第5章 chapter 5

正则表达式

正则表达式描述了一种匹配字符串的模式，可以用来检查一个字符串是否含有正则表达式描述的字符串。本章主要介绍：正则表达式的构成，正则表达式的分组匹配，正则表达式的选择匹配，正则表达式的贪婪匹配与懒惰匹配，正则表达式模块 re，正则表达式中的(?:pattern)、(?=pattern)、(?!pattern)、(?<=pattern)和(?<!pattern)。


正则表达式的构成

5.1 正则表达式的构成

正则表达式(Regular Expression，通常缩写为 Regex 或 RegExp)是一种用于文本匹配和模式搜索的强大工具。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 到 9 的十个数字集[0-9]，用于匹配数字)以及元字符(例如 * 表示匹配位于 * 之前的字符或子表达式 0 次或多次出现)组成的字符串，该字符串描述一个可以识别某些字符串的模式(pattern)，也称为模板。这些模式可以用于如下方面。

(1) 搜索文本。可以使用正则表达式搜索文本中是否包含特定的字符串、子字符串或模式。

(2) 替换文本。可以使用正则表达式来查找并替换文本中的特定模式。

(3) 验证文本。正则表达式可用于验证文本是否符合特定格式或规则，如电子邮件地址、电话号码或日期。

(4) 数据提取。正则表达式可以从文本中提取特定的信息，如从 HTML 中提取链接、从日志文件中提取关键信息等。

若正则表达式是“Python 3”，该正则表达式仅仅由普通字符组成，没有使用任何预定义字符以及元字符，因此，该正则表达式只能匹配所描述的内容，即能够匹配这个正则表达式的只有“Python 3”字符串。正则表达式的强大之处在于引入预定义字符和元字符来定义正则表达式，使得正则表达式可以匹配众多不同的字符串，而不仅仅只是某一个字符串。

5.1.1 预定义字符

正则表达式中的预定义字符是一组用于匹配常见字符类型的特殊字符。这些预定

义字符类非常有用，因为它们可以简化正则表达式的编写，使我们能够更轻松地匹配数字、字母、空白字符等。一些用反斜杠字符(\)开始的字符表示预定义字符，表 5-1 列出了在正则表达式中常用的预定义字符。

表 5-1 正则表达式中常用的预定义字符

预定义字符	功 能
\d	匹配任一数字字符，相当于 0 到 9 的十个数字集[0-9]
\D	匹配任一非数字字符，相当于[^0-9]
\w	匹配单词字符(即字母、数字、下画线)中任意一个字符，相当于[a-zA-Z0-9_]
\W	匹配任一非单词字符，相当于 [^a-zA-Z0-9_]
\s	匹配任一空白字符，包括空格、换页符\f、换行符\n、回车符\r 等，相当于[\f\n\r\t\v]
\S	匹配任一非空白字符，相当于[^ \f\n\r\t\v]
\A	匹配字符串开始位置，忽略多行模式
\Z	匹配字符串结束位置，忽略多行模式
\b	表示单词字符与非单词字符的边界，不匹配任何实际字符，在正则表达式中使用\b 时需在其前面加\
\B	表示单词字符与单词字符的边界，非单词字符与非单词字符的边界

在 Python 中，是通过 re 模块让正则表达式发挥处理功能的。导入 re 模块后，可使用其中的 findall()函数在字符串中查找满足模式的所有子字符串。

```
re.findall(pattern, string[, flags])
```

函数功能：扫描整个字符串 string，并返回 string 中所有与模式(pattern)匹配的子字符串，并把它们作为一个列表返回。

参数说明如下。

pattern：模式，一个正则表达式。

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。

```
>>> import re
>>> string="hello2worldhello3worldhello4"
#获取 string 中所有"hello"后跟一个数字的子字符串
>>> re.findall("hello\d", string)
['hello2', 'hello3', 'hello4']
>>> re.findall("\Ahello\d", string)     #在字符串开始位置查找
['hello2']
>>> re.findall("hello\d\Z", string)     #在字符串结束位置查找
['hello4']
```

5.1.2 元字符

元字符就是一些有特殊含义的字符。若要匹配元字符，必须对元字符“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
\	转义字符,将其后字符标记为特殊字符、或原义字符、或向后引用等。例如,'\n' 匹配换行符,'\\' 匹配 "\"
.	匹配任一字符,除了换行符(\n),要匹配'.',需使用'\.'
^...	在字符串开头匹配...
... $	在字符串结尾匹配...
(...)	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
*	匹配位于 * 之前的字符或子表达式 0 次或多次
+	匹配位于+之前的字符或子表达式 1 次或多次
?	匹配位于? 之前的字符或子表达式 0 次或 1 次
{m}	匹配{m}之前的字符或子表达式 m 次
{m,n}	匹配{m,n}之前的字符或子表达式 m 至 n 次,m 和 n 可以省略,若省略 m,则匹配 0 至 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[...]中的任意一个字符,[^abc]表示匹配除了 a、b、c 之外的任一字符
|	匹配位于|之前或之后的字符或子表达式

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的字面值来匹配字符串,这和在 Word 等字符处理程序中使用关键字查找类似。

```
>>> import re
>>> re.findall("java", "javacjava")     #获取"javacjava"中所有的字符串"java"
['java', 'java']
```

2. 匹配数字

预定义的字符'\d'用于匹配任一数字,也可用字符组'[0-9]'替代'\d'来匹配任一数字。

```
>>> re.findall("\d", "12java34java56")                    #匹配所有的数字
['1', '2', '3', '4', '5', '6']
>>> re.findall("\\b\d{3}\\b", " 123 a * 456#b789")        #匹配 3 位数字
['123', '456']
>>> re.findall("\\b\d{3,}\\b", " 123 a * 4567#b7890 * ")  #匹配至少 3 位的数字
['123', '4567']
#匹配非零开头的最多带两位小数的数字
>>> re.findall("[1-9][0-9] * \.\d{1,2}\\b", " 1.23 a * 45.6#b7.892 * ")
```

```
['1.23', '45.6']
#匹配正数、负数和小数
>>> re.findall("-?\d+\.?\d+", "1.23@0.7g1897f-1.32")
['1.23', '0.7', '1897', '-1.32']
```

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，与'[^0-9]'与'[^\d]'的作用相同。

```
>>> re.findall("\D", "1java2java3")
['j', 'a', 'v', 'a', 'j', 'a', 'v', 'a']
#匹配汉字
>>> re.findall("[\u4e00-\u9fa5]+", "凡事总需研究，才会明白。")
['凡事总需研究', '才会明白']
```

4. 匹配单词和非单词字符

预定义字符'\w'用于匹配单词字符，用'[a-zA-Z0-9_]'可达到同样的效果。预定义字符'\W'用于匹配非单词字符。'\W'与'[^a-zA-Z0-9_]'的作用一样。

'a\we'可以匹配'afe'、'a3e'、'a_e'。

'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'。

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。

'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，比如用'.'查找'.'，就会出现问题，因为它们会被解释成特殊含义。这时我们就得使用'\'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。

7. 正则表达式的边界匹配

匹配字符串的起始位置要使用字符'^'，匹配字符串的结尾位置要使用字符'$'。

```
>>> import re
#匹配字符串的起始位置为 Ea 的一句话
>>> re.findall('^Ea[a-zA-Z]*\.',"Each of us holds a unique place in the
world. You are special, no matter what others say or what you may think. So
forget about being replaced. You can't be.")
['Each of us holds a unique place in the world.']
#匹配字符串的起始位置为 Ea 的字符串
>>> re.findall('^Ea.*\.$',"Each of us holds a unique place in the world. You
are special, no matter what others say or what you may think. So forget about
being replaced. You can't be.")
["Each of us holds a unique place in the world. You are special, no matter what
others say or what you may think. So forget about being replaced. You can'
t be."]
```

匹配单词边界要使用'\b'，如正则表达式'\bWe\b'匹配单词 We，而当它是其他单词的一部分的时候不匹配。

```
>>> re.findall('\\bWe\\b',"We Week Weekend.")
['We']
```

可以使用'\B'匹配非单词边界，'\B'表示单词字符与单词字符的边界，非单词字符与非单词字符的边界。

```
>>> re.findall('\\B\d*\d\\B',"#W12345e#")
['12345']
```

正则表达式的分组匹配

5.2 正则表达式的分组匹配

在前面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}等重复操作符就行了。但如果想要重复一个字符串，则需要使用圆括号来指定子表达式(也叫作分组或子模式)，然后就可以通过在圆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了。如'(abc){2}'可以匹配'abcabc'，{2}表示匹配{2}之前的表达式(abc)两次。

在正则表达式中，分组就是用一对圆括号“()”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1个分组，遇到第二个左括号“(”表示该正则表达式的第2个分组，以次类推。需要注意的是，有一个隐含的全局分组(即0分组)，就是整个正则表达式匹配的结果。可以使用 Match 对象的 group(num)方法获取正则表达式中分组号为 num 的分组匹配的内容，这是因为分组匹配到的内容会被临时存储到内存中，所以能够在需要的

时候被提取。

5.2.1 无名分组匹配

正则表达式基本分组匹配指的是把正则表达式中括号内的正则表达式作为一个分组，系统自动分配组号，可以通过分组号引用该分组匹配的内容。

【例 5-1】 无名分组匹配举例 1。

```
>>> import re
>>> text = "My email addresses are cjjiecao@qq.com and cjjiecao@163.com"
>>> pattern = "(\w+@\w+\.\w+)"
>>> matches = re.findall(pattern, text)
>>> text = "My email addresses are cjjiecao@qq.com and cjjiecao@163.com"
>>> pattern = "(\w+@\w+\.\w+)"
>>> matches = re.findall(pattern, text)
>>> for match in matches:
    print("Email:", match)
```

执行上述 for 循环，得到的输出结果如下。

```
Email: cjjiecao@qq.com
Email: cjjiecao@163.com
```

【例 5-2】 无名分组匹配举例 2。

```
>>> text = "现在是北京时间 2023 年 12 点 10 分"
>>> pattern = '\D*(\d{1,4})\D*(\d{1,2})\D*(\d{1,2})\D*'
>>> m =re.match(pattern,text)
>>> m
<re.Match object; span=(0, 18), match='现在是北京时间 2023 年 12 点 10 分'>
>>> print(m.group(1))                    #提取分组 1 的内容
2023
>>> print(m.group(2))                    #提取分组 2 的内容
12
>>> print(m.group(3))                    #提取分组 3 的内容
10
>>> print(m.group(0))                    #提取分组 0 的内容
现在是北京时间 2023 年 12 点 10 分
```

5.2.2 命名分组匹配

按照正则表达式进行匹配后，就可以通过分组提取到想要的内容，但是如果正则表达式中括号比较多，在提取想要的内容时，就需要挨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。这个时候 Python 又引入了另一种分组，也就是命名分组，前面的叫无名分组。

命名分组的语法格式如下。

```
(?P<name>正则表达式)        #name 是用户给分组命名的名字，一个合法的标识符
```

re.search(pattern, string[, flags])方法用于扫描整个字符串 string,找到与样式 pattern 相匹配的第一个字符串的位置,返回一个相应的 Match 对象。如果没有匹配的内容,则返回 None。

【例 5-3】 命名分组匹配举例。

```
import re
text = "My email address is cjjiecao@163.com"
pattern = r"(?P<name>\w+)@(?P<域名>\w+\.\w+)"
match = re.search(pattern, text)
if match:
    username = match.group("name")
    domain = match.group("域名")
    print("用户名:", username)
    print("域名  :", domain)
```

运行上述程序代码,得到的输出结果如下。

```
用户名: cjjiecao
域名  : 163.com
```

5.2.3 分组后向引用匹配

正则表达式中,用圆括号“()”括起来的内容表示一个组。当用“()”定义了一个正则表达式分组后,正则引擎就会把被匹配到的组按照顺序编号,然后存入缓存中。这样就可以在正则表达式的后面引用前面分组已经匹配出的内容,这就叫分组后向引用匹配。

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。而'\0'则引用整个正则表达式匹配出的内容。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 5-4】 分组引用匹配举例 1。

```
>>> import re
>>> re.search('(?P<name>\w+)\s+(?P=name)\s+(?P=name)', 'python python
python').group(1)
'python'
>>> re.search('(?P<name>\w+)\s+(?P=name)\s+(?P=name)', 'python python
python').group(0)
'python python python'
>>> s = 'Python.Java'
>>> re.sub(r'(.*)\.(.*)', r'\2.\1', s)
'Java.Python'
```

【例 5-5】 分组引用匹配举例 2。

```
import re
text = "abcabc"
pattern = r"(a\wc)(\1)"
result = re.search(pattern, text)
print(result.group(0))          #输出: abcabc
print(result.group(1))          #输出: abc
print(result.group(2))          #输出: abc
```

运行上述代码，得到的输出结果如下。

```
abcabc
abc
abc
```

5.3 正则表达式的选择匹配


正则表达式的选择匹配

选择匹配根据可以选择的情况有二选一或多选一，这涉及“()”和“|”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)
```

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，上述模式的另一个可选的模式是：

```
"(?i)Doctor|Dr\.?"
```

再如，带选择操作的模式(he|He)可以简写成(?i)he。

```
>>> import re
>>> re.findall("(?i)he","When the fox first saw1 the lion he was2 terribly3
frightened4. He ran5 away, and hid6 himself7 in the woods.")
['he', 'he', 'he', 'he', 'He', 'he']
>>> re.findall("(?i)Doctor|Dr\.?","Doctor doctor Dr. Dr")
['Doctor', 'doctor', 'Dr.', 'Dr']
```

5.4 正则表达式的贪婪匹配与懒惰匹配

正则表达式中的贪婪匹配和懒惰匹配是指匹配模式的两种不同行为，它们决定了正则表达式在匹配字符时是尽可能多地匹配字符(贪婪匹配)，还是尽可能少地匹配字符(懒惰匹配)。

5.4.1 贪婪匹配

默认情况下，正则表达式是贪婪匹配的，这意味着它会尽可能多地匹配符合条件的字符，即正则表达式中包含重复的限定符时，通常的行为是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。

```
>>> import re
>>> text = "Hope for the best, but prepare for the worst."
>>> pattern = ".*st"
>>> re.findall(pattern, text)
['Hope for the best, but prepare for the worst']
```

在这个示例中,正则表达式".*st"匹配以 "st" 结尾的字符串。但由于是贪婪匹配,它会匹配整个字符串中的最长匹配,即 "Hope for the best,but prepare for the worst"。

5.4.2 懒惰匹配

有时我们也需要懒惰匹配,也就是匹配尽可能少的字符。前面给出的重复限定符都可以被转化为懒惰限定符,称为懒惰匹配模式,只需在这些限定符后面加上一个问号"?"即可。例如'a.*?b'是匹配最短的以 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 1 到第 3 个字符)和 ab(第 4 到第 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,它比懒惰、贪婪规则的优先级更高,即"最先开始的匹配拥有最高的优先权"。表 5-3 列出了常用的懒惰限定符。

表 5-3 常用的懒惰限定符

懒惰限定符	描　　述
*?	重复任意次,但尽可能少地重复
+?	重复 1 次或更多次,但尽可能少地重复
??	重复 0 次或 1 次,但尽可能少地重复
{m,n}?	重复 m 到 n 次,但尽可能少地重复
{m,}?	重复 m 次以上,但尽可能少地重复

```
>>> import re
>>> text = "Hope for the best, but prepare for the worst."
>>> pattern = ".*?st"
>>> re.findall(pattern, text)
['Hope for the best', ', but prepare for the worst']
```

在这个示例中,正则表达式 ".*?st" 匹配以 "st" 结尾的字符串,但由于是懒惰匹配,它会尽可能少地匹配字符,因此会输出两个匹配,分别是 "Hope for the best"和 ", but prepare for the worst"。

再看一个懒惰匹配的示例。

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

5.5 正则表达式模块 re

Python 的 re 模块提供了对正则表达式的支持，表 5-4 列出了 re 模块中常用的函数。

表 5-4 re 模块中常用的函数

函 数	描 述
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。如果没有找到匹配项，则返回空列表
re.search(pattern,string[,flags])	扫描整个字符串 string，找到与样式 pattern 相匹配的第一个字符串的位置，返回一个相应的 Match 对象。如果没有匹配，则返回 None
re.match(pattern,string[,flags])	从字符串 string 的起始位置匹配模式 pattern，如果 string 的开始位置能够找到这个模式 pattern 的匹配，返回一个相应的匹配对象。如果不匹配，则返回 None
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.compile(pattern[,flags])	把正则表达式 pattern 转化为正则表达式对象
re.split(pattern,string[,maxsplit=0,flags])	用匹配 pattern 的子字符串来分割 string，并返回一个列表
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字符的字符串

函数参数说明。

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(忽略大小写)、re.M(多行匹配模式，改变‘^’和‘$’的行为)、re.S(使元字符“.”匹配任意字符，包括换行符)、re.X(忽略模式中的空格和#后面的注释，这个模式下正则表达式可以是多行并可以加入注释)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。

5.5.1 search()与 match()函数匹配字符串

1. search()函数

re.search(pattern,string[,flags])函数会在字符串 string 内查找与正则表达式 pattern 相匹配的字符串，只要找到第一个和该正则表达式相匹配的字符串就立即返回一个 Match 对象，Match 对象中包括匹配的字符串以及匹配的字符串在 string 中所处的位置。如果没有匹配的字符串，则返回 None。