第5章 chapter 5

正则表达式

正则表达式(Regular Expression,Regex)描述了一种字符串匹配的模式,可以用来检查一个字符串是否含有某种子字符串。构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符,可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。


什么是正则表达式

5.1 什么是正则表达式

字符是计算机处理文字时的最基本单位,可能是字母、数字、标点符号、空格、换行符、汉字等。文本是字符的集合,文本也就是字符串。在处理字符串时,常常需要检查字符串中是否有满足某些规则(模式)的字符串。正则表达式就是用于描述这些规则的。说某个字符串匹配某个正则表达式,通常是指这个字符串里有一部分(或几部分分别)能满足正则表达式给出的规则。具体地说,正则表达式是一些由字符和特殊符号组成的字符串。例如:

'feelfree+b',可以匹配'feelfreeb'、'feelfreeeb'、'feelfreeeeeb'等,+号表示匹配位于+之前的字符一次或多次出现。

'feelfree * b',可以匹配'feelfreb'、'feelfreeb'、'feelfreeeeb'等,* 号表示匹配位于 * 之前的字符 0 次或多次出现。

'colou? r',可以匹配'color'或者'colour',? 表示匹配位于 ? 之前的字符 0 次或一次出现。


正则表达式的构成

5.2 正则表达式的构成

正则表达式是由普通字符(例如大写和小写字母、数字等)、预定义字符(例如\d 表示 0～9 的 10 个数字集[0-9],用于匹配数字)以及元字符(例如 * 匹配位于 * 之前的字符或子表达式 0 次或多次出现)组成的字符序列模式,也称为模板。模式描述了在搜索文本时要匹配的字符串。在 Python 中,通过 re 模块来实现正则表达式处理的功能。导入 re

模块后，使用如下的 search()函数可进行匹配。

```
re.search(pattern, string, flags=0)          #扫描整个字符串并返回第一个成功的匹配
```

函数参数说明如下。

pattern：匹配的正则表达式。

string：要匹配的字符串。

flags：标志位，用于控制正则表达式的匹配方式，如是否区分大小写、多行匹配等。

一些用“\”开始的字符表示预定义的字符，表 5-1 列出了一些常用的预定义字符。

表 5-1 常用的预定义字符

预定义字符	描 述
\d	表示 0～9 的 10 个数字集[0-9]，用于匹配数字
\D	表示非数字字符集，等价于[^0-9]，用于匹配一个非数字字符
\f	用于匹配一个换页符
\n	用于匹配一个换行符
\r	用于匹配一个回车符
\s	表示空白字符集[\f\n\r\t\v]，用于匹配空白字符，包括空格、制表符、换页符等
\S	表示非空白字符集，等价于[^ \f\n\r\t\v]，用于匹配非空白字符
\w	表示单词字符集[a-zA-Z0-9_]，用于匹配单词字符
\W	表示非单词字符集[^a-zA-Z0-9_]，用于匹配非单词字符
\t	用于匹配一个制表符
\v	用于匹配一个垂直制表符
\b	匹配单词头或单词尾
\B	与\b 含义相反

元字符是一些有特殊含义的字符。若要匹配元字符，必须首先使元字符“转义”，即将反斜杠字符“\”放在它们前面，使之失去特殊含义，成为一个普通字符。表 5-2 列出了一些常用的元字符。

表 5-2 常用的元字符

元字符	描 述
\	将下一个字符标记为特殊字符、原义字符、向后引用等。例如，'\n' 匹配换行符，'\\' 匹配 "\"，'\(' 则匹配 "("
.	匹配任何单字符(换行符\n 除外)。要匹配“.”，需使用\.
^...	匹配以^后面的“...”字符序列开头的行首，要匹配^字符本身，需使用 \^
... $	匹配以 $ 之前的字符结束的行尾。要匹配 $ 字符本身，需使用\ $
(...)	标记一个子表达式的开始和结束位置，即将位于()内的字符作为一个整体看待

续表

元字符	描　　述
*	匹配位于 * 之前的字符或子表达式 0 次或多次。要匹配 * 字符，需使用 *
+	匹配位于+之前的字符或子表达式的 1 次或多次。要匹配+字符，需使用\+
?	匹配位于“?”之前的字符或子表达式 0 次或 1 次。当“?”紧随任何其他限定符（*、+、?、{n}、{n,}、{m,n}）之后时，匹配模式是“非贪心的”。“非贪心的”模式匹配搜索到尽可能短的字符串，而默认的“贪心的”模式匹配搜索到尽可能长的字符串，如在字符串"oooo"中，“o+?”只匹配单个 o，而 o+匹配所有 o
{m}	匹配{m}之前的字符 m 次
{m, n}	匹配{m, n}之前的字符 m～n 次，m 和 n 可以省略，若省略 m，则匹配 0～n 次；若省略 n，则匹配 m 至无限次
[...]	匹配位于[...]中的任意一个字符
[^...]	匹配不在[...]中的字符，[^abc]表示匹配除了 a、b、c 之外的字符
|	匹配位于|之前或之后的字符。要匹配|，需使用 \\|

下面给出正则表达式的应用实例。

1. 匹配字符串字面值

正则表达式最为直接的功能就是用一个或多个字符字面值来匹配字符串。所谓字符字面值，就是字符看起来是什么就是什么。这和在 Word 等字符处理程序中使用关键字查找类似。当以逐个字符对应的方式在文本中查找字符串时，就是在用字符串字面值查找。

```
'python'                                    #匹配字符串'python3 python2'中的'python'
```

2. 匹配数字

预定义的字符\d 用于匹配任一阿拉伯数字，也可用字符组[0-9]替代\d 来匹配相同的内容，即\d 与[0-9]的效果是一样的。此外，也可以列出 0～9 内的所有数字[0123456789]来进行匹配。如果只想匹配 1 和 2 两个数字，可以使用字符组[12]来实现。

3. 匹配非数字字符

预定义的字符\D 用于匹配一个非数字字符，\D 与字符组[0-9]取反后的[^0-9]的作用相同（字符组取反的意思就是“不匹配这些”或“匹配除这些以外的内容”），也与[^\d]的作用一样。

'a\de'可以匹配'a1e'、'a2e'、'a0e'等，\d 匹配 0～9 的任一数字。

'a\De'可以匹配'aDe'、'ase'、'ave'等，\D 匹配一个非数字字符。

4. 匹配单词和非单词字符

预定义的字符\w用于匹配单词字符，与\w匹配相同内容的字符组为[a-zA-Z0-9_]。用\W匹配非单词字符，即用\W匹配空格、标点以及其他非字母、非数字字符。此外，\W与[^a-zA-Z0-9_]的作用一样。

'a\we'可以匹配'afe'、'a3e'、'a_e'，\w用于匹配大小写字母、数字或下画线字符。

'a\We'可以匹配'a.e'、'a，e'、'a * e'等字符串，\W用于匹配非单词字符。

'a[bcd]e'可以匹配'abe'、'ace'和'ade'，[bcd]匹配'b'、'c'和'd'中的任意一个。

5. 匹配空白字符

预定义的字符\s用于匹配空白字符，与\s匹配内容相同的字符组为[\f\n\r\t\v]，包括空格、制表符、换页符等。用\S匹配非空白字符，或者用[^\s]，或者用[^\f\n\r\t\v]。

'a\se'可以匹配'a e'，\s用于匹配空白字符。

'a\Se'可以匹配'afe'、'a3e'、'ave'等字符串，\S用于匹配非空白字符。

6. 匹配任意字符

用正则表达式匹配任意字符的一种方法是使用点号“.”，点号可以匹配任何单字符(换行符\n之外)。要匹配"hello world"这个字符串，可使用11个点号。但这种方法太麻烦，推荐使用量词.{11}，{11}表示匹配{11}之前的字符11次。再如'a.c'可以匹配'abc'、'acc'、'adc'等。

'ab{2}c'可以匹配'abbc'。'ab{1，2}c'，可完整匹配的字符串有'abc'和'abbc'，{1，2}表示匹配{1，2}之前的字符'b' 1次或2次。

'abc * '可以匹配'ab'、'abc'、'abcc'等字符串，* 表示匹配位于 * 之前的字符'c' 0次或多次。

'abc+'可以匹配'abc'、'abcc'、'abccc'等字符串，+表示匹配位于+之前的字符'c'1次或多次。

'abc? '可以匹配'ab'和'abc'字符串，? 表示匹配位于? 之前的字符'c' 0次或1次。

如果想查找元字符本身，例如用'.'查找“.”，就会出现问题，因为它们会被解释成特殊含义。这时就得使用\来取消该元字符的特殊含义。因此，查找“.”应该使用'\.'。要查找'\'本身，需要使用'\\'。

例如，'baidu\.com'匹配'baidu.com'，'C：\\ Program Files'匹配'C：\ Program Files'。

5.3 正则表达式的模式匹配

正则表达式的模式匹配

5.3.1 正则表达式的边界匹配

要对关键位置进行字符串匹配。例如，匹配一行文本的开头、一个字符串的开头或者结尾，这时就需要使用正则表达式的边界符来进行匹配。常用的边界符如表5-3所示。

表 5-3 常用的边界符

边界符	描　　述	边界符	描　　述
^	匹配字符串的开头或一行的开头	\b	匹配单词头或单词尾
$	匹配字符串的结尾或一行的结尾	\B	与\b 含义相反

匹配行首或字符串的起始位置要使用字符^。要匹配行或字符串的结尾位置要使用字符 $ 。

正则表达式'^We. * \. $ '可以匹配以 We 开头的整行。请注意结尾的点号之前有一个反斜杠，对点号进行转义，这样点号就被解释为字面值。如果不对点号进行转义，它就会匹配任意字符。

匹配单词边界要使用\b，如正则表达式'\bWe\b'匹配单词 We。\b 匹配一个单词的边界，如空格等，\b 匹配的字符串不会包括那个分界的字符，而如果用\s 来匹配，则匹配出的字符串中会包含那个边界符，如'\bHe\b'匹配一个单独的单词 He，而当它是其他单词的一部分时不匹配。

可以使用\B 匹配非单词边界，非单词边界匹配的是单词或字符串中间的字母或数字。

'^asddg' 可以匹配行首以'asddg'开头的字符串。

'rld $ ' 可以匹配行尾以'rld'结束字符串。

5.3.2 正则表达式的分组、选择和引用匹配

在使用正则表达式时，括号是一种很有用的工具。可以根据不同的目的用括号进行分组、选择和引用匹配。

1. 分组

在前面我们已经知道了怎么重复单个字符，即直接在字符后面加上诸如＋、*、{m，n}等重复操作符即可。但如果想要重复一个字符串，需要使用小括号来指定子表达式（也叫作分组或子模式），然后通过在小括号后面加上重复操作符来指定这个子表达式的重复次数。例如，'(abc){2}'可以匹配'abcabc'，{2}表示匹配{2}之前的表达式(abc)两次。

在 Python 中，分组就是用一对括号括起来的子正则表达式，匹配出的内容就表示匹配出了一个分组。从正则表达式的左边开始，遇到第一个左括号“(”表示该正则表达式的第一个分组，遇到第二个左括号“(”表示该正则表达式的第二个分组，以此类推。需要注意的是，有一个隐含的全局分组(就是 0 组)表示整个正则表达式。正则表达式分组匹配后，要想获得已经匹配的分组的内容时，就可以使用 group(num)和 groups()函数对各个分组进行提取。这是因为分组匹配到的内容会被临时存储于内存中，所以能够在需要时被提取。

```
>>>import re
```

```
>>>m=re.match(r'www\.(.*)\..{3}','www.python.org')  #正则表达式只包含一个分组
>>>print(m.group(1))                                 #提取分组 1 的内容
python
```

按照正则表达式进行匹配后，就可以通过分组提取到我们想要的内容，但是如果正则表达式中括号比较多，在提取想要的内容时，就需要去逐个数想要的内容是第几个括号中的正则表达式匹配的，这样会很麻烦。这时 Python 又引入了另一种分组，那就是命名分组，上面的叫无名分组。

命名分组就是给具有默认分组编号的组另外再取一个别名。命名分组的语法格式如下：

```
(?P<name>正则表达式)                                #name 是一个合法的标识符
>>>import re
>>>s ="ip='230.192.168.78',version='1.0.0'"
>>>h=re.search(r"ip='(?P<ip>\d+\.\d+\.\d+\.\d+).*", s)  #只有一个分组
>>>print(h.group('ip'))                              #通过分组别名提取分组 1 的内容
230.192.168.78
>>>print(h.group(1))                                 #提取分组 1 的内容
230.192.168.78
```

2. 选择操作

括号的另一个重要的应用是表示可选择性，根据可以选择的情况建立支持二选一或多选一的应用，这涉及括号()和竖线|两种元字符。|表示逻辑或的意思，如'a(123|456)b'可以匹配'a123b'和'a456b'。

假如要统计文本“When the fox first saw1 the lion he was2 terribly3 frightened4. He ran5 away, and hid6 himself7 in the woods.”中的 he 出现了多少次，he 的形式应包括 he 和 He 两种形式。查找 he 和 He 两个字符串的正则表达式可以写成：(he|He)。另一个可选的模式是(h|H) e。

假如要查找一个高校具有博士学位的教师，在高校的教师数据信息中，博士的写法可能有 Doctor、doctor、Dr.或 Dr，要匹配这些字符串可用下面的模式：

```
(Doctor|doctor|Dr\. |Dr)
```

注意：句点在正则表达式模式中是一个元字符，它可以匹配任何单字符(换行符\n除外)。要匹配“.”，需使用'\.'。上述模式的另一个可选的模式如下。

```
(Doctor|doctor|Dr\.?)
```

借助不区分大小写选项可使上述分组匹配更简单，选项(? i)可使匹配模式不再区分大小写，带选择操作的模式(he|He)就可以简写成(? i)he。表 5-4 列出了正则表达式中常用的选项。

表 5-4 正则表达式中常用的选项

选项	描　述	选项	描　述
(?i)	不区分大小写	(?s)	单行
(?m)	多行	(?U)	默认最短匹配

3. 分组后向引用

正则表达式中，用括号括起来的表示一个组。所谓分组后向引用，就是对前面出现过的分组再一次引用。当用括号定义了一个正则表达式组后，正则引擎就会把被匹配到的组按照顺序编号，存入缓存。

注意：()用于定义组，[]用于定义字符集，{}用于定义重复操作。

我们想在后面对已经匹配过的分组内容进行引用时，可以用“\数字”的方式或者通过命名分组“(?P=name)”的方式进行引用。\1 表示引用第一个分组，\2 表示引用第二个分组，以此类推，\n 表示引用第 n 个组。\0 则引用整个被匹配的正则表达式本身。这些引用都必须是在正则表达式中才有效，用于匹配一些重复的字符串。例如：

```
>>>import re
>>>re.search(r'(?P<name>\w+)\s+(?P=name)\s+(?P=name)', 'python python
python').group('name')              #通过命名分组进行后向引用
'python'
>>>re.search(r'(?P<name>\w+)\s+(?P=name)\s+(?P=name)', 'python python
python').group()
'python python python'
>>>re.search(r'(?P<name>\w+)\s+\1\s+\1', 'python python python').group()
'python python python'              #通过默认分组编号进行后向引用
```

下面看一个嵌套分组的例子：

```
>>>s ='2017-10-10 18:00'
>>>import re
>>>p =re.compile(r'(((\d{4})-\d{2})-\d{2}) (\d{2}):(\d{2})')
>>>re.findall(p,s)
[('2017-10-10', '2017-10', '2017', '18', '00')]
>>>se =re.match(p,s)
>>>print(se.group())
2017-10-10 18:00
>>>print(se.group(0))
2017-10-10 18:00
>>>print(se.group(1))
2017-10-10
>>>print(se.group(2))
2017-10
>>>print(se.group(3))
```

```
2017
>>>print(se.group(4))
18
>>>print(se.group(5))
00
```

可以看出,分组的序号是以左小括号“(”从左到右的顺序为准的。

```
>>>import re
>>>s ='1234567890'
>>>s =re.sub(r'(...)',r'\1,',s)  #在字符串中从前往后每隔 3 个字符插入一个“,”符号
>>>s
'123,456,789,0'
```

5.3.3 正则表达式的贪婪匹配与懒惰匹配

当正则表达式中包含重复的限定符时,通常的行为是(在使整个表达式能得到匹配的前提下)匹配尽可能多的字符。例如'a.*b',它将会匹配最长的以 a 开始、以 b 结束的字符串。如果用它来匹配'aabab',它会匹配整个字符串'aabab'。这被称为贪婪匹配。

有时,我们需要懒惰匹配,也就是匹配尽可能少的字符。前面给出的限定符都可以被转化为懒惰匹配模式,只要在*后面加上一个问号“?”。这样'a.*?b'就意味着匹配任意数量的重复,但是在使整个表达式能得到匹配的前提下使用最少的重复。

'a.*?b'匹配最短的以 a 开始、以 b 结束的字符串。如果把它应用于'aabab',它会匹配'aab'(第一到第三个字符)和'ab'(第 4～5 个字符)。匹配的结果为什么不是最短的'ab'而是'aab'和'ab'? 这是因为正则表达式有另一条规则,比懒惰/贪婪规则的优先级更高,这就是“最先开始的匹配拥有最高的优先权”。表 5-5 列出了常用的懒惰限定符。

表 5-5 常用的懒惰限定符

懒惰限定符	描 述
*?	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
{n,m}?	重复 n～m 次,但尽可能少重复
{n,}?	重复 n 次以上,但尽可能少重复

```
>>>import re
>>>s ="abcdakdjd"
>>>p =re.compile("a.*?d")        #懒惰匹配
>>>m =re.compile("a.*d")         #贪婪匹配
>>>p.findall(s)
['abcd', 'akd']
```

```
>>>m.findall(s)
['abcdakdjd']
```


正则表达式模块 re

5.4 正则表达式模块 re

Python 通过 re 模块提供对正则表达式的支持。表 5-6 列出了常用的 re 模块函数。

表 5-6 常用的 re 模块函数

函 数	描 述
re.compile(pattern[, flags])	把正则表达式 pattern 转化成正则表达式对象，然后可以通过正则表达式对象调用 match()和 search()方法
re.split(pattern, string[, maxsplit=0, flags])	用匹配 pattern 的子串来分隔 string，并返回一个列表
re.match(pattern, string[, flags])	从字符串 string 的起始位置匹配模式 pattern，string 如果包含 pattern 子串，则匹配成功，返回 Match 对象，失败则返回 None
re.search(pattern, string[, flags])	若 string 中包含 pattern 子串，则返回 Match 对象，否则返回 None。注意，如果 string 中存在多个 pattern 子串，只返回第一个
re.findall(pattern, string[, flags])	找到模式 pattern 在字符串 string 中的所有匹配项，并把它们作为一个列表返回
re.sub(pattern, repl, string[, count=0, flags])	替换匹配到的字符串，即用 pattern 在 string 中匹配要替换的字符串，然后把它替换成 repl
re.escape(string)	对字符串 string 中的非字母数字进行转义，返回非字母数字前加反斜杠的字符串

函数参数说明如下。

pattern：匹配的正则表达式。

string：要匹配的字符串。

flags：用于控制正则表达式的匹配方式，flags 的值可以是 re.I(表示忽略大小写)、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.X(忽略模式中的空格，并可以使用#注释)的不同组合(使用|进行组合)。

repl：用于替换的字符串，也可为一个函数。

count：模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。

1. re.search()函数

re.search()函数会在字符串内查找模式的匹配字符串，只要找到第一个和模式相匹配的字符串就立即返回，返回一个 Match 对象，如果没有匹配的字符串，则返回 None。Match 对象有以下方法。

group()：返回被 re.search()匹配的字符串。

start()：返回匹配开始的位置。

end()：返回匹配结束的位置。

span()：返回一个元组(匹配开始的位置,匹配结束的位置)。

group (m, n)：返回组号 m、n 所匹配的字符串组成的元组,如果组号不存在,则返回 indexError 异常。

groups()：返回正则表达式中所有小组匹配到的字符串所组成的元组。

```
>>>import re
>>>print(re.search('www', 'www.baidu.com'))          #在起始位置匹配
<_sre.SRE_Match object; span=(0, 3), match='www'>
>>>print(re.search('www', 'www.baidu.com').span())
(0, 3)
>>>print(re.search('com', 'www.baidu.com'))          #不在起始位置匹配
<_sre.SRE_Match object; span=(10, 13), match='com'>
>>>print(re.search('com', 'www.baidu.com').end())    #返回匹配结束的位置
13
>>>print(re.search('com', 'www.baidu.com').start()) #返回匹配开始的位置
10
>>>str1="abc123def"
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).group())
                                                    #返回 abc123def 整体
abc123def
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).group(1))
                                                    #列出第一个括号匹配部分
abc
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).group(2))
                                                    #列出第二个括号匹配部分
123
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).group(3))
                                                    #列出第三个括号匹配部分
def
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).group(1,3))
                                                  #列出第一、第三个括号匹配部分
('abc', 'def')
>>>print(re.search("([a-z]*)([0-9]*)([a-z]*)",str1).groups())
('abc', '123', 'def')
```

2. re.match()函数

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,re.match ()就返回 None。

```
>>>import re
>>>print(re.match('www', 'www.baidu.com'))          #在起始位置匹配
<_sre.SRE_Match object; span=(0, 3), match='www'>
```