方括号`[`，那么它就是普通字符，其他情况下都是元字符；而对于其他元字符，取消特殊含义的做法都是转义，即在其前面加上反斜线字符`\`。

如果要在字符组内部使用横线`-`，最好的办法是将它排列在字符组的最开头。`[-09]`就是包含三个字符`-`、`0`、`9`的字符组。

```
re.search("^[-09]$", "3") != None   # => False
re.search("^[-09]$", "-") != None   # => True

re.search("^[0\\-9]$", "3") != None  # => False
re.search("^[0\\-9]$", "-") != None  # => True
```

Python提供了**原生字符串**（Raw String），其非常适合正则表达式：正则表达式是怎样，原生字符串就是怎样，完全不需要考虑正则表达式之外的转义（只有双引号字符是例外，原生字符串内的双引号字符必须转义写成`\"`）。原生字符串的形式是`r"string"`。

```
r"^[0\-9]$" == "^[0\\-9]$"  # => True
```

排除型字符组

排除型字符组（Negated Character Class）非常类似普通字符组`[...]`，只是在开方括号`[`之后紧跟一个脱字符`^`，写作`[^...]`，表示“在当前位置，匹配一个没有列出的字符”。

```
# 第一个不是数字第二个是数字
re.search(r"^[^0-9][0-9]$", "A8") != None   # => True
re.search(r"^[^0-9][0-9]$", "08") != None   # => False
```

“在当前位置，匹配一个没有列出的字符”和“在当前位置不要匹配列出的字符”是不同的，后者暗示“这里不出现任何字符也可以”。排除型字符组必须匹配一个字符。

```
re.search(r"^[^0-9][0-9]$", "8") != None   # => False
```

在排除型字符组中，如果需要表示横线字符`-`，那么`-`应紧跟在`^`之后。

```
# 匹配一个-、0、9之外的字符
re.search(r"^[^-09]$", "-") != None    # => False
```

字符组简记法

- `\d` 等价于`[0-9]`，`d`代表“数字（digit）”
- `\w` 等价于`[0-9a-zA-Z_]`，`w`代表“单词字符（word）”
- `\s` 等价于`[ \t\r\n\v\f]`（第一个字符是空格），`s`表示“空白字符（space）”

注意：字符组简记法中的“单词字符”不只有大小写单词，还包括数字字符和下划线`_`。

“空白字符”可以是空格字符、制表符`\t`、回车符`\r`、换行符`\n`等各种“空白”字符。

字符组简记法可以单独出现，也可以使用在字符组中，比如`[0-9a-zA-Z]`也可以写作`[\da-zA-Z]`，`[^0-9a-zA-Z_]`可以写作`[^\w]`。

相对于`\d`、`\w`和`\s`这三个普通字符组简记法，正则表达式也提供了对应排除型字符组的简记法：`\D`、`\W`和`\S`——字母完全一样，只是改为大写。这些简记法匹配的字符互补：`\s`能匹配的字符，`\S`一定不能匹配；`\w`能匹配的字符，`\W`一定不能匹配；`\d`能匹配的字符，`\D`一定不能匹配。

利用这种互补的属性，就能得到巧妙的效果：`[\s\S]`、`[\w\W]`、`[\d\D]`匹配的就是“所有的字符”（或者叫“任意字符”）。

量词

匹配确定的长度或者不确定的长度。

- `prev{m}` 限定之前的元素出现m次。
- `prev{m,n}` 限定之前的元素最少出现m次，最多出现n次（均为闭区间）。

如果不确定长度的上限，也可以省略，只指定下限，写成`prev{m,}`。

常用量词

- `*`，等价于`{0,}`，可能出现，也可能不出现，出现次数没有上限
- `+`，等价于`{1,}`，至少出现1次，出现次数没有上限
- `?`，等价于`{0,1}`，至多出现一次，也可能不出现

使用正则表达式的一条根本规律：使用合适的结构（包括字符组和量词），精确表达自己的意图，界定能匹配的文本。

Python的`re`模块

```
import re

result = re.match(r'^travell?er$', 'traveler')
print(result)
result.group()
```

`match()`函数用于查看源（source）字符串是否以模式（pattern）字符串开头。

```
pattern = re.compile(r'travell?er')
pattern.match('traveler')
```

`re`模块其他可用的方法：

- `search()` 返回第一次成功匹配，如果存在的话；
- `findall()` 返回所有不重叠的匹配，如果存在的话；
- `split()` 会根据pattern将source切分成若干段，返回由这些片段组成的列表；
- `sub()` 需要一个额外的参数`replacement`，它会把source中所有匹配的pattern改成replacement。

```
# search() 寻找首次匹配
m = pattern.search('traveller')
if m:
   print(m.group())
```

```
# findall() 寻找所有匹配
m = pattern.findall('traveller and traveler')
m
```

```
# split() 按匹配切分
astring = 'a3b2c1d4e10'

def uncompress(astring):
   a = re.split(r'\d+', astring)[:-1]
   d = re.split(r'[a-zA-Z]', astring)[1:]
   return ''.join([a[i] * int(d[i]) for i in range(len(a))])

uncompress(astring)
```

模式

- 普通的文本值代表自身，用于匹配非特殊字符；
- 使用`.`代表任意除`\n`外的字符；
- 使用`*`表示任意多个字符（包括0个）；
- 使用`?`表示可选字符（0个或1个）。

```
import re
# 换行符的匹配
re.search(r'^.$', '\n') != None  # => False
# 单行模式
re.search(r'(?s)^.$', '\n') != None  # => True
# 自制“通配字符组”
re.search(r'^[\s\S]$', '\n') != None  # => True
```

特殊字符（参考教材p140）

正则表达式不仅仅适用于ASCII字符，还适用于Unicode的字符。

```
x = 'abc-/*\u00ea\u0115'
re.findall(r'\w', x)
```

模式标识符

模式	匹配
`abc`	文本值abc
`(expr)`	expr
`expr1|expr2`	`expr1`或`expr2`
`.`	除`\n`外的任何字符
`^`	源字符串的开头
`$`	源字符串的结尾
`prev?`	0个或1个`prev`
`prev*`	0个或多个`prev`，尽可能多地匹配
`prev*?`	0个或多个`prev`，尽可能少地匹配
`prev+`	1个或多个`prev`，尽可能多地匹配
`prev+?`	1个或多个`prev`，尽可能少地匹配
`prev{m}`	m个连续的`prev`
`prev{m,n}`	m到n个连续的`prev`，尽可能多地匹配
`prev{m,n}?`	m到n个连续的`prev`，尽可能少地匹配
`[abc]`	a或b或c，等价于`a|b|c`
`[^abc]`	非（a或b或c）
`prev(?=next)`	如果后面为`next`，返回`prev`
`prev(?!next)`	如果后面非`next`，返回`prev`
`(?<=prev)next`	如果前面为`next`，返回`prev`
`(?<!prev)next`	如果前面非`next`，返回`prev`

定义匹配的输出

使用`match()`或`search()`时，所有的匹配会以`MatchObject`对象返回，可以调用其`group()`方法获取匹配的结果。另外，可以使用括号`()`将某一模式包裹起来，括号中模式匹配得到的结果归入自己的group（无名称）中，等到匹配完成后，通过`group(num)`之类的方法“引用”分组在匹配时捕获的内容。其中`num`表示对应括号的编号，括号分组的编号规则是从左向右计数，从1开始。调用`m.groups()`可以得到包含这些匹配的元组。

```
# 匹配年月日
astring = '2017-04-07'
m = re.search(r'(\d{4})-(\d{2})-(\d{2})', astring)
print(m.group(1))
print(m.group(2))
print(m.group(3))
print(m.group(0))
print(m.group())
```

也有编号为0的分组，它是默认存在的，对应整个表达式匹配的文本。直接调用`group()`，不给出参数`num`，默认就等于调用`group(0)`。

如果正则表达式里包含嵌套的括号，其括号的编号将以这种方式定义：无论括号如何嵌套，分组的编号都是根据开括号出现顺序来计数的；开括号是从左向右数起第多少个开括号，整个括号分组的编号就是多少。

```
 (((\d{4})-(\d{2}))-(\d{2}))
0--------------------------
1--------------------------
 2-----------------
  3-------
          4-------
                   5-------
```

```
m = re.search(r'(((\d{4})-(\d{2}))-(\d{2}))', '2017-04-07')
print(m.group())
print(m.group(0))
print(m.group(1))
print(m.group(2))
print(m.group(3))
print(m.group(4))
print(m.group(5))
```

命名分组

`(?P<name>expr)`会匹配`expr`，并将匹配结果存储到名为`name`的组中。命名分组捕获时仍然保留了数字编号。

```
m = re.search(r'(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})', '2017-04-07')
print(m.group())
print(m.group(0))
print(m.groups())
print(m.group(1))
print(m.group('year'))
print(m.group(2))
print(m.group('month'))
print(m.group(3))
print(m.group('day'))
```

注意：不要弄错分组的结构！

```
astring = '2017-04-07'
print(re.search(r'(\d{4})-(\d{2})-(\d{2})', astring).group(1))
print(re.search(r'(\d){4}-(\d{2})-(\d{2})', astring).group(1))
```

第二个表达式中，编号为1的括号是`(\d)`，表示匹配一个数字字符，因为之后有量词`{4}`，所以整个括号作为单个元素，要重复出现4次，而且编号都是1；于是每重复出现一次，就要更新一次匹配结果。所以在匹配过程中，编号为1的分组匹配的文本的值依次是2、0、1、7，最后的结果是7。

正则表达式替换

分组捕获的文本，不仅仅用于数据提取，也可以用于替换，比如对于上面的例子，希望将**YYYY-MM-DD**格式的日期变为**MM/DD/YYYY**，就可以使用正则表达式替换。

替换方法：`re.sub(pattern, replacement, string)`

```
print(re.sub(r'[a-z]', ' ', 'a3b2c1d4'))
print(re.sub(r'[0-9]', ' ', 'a3b2c1d4'))
```

在`replacement`中也可以引用分组，形式是`\num`，其中的`num`是对应分组的编号，`replacement`是一个普通的字符串，也必须指定其为原生字符串。

```
print(re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\2/\3/\1', '2017-04-07'))
print(re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\1年\2月\3日', '2017-04-07'))
```

如果想在`replacement`中引用整个表达式匹配的文本，可以给整个表达式加上一对括号，之后用`\1`来引用。

```
re.sub(r'((\d{4})-(\d{2})-(\d{2}))', r'[\1]', '2017-04-07')
```

反向引用

如何检查某个单词是否包含重叠出现的字母（例如，shoot或beep）？

`[a-z][a-z]`可以吗?

“重叠出现”的字符，取决与第一个`[a-z]`在运行时的匹配结果，而不能预先设定，即必须知道之前匹配的确切内容。

反向引用（back-reference）允许在正则表达式内部引用之前的捕获分组匹配的文本（左侧），其形式也是`\num`，其中`num`表示所引用分组的编号，编号规则与之前介绍的相同。

```
re.search(r'^([a-z])\1$', 'aa') != None  # => True
re.search(r'^([a-z])\1$', 'ac') != None  # => False
```

```
# 用反向引用匹配成对的tag
paired_tag_regex = r'<([^>]+)>[\s\S]*?</\1>'
re.search(paired_tag_regex, '<bold>text</bold>') != None  # => True
re.search(paired_tag_regex, '<h1>text</bold>') != None  # => False
```

反向引用重复的是对应捕获分组匹配的文本，而不是之前的表达式。

具有二义性的反向引用

```
re.sub(r'(\d)', r'\10', '123')
# error: invalid group reference 10 at position 1
```

Python提供了`\g<num>`表示法，将`\10`写成`\g<1>0`，这样就避免了替换时无法使用`\0`的问题。

```
re.sub(r'(\d)', r'\g<1>0', '123')
```

命名分组的引用方法

如果使用了命名分组，在表达式中反向引用时，必须使用`(?P=name)`的记法。而要进行正则表达式替换，则需要写作`\g<name>`，其中`name`是分组的名字。

```
re.search(r'^(?P<char>[a-z])(?P=char)$', 'aa') != None  # => True

re.sub(r'(?P<digit>\d)', r'\g<digit>0', '123')  # => '102030'
```

正则表达式工具

- https://regexper.com
- https://www.debuggex.com