Python 正则表达式

1 正则表达式

字符是计算机软件处理文字时最基本的单位，可能是字母，数字，标点符号，空格，换行符，汉字等等。字符串是 0 个或更多个字符的序列，因此，文本也是字符串一种形式。在实际工作中，我们往往会对大段的文字进行复杂地查找、替换等工作，这时候我们一般会用到正则表达式。如果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）能满足表达式给出的条件。

在从事科研工作的时候，有时候需要从互联网上抓取相关数据，当数据量较小的时候，一般可以通过“复制和粘贴”的手工操作完成；当数据量较大时候，手工操作一般无法完成，利用计算机编程往往是从网页上抠取信息的有效手段。这时候，我们需要处理大量的字符串，为了从中搜索出我们需要的信息，我们要设计一些复杂规则来过滤出我们需要的字符串。而正则表达式就是用于描述这些复杂规则的工具，它实际上是记录文本规则的代码。

1.1 何为正则表达式？

大家应该都使用过 word 中查找功能，如果你需要查找“节操”二字，仅需在查找窗口中输入“节操”并点查找按钮，就可以实现对文件内容的搜索。如果你需要查找某个目录下文件名均包含“节操”二字的文件，就必须要用到通配符（*和？），我们只要简单地搜索“*节操*”就可以找到相应的文件。在这里，*是百搭，被解释成任意字符串。**与此类似，正则表达式也是用来进行文本匹配的工具，而且它能更精确地描述我们的需求。**

1.2 正则表达式的入门

例 1：现有一本英文小说 Game of Throne，请找出小说中所有的单词 hi。

首先考虑最简单的**正则表达式“hi”**（类似于 Word 中查找），它可以精确匹配这样的字符串：**由两个字符组成，前一个字符是 h，后一个是 i**。若忽略大小写，它可以匹配 hi, HI, Hi, hI 这四种情况中的任意一种。

不幸的是，很多单词里包含 hi 这两个连续的字符，比如 him, history, high 等等。用**正则表达式“hi”**来查找的话，包含 hi 的单词均会被找出来。因此，要精确地查找 hi 这个单词的话，我们应该使用**正则表达式“\bhi\b”**。其中，**“\b”**是正则表达式规定的一个特殊代码，叫做**元字符，匹配单词的开头或结尾，也就是单词的分界处**。虽然英文的单词通常由空格，标点符号或者换行符来分

隔的，但是**“\b”**并不匹配这些分隔字符（空格、标点符号、换行符）中的任何一个，它**只匹配一个位置**。

例 2：若我们需要找的 hi 同时满足其后不远处还跟着单词 Lucy。

满足该规则的正则表达式应该写为**“\bhi\b.*\bLucy\b”**。其中，**“.”是元字符，匹配除了换行符以外的任意字符**。同样，**“*”也是元字符，不过它匹配的不是字符，也不是位置，而是数量，表示“*”前边的内容可以连续重复使用任意次以使整个表达式得到匹配。**因此，**“.*”连在一起就意味着任意数量的不包含换行的字符。**现在**“\bhi\b.*\bLucy\b”**的意思是：先是一个单词 hi，然后是任意个字符（除换行符），最后是 Lucy 这个单词。

例 3：需要在某通讯录中找出所有的上海地区的固定电话号码。

我们知道上海的区号为 021，号码是八位数字，如 021-64253634。因此，正则表达式可写为：**“021-\d\d\d\d\d\d\d\d”**。这里的**“\d”**是元字符，表示**匹配一个数字（0, 1, 2, 3, 4, 5, 6, 7, 8, 9）**。**“-”**不是元字符，只匹配它本身（连字符）。为了避免重复，我们也可以这样写这个表达式：**“021-\d{8}”**。**这里“\d”后面的“{8}”的意思是前面“\d”必须连续重复匹配 8 次**。

1.3 正则表达式的元字符

“\b”，“.”，“*”，“\d”是我们已经知道的元字符。正则表达式里还有其他常用的一些元字符，如下表所示。

表 1.常用的元字符代码说明

.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意的空白符
\d	匹配数字
\b	通常匹配是单词分界位置，但如果在字符类里使用代表退格
\a	匹配报警字符（打印它的效果是电脑嘀一声）
\t	匹配制表符，Tab
\r	匹配回车
\v	匹配竖向制表符
\f	匹配换页符
\n	匹配换行符
\e	匹配 Escape
^	匹配字符串的开始
$	匹配字符串的结束

例 4：正则表达式**“\ba\w*\b”表示匹配以字母“a”开头的单词**。该正则表达式

先匹配某个单词开始处（**“\b”**），然后是字母 a（**“a”**），然后是任意数量的字母或数字（**“\w*”**），最后是单词结束处（**“\b”**）。

例 5：某网站要求填写的 QQ 号必须为 5 位到 12 位数字，如何用正则表达式来检验输入的 QQ 号。

从表中可知，元字符**“^”**和**“$”**都匹配一个位置，这和**“\b”**有点类似。**“^”**匹配字符串的开头，**“$”**匹配字符串的结尾。则验证正则表达式为：**“^\d{5,12}$”**。这里的**“{5,12}”**和前面介绍过的**“{2}”**是类似的，只不过**“{2}”匹配只能重复 2 次，“{5,12}”匹配重复的次数不能少于 5 次，不能多于 12 次，否则都不匹配。**因为使用了**“^”**和**“$”**，所以输入的整个字符串都要和**“\d{5,12}”**来匹配，也就是说整个输入必须是 5 到 12 个数字，如果输入的 QQ 号能匹配这个正则表达式的话，那就符合要求了。

1.4 正则表达式的字符转义

如果你想查找字母 h，可以正则表达式可写成**“h”**表示。如果你想查找元字符本身的话（如 **“.”** 或者 **“*”** ），若在正则表达式中直接用 **“.”** 或者 **“*”**，就会出现问题，因此它们会被解释成别的意思。怎么破？使用转义符**“\”**来取消这些字符的特殊意义。因此，正则表达式应该写成**“\.”**或者**“*”**。由于**“\”**是转义符，所以要查找**“\”**本身，得用**“\\”**。

例 6：**“deerchao\.net”**匹配 **deerchao.net**；**“C:\\Windows”**匹配 **C:\Windows**。

1.5 正则表达式的重复

前面我们已经接触过几个重复匹配的方式了，如：**“*”**，**“+”**，**“{2}”**，**“{5,12}”**。下表是正则表达式中所有的限定符：

表 2. 常用的限定符

*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次

例 7：**“Windows\d+”**匹配 **Windows 后面跟 1 个或更多数字**；**“^\w+”**匹配**一行的第一个单词或整个字符串的第一个单词**；**“\d+”**匹配 **1 个或更多连续的数字**（注意：这里的**“+”**是和**“*”**类似的元字符，不同的是**“*”**匹配重复任意次(可以是 0 次)，而**“+”**匹配重复 1 次或更多次）；**“\b\w{6}\b”** 匹配 **6 个字符的单词**。

1.6 正则表达式查找字符

要想查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集合的元字符，但是如果你想匹配没有预定义元字符的字符集合（比如元音字母 a, e, i, o, u），怎么破？

So easy！你只需要在方括号里列出它们就行了，正则表达式**“[aeiou]”**就可以匹配**任何一个英文元音字母**，**“[.?!]”**匹配**标点符号（. 或 ？ 或 !）**。

举一反三，我们也可以轻松地指定一个字符范围，像**“[0-9]”**代表的含意与**“\d”**就是完全一致的，表示匹配**一位数字**；同理**“[a-z0-9A-Z_]”**也完全等同于**“\w”**（如果只考虑英文的话）。

例 8：正则表达式：**“\(?0\d{2}[) -]?\d{8}”**。该表达式可以匹配几种格式的电话号码，像(010)88886666，022-22334455，02912345678 等。我们对它进行一些分析吧：首先是一个转义字符**“\(?”**，表示 **(能出现 0 次或 1 次**；然后是一个 0；后面跟着 2 个数字（**“\d{2}”**）；**“[) -]?”**表示**“)”或“-”或“空格”中的一个，出现 1 次或不出现**；最后是 8 个数字(**“\d{8}”**)。

1.7 正则表达式的反义

有时需要查找不属于某个能简单定义的字符类的字符。比如想查找除了数字以外，其它任意字符都行的情况，这时需要用到**反义**：

表 3. 常用的反义代码

\W	匹配任意不是字母，数字，下划线，汉字的字符
\S	匹配任意不是空白符的字符
\D	匹配任意非数字的字符
\B	匹配不是单词开头或结束的位置
[^x]	匹配除了 x 以外的任意字符
[^aeiou]	匹配除了 aeiou 这几个字母以外的任意字符

例 9：**“\S+”**匹配**不包含空白符的字符串**；

“<a[^>]+>”匹配**用尖括号括起来的以 a 开头的不包含>的字符串**。

1.8 贪婪与懒惰

当正则表达式中包含能接受重复的限定符时，可以使整个表达式在得到匹配的前提下**尽可能匹配多**的字符。如：正则表达式**“a.*b”**，将匹配**最长的以 a 开始，以 b 结束的字符串**。如果用它来搜索 aabab 的话，将会匹配整个字符串 aabab，称为**贪婪**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。前面给出的限定符都可以被转化为懒惰匹配模式，只要在它后面加上一个问号**“?”**。这样**“.*?”**就意味着**匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复**。

例 10：**“a.*?b”**匹配最短的、以 a 开始、以 b 结束的字符串。如果把它应用于字符串 aabab 的话，它会匹配 aab（第一到第三个字符）和 ab（第四到第五个字符）。

表 5. 懒惰限定符

***?**	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 到 m 次，但尽可能少重复
{n,}?	重复 n 次以上，但尽可能少重复

综上所述，本部分概括了正则表达式的写法，即给出了一种识别并提取文字的模式（pattern），需要放到 Python 特定的函数中对文本（string）进行解析，才可以完成对指定模式文本的提取。下面对相关的 Python 函数进行系统性地介绍。

2 正则表达式在 Python 中的编程——re 模块函数

2.1 查找一个匹配项

查找并返回一个匹配项的函数有 3 个：**search、match、fullmatch**，他们的区别分别是：

re.search(pattern, string)——扫描整个字符串并返回第一个成功的匹配，即查找任意位置的匹配项。

re.match(pattern, string)——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的话，match()就返回 None，即必须从字符串开头匹配。

re.fullmatch(pattern, string)——整个字符串与正则完全匹配。

注意：查找 一个匹配项 返回的都是一个匹配对象（Match）。

例 11：查找一个匹配项（1）：

代码：

```
string = 'a 金融计算， 金融计算'
pattern = r'金融计算'
print('search:', re.search(pattern, string).group())
print('match:', re.match(pattern, string))
```

print('fullmatch:', re.fullmatch(pattern, string))

结果：

search: 金融计算

match: None

fullmatch: None

分析：**search 函数**是在字符串中任意位置匹配，只要有符合正则表达式的字符串就匹配成功，其实有两个匹配项，但 search 函数值返回一个。**match 函数**是要从头开始匹配，而字符串开头多了个字母 a，所以无法匹配，**fullmatch 函数**需要完全相同，故也不匹配！需注意，这里的**.group()**是将 **match 对象**转化为匹配值，具体将在 3.3 分组匹配中展开。

例 12：查找一个匹配项（2）：

代码：

string = '金融计算，金融计算'

pattern = r'金融计算'

print('search:', re.search(pattern, string).group())

print('match:', re.match(pattern, string).group())

print('fullmatch:', re.fullmatch(pattern, string))

结果：

search: 金融计算

match: 金融计算

fullmatch: None

分析：删除了 text 最开头的字母 a，这样 **match 函数**就可以进行匹配，而 **fullmatch 函数**依然不能完全匹配！

例 13：查找一个匹配项（3）：

代码：

string = '金融计算'

pattern = r'金融计算'

print('search:', re.search(pattern, string).group())

print('match:', re.match(pattern, string).group())

print('fullmatch:', re.fullmatch(pattern, string).group())

结果：

search: 金融计算

match: 金融计算

fullmatch: 金融计算

分析：只留下一段文字，并且与正则表达式一致；这时 **fullmatch 函数**终于可以匹配了。

2.2 查找多个匹配项

查找多项函数主要有：**re.findall 函数** 与 **re.finditer 函数**：

re.findall——在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回**空列表。**

re.finditer——在字符串中找到正则表达式所匹配的所有子串，并返回一个迭代器。

两个方法基本类似，只不过一个是返回列表，一个是返回迭代器。列表是一次性生成在内存中，而迭代器是需要使用时一点一点生成出来的，内存使用更优。如果可能存在大量的匹配项的话，建议使用 **re.finditer 函数**，一般情况使用 **re.findall 函数**基本没影响。

例 14：查找多个匹配项：

代码：

```
string = 'a 金融计算， 金融计算'
pattern = r'金融计算'
print('findall:', re.findall(pattern, string))
print('finditer:', list(re.finditer(pattern, string)))
```

结果：

```
findall: ['金融计算', '金融计算']
finditer: [<re.Match object; span=(1, 5), match='金融计算'>, <re.Match object; span=(6, 10), match='金融计算'>]
```

2.3 分割

re.split(pattern, string, maxsplit=0, flags=0) 函数：用 **pattern** 分开 **string** ，**maxsplit** 表示最多进行分割次数， **flags** 表示正则表达式的常量，如忽略大小写等，有关 **flags** 的具体阐述将在后文展开。

例 15：分割展示：

代码：

```
string = 'a 金融计算， b 金融计算， c 金融计算'
pattern = r'， '
print('split:', re.split(pattern, string, maxsplit=1, flags=re.I))
```

结果：

```
split: ['a 金融计算', 'b 金融计算， c 金融计算']
```

2.4 替换

替换主要有 **sub 函数**与 **subn 函数**，他们功能类似：

re.sub(pattern, repl, string, count=0, flags=0)：repl 替换掉 string 中被 pattern 匹配的字符，count 表示最大替换次数，flags 表示正则表达式的常量。（**repl 替换内容既可以是字符串，也可以是一个函数**）

re.subn(pattern, repl, string, count=0, flags=0)：与 re.sub 函数功能一致，只不过返回一个元组 (字符串, 替换次数)。

例 16：替换展示：

代码：

```
string = 'a 金融计算，b 金融计算，c 金融计算'
pattern = r'，'
repl = '、'
print('sub-repl 为字符串:', re.sub(pattern, repl, string, count=2, flags=re.I))
print('subn-repl 为字符串:', re.subn(pattern, repl, string, count=2, flags=re.I))
```

结果：

```
sub-repl 为字符串: a 金融计算、b 金融计算、c 金融计算
subn-repl 为字符串: ('a 金融计算、b 金融计算、c 金融计算', 2)
```

2.5 编译正则对象

re.compile 函数能将正则表达式的样式编译为一个**正则表达式对象**，这个对象与 re 模块有同样的正则函数，即可以使用**.match()、.search()、.findall()**等函数。官方文档推荐：**在多次使用某个正则表达式时推荐使用正则对象 Pattern** 以增加复用性，因为通过 **re.compile(pattern)**编译后的模块级函数会被缓存！下面举一个例子。

例 17：编译正则对象展示：

代码：

```
string = 'a 金融计算，金融计算'
pattern = r'金融计算'
pattern_obj = re.compile(pattern)
print('pattern_obj.search:', pattern_obj.search(string).group())
```

结果：

```
pattern_obj.search:  金融计算
```

2.6 re 模块常量

前文已出现过 **flags**，即正则表达式的常量，可以控制大小写、换行匹配等功能，本节将对常用的常量进行详细阐述。

re.IGNORECASE 或简写为 **re.I**——进行忽略大小写匹配。

re.DOTALL 或简写为 **re.S**——DOT 表示.，ALL 表示所有，连起来就是.匹配所有，包括换行符\n。默认模式下.是不能匹配行符\n 的。

re.MULTILINE 或简写为 **re.M**——多行模式，当某字符串中有换行符\n，默认模式下是不支持换行符特性的，比如：行开头和行结尾，而多行模式下是支持匹配行开头的。

re.VERBOS 或简写为 **re.X**——详细模式，可以在正则表达式中加注解。

例 18：re.I 展示：

代码：

```
string = '金融计算 a'
pattern = r'金融计算 A'
print('默认模式:', re.findall(pattern, string))
print('忽略大小写  模式:', re.findall(pattern, string, re.I))
```

结果：

```
默认模式: []
忽略大小写  模式: ['金融计算 a']
```

分析：

在默认匹配模式下**大写字母 B** 无法匹配**小写字母 b**，而在忽略大小写模式下是可以的。

例 19：re.S 展示：

代码：

```
string = '金融\n 计算'
pattern = r'.*'
print('默认模式:', re.findall(pattern, string))
print('.匹配所有模式:', re.findall(pattern, string, re.S))
```

结果：

```
默认模式: ['金融', '', '计算', '']
.匹配所有模式: ['金融\n 计算', '']
```

分析：

在默认匹配模式下.并没有匹配换行符\n，而是将字符串分开匹配；而在 **re.S** 模式下，换行符\n 与字符串一起被匹配到。注意：**默认匹配模式下.并不会匹配换行符\n。**

例 20：re.M 展示：

代码：

```
string = '金融\n 计算'
pattern = r'^计算'
print('默认模式:', re.findall(pattern, string))
print('多行模式:', re.findall(pattern, string, re.M))
```

结果：

默认模式: []

多行模式: ['计算']

分析：

正则表达式中^表示匹配行的开头，默认模式下它只能匹配字符串的开头；而在多行模式下，它还可以匹配换行符\n 后面的字符。注意：**正则语法中^匹配行开头、\A 匹配字符串开头，单行模式下他两效果一致，多行模式下\A 不能识别\n。**

例 21：re.X 展示：

代码：

```
string = '金融计算有意思'
pattern = r'''^金融计算    #注释 1
                有意思    #注释 2
            '''
print('默认模式:', re.findall(pattern, string))
print('详细模式:', re.findall(pattern, string, re.X))
```

结果：

默认模式: []

详细模式: ['金融计算有意思']

分析：

默认模式下并不能识别正则表达式中的注释，而详细模式是可以识别的。当一个正则表达式十分复杂的时候，详细模式或许能为你提供另一种注释方式，但它不应该成为炫技的手段，建议谨慎考虑后使用！

2.7 其他值得注意的地方：

r 的作用——正则表达式使用反斜杠（’\’）来表示特殊形式，或者把特殊字符转义成普通字符。而反斜杠在普通的 Python 字符串里也有相同的作用，所以就产生了冲突。解决办法是对于正则表达式样式使用 Python 的原始字符串表示法；在带有‘r’前缀的字符串字面值中，反斜杠不必做任何特殊处理。

正则查找函数返回匹配对象——查找一个匹配项（search、match、fullmatch）的函数返回值都是一个 **对象**，需要通过 **match.group()** 获取匹配值，这个很容易忘记。另外，如果要匹配相对应的起止位置，可用 **match.start()**返回开始位置，用 **match.end()**返回结束位置，或可使用 **match.span()**以 tuple 形式返回范围，具体例子将在后文展开。（请注意 re.findall 返回的是一个列表，不是 match 对象，因此以上方法不适用。）

一个实用的正则测试网站——https://regex101.com/

3 基于正则表达式和 re 库的文本提取

前两部分分开介绍了正则表达式与 re 库的用法，所举的例子几乎没有将二者结合起来。本部分将着重使用正则表达式写 pattern，并代入到 re 下的相应函数中，以期完成指定规则下的文本提取。

3.1 简单匹配

例 22：假设你在玩英文拼字游戏，想要找出三个字母的单词，而且这些单词必须以't'字母开头，以'n'字母结束；另外，有一本英文字典，你可以用正则表达式搜索它的全部内容。要构造出这个正则表达式，你可以使用一个通配符（**句点符号“.”**）。这样，完整的表达式就是**“t.n”**，它匹配 tan、ten、tin 和 ton，还匹配 t#n、tpn 甚至 t n，还有其他许多无意义的组合。

Python 程序实例：

```
string = 'ten,&8yn2tin6ui>&ton, t n,-356tpn,$$$$t#n,4@).,t@nT&nY'
pattern = 't.n'
% 输出 output_start，记录匹配正则表达式的字符串的起始位置。
    output_start = [x.span()[0] for x in re.finditer(pattern, string, re.I)]
    输出为：
    [0, 9, 17, 22, 30, 38, 47, 50]
%输出 output_end，记录匹配正则表达式的字符串的结束位置。
    output_end = [x.span()[1] for x in re.finditer(pattern, string, re.I)]
    输出为：
    [3, 12, 20, 25, 33, 41, 50, 53]
% 指定输出 ouput_match，记录匹配正则表达式的字符串。
    output_match = re.findall(pattern, string, re.I)
    输出为：
    ['ten', 'tin', 'ton', 't n', 'tpn', 't#n', 't@n', 'T&n']
```

例 23：现要求仅匹配上述字符串中的 ten, tin, ton, 则正则表达式为**“t[eio]n”**。其中，“[eio]”表示匹配方括号中的任意一个（注：若在方括号“[]”里面指定的字符才可以参与匹配，而且仅能匹配一个）也就是说，正则表达式 t[eio]n 只匹配 ten, tin, ton, 而其他字符一律不能匹配。

Python 程序实例：

```
string = 'ten,&8yn2tin6ui>&ton, t n,-356tpn,$$$$t#n,4@).,t@nT&nY'
pattern = 't[eio]n'
% 同时输出匹配的字符串及其起始位置
  output_start = [x.span()[0] for x in re.finditer(pattern, string, re.I)]
  print(output_start)
  output_end = [x.span()[1] for x in re.finditer(pattern, string, re.I)]
  print(output_end)
  output_match = re.findall(pattern, string, re.I)
  print(output_match)
```

输出为：

```
[0, 9, 17]
[3, 12, 20]
['ten', 'tin', 'ton']
```

例 24：**“[c1-c2]”**匹配从字符 c1 开始到字符 c2 结束的字母序列（按字母表中的顺序）中的任意一个。**“[a-c]”**匹配 a, b, c。正则表达式**“t[a-z]n”**匹配 tan, tbn, tcn, tdn, ten, …, txn, tyn, tzn。

Python 程序实例：

```
string = 'ten,&8yn2tin6ui>&ton, t n,-356tpn,$$$$t#n,4@).,t@nT&nY'
pattern = 't[a-z]n';
% 同时输出匹配的字符串及其起始位置
  output_start = [x.span()[0] for x in re.finditer(pattern, string, re.I)]
  print(output_start)
  output_end = [x.span()[1] for x in re.finditer(pattern, string, re.I)]
  print(output_end)
  output_match = re.findall(pattern, string, re.I)
  print(output_match)
```

输出为：

```
[0, 9, 17, 30]
[3, 12, 20, 33]
['ten', 'tin', 'ton', 'tpn']
```

例 25：**元字符的匹配。**某些在正则表达式中有语法功能或特殊意义的字符 c，要用“\c”来匹配，而不能直接用“c”匹配。例如：“.”用正则表达式“\.”匹配，而“\”用正则表达式 “\\”匹配。

Python 程序实例：

```
string = 'l.[a-c]i.'
pattern1 = '.'
pattern2 = '\.'
output_1 = re.findall(pattern1, string, re.I)
output_2 = re.findall(pattern2, string, re.I)
```

输出为：

['l', '.', '[', 'a', '-', 'c', ']', 'i', '.']

['.', '.']

例 26：**类表达式。**我们已经知道“\w”，“\s”，“\d”等用于匹配某一类字符中的一个。**和上面的“\n”等表中的转义字符有所不同，“\w”，“\s”，“\d”等匹配的不是某个特定的字符，而是某一类字符。**具体说明如下：

\w 匹配任意的单个文字字符，相当于[a-zA-Z0-9_];

\s 匹配任意的单个空白字符，相当于[\t\f\n\r];

\d 匹配任意单个数字，相当于[0-9];

\S 匹配除空白符以外的任意单个字符，相当于[^\t\f\n\r]，其中^表示取反;

\W 匹配任意单个字符，相当于[^a-zA-Z0-9_];

\D 匹配除数字字符外的任意单个字符，相当于 [^0-9]。

Python 程序实例：

```
string = 'This city has a population of more than 1,000,000.'
pattern = '\d'
output = re.findall(pattern, string)
```

输出为：

['1', '0', '0', '0', '0', '0', '0']

3.2 字符串的匹配

3.2.1 多次匹配

如果需要匹配 ppp，那么正则表达式可以写成“ppp”，还可以简单地写成“p{3}”。**正则表达式中用“{}”表示表达式匹配的次数。**如果正则表达式写成“p{2,3}”，则表示匹配“pp”和“ppp”。除“{}”之外，还有几个字符，用在表示单个字符的正则表达式后面表示次数，如下所述：

exp? 与 exp 匹配的元素出现 0 或 1 次，相当于{0,1}

exp*　　与 exp 匹配的元素出现 0 次或更多，相当于{0,}

exp+　　与 exp 匹配的元素出现 1 次或更多，相当于{1,}

exp{n}　　与 exp 匹配的元素出现 n 次，相当于{n,n}

exp{n,}　　与 exp 匹配的元素至少出现 n 次

exp{n,m}　　与 exp 匹配的元素出现 n 次但不多于 m 次

假设我们要在文本文件中搜索美国的社会安全号码。这个号码的格式是 999-99-9999。用来匹配它的正则表达式为**“[0-9]{3}\-[0-9]{2}\-[0-9]{4}”**。在正则表达式中，连字符（**“-”**）有着特殊的意义，因此，它的前面要加上一个转义字符**“\”**。如果希望连字符号可以出现，也可以不出现，即 999-99-9999 和 999999999 都属于正确的格式。那么可在连字符号后面加上**“?”**数量限定符，这时正则表达式为**“[0-9]{3}\-?[0-9]{2}\-?[0-9]{4}”**。**需要指出的是，当我们使用“exp*”时，Python 将尽可能的匹配最长的字符串。**

例 27：比较“exp*”，“exp*?”。

```
string = '<tr valign=top><td><a name="19184"></a>xyz'
pattern = '<.*>'
output = re.findall(pattern, string)
```

输出为：['<tr valign=top><td>']

如果希望匹配尽可能短的字符子串时，可以在**“.*”**后使用**“?”**，即**“exp*?”**如：

```
string = '<tr valign=top><td><a name="19184"></a>xyz'
pattern = '<.*?>'
output = re.findall(pattern, string)
```

输出为：['<tr valign=top>', '<td>', '', '']

3.2.2 一些固定的语法

“(exp)”将 exp 标为一组，匹配 exp。关于这部分内容下面还会有更详细介绍。

“(?:exp)”表示 exp 为一组，相当于数学表达式中的（）。

例 28：

```
string = 'A body or collection of such stories'
pattern = '(?:[^aeiou][aeiou]){2,}'
output = re.findall(pattern, string)
```

输出为：['tori']

分析：上面的表达式中{2,}对**“[^aeiou][aeiou]”**起作用，如果去掉分组，则只对**“[aeiou]”**起作用，如下所示：

```
pattern = '[^aeiou][aeiou]{2,}'
output = re.findall(pattern, string)
```

输出为：['tio', 'rie']

“(?>exp) ” 仅进行自动分组。

“exp1|exp2” 表示匹配 exp1 或匹配 exp2，两者满足之一。

pattern = '[^aeiou\s]o|[^aeiou\s]i'

output = re.findall(pattern, string)

输出为：['bo', 'co', 'ti', 'to', 'ri']

“(?#exp) ” 放在 **“(?#”** 和 **“)”** 之间的是注释，如下所示：

pattern = '(?# Match words in caps)[A-Z]\w*'

output = re.findall(pattern, string, re.I)

输出为：['A', 'body', 'or', 'collection', 'of', 'such', 'stories']

“^exp” 匹配 exp，并且出现在**原字符串最前端的子串**。

“exp$” 匹配 “exp”，并且出现在**原字符串最末端的子串**。

pattern = '^a\w*|\w*s$'

output = re.findall(pattern, string)

输出为：output = 'A' 'stories'

“\bexp” 匹配 exp，并且出现在**一个单词最前端的子串**。

pattern = r'\bs\w+'

output = re.findall(pattern, string)

输出为：['such', 'stories']

“exp\b” 匹配 exp，并且出现在**一个单词最末端的子串**。

pattern = r'\w*tion\b'

output = re.findall(pattern, string)

输出为：['collection']

“\bexp\b” 更严格的单词匹配，如：以 s 开头，并且以 h 结尾的单词。

pattern = r'\bs\w*h\b'

output = re.findall(pattern, string)

输出为：['such']

3.2.3 利用上下文匹配查找我们需要的内容

“exp1(?=exp2)” 找到匹配 exp1 的字符串，同时要求其后的字符串也匹配 exp2。

例 29：查找所有在','之前的字符串。

string = 'Grammar Of, relating to, or being a noun or pronoun case that indicates possession.'

pattern = '\w+(?=,)'

output = re.findall(pattern, string)

输出为：['Of', 'to']

“exp1(?!exp2)” 找到匹配 exp1 的字符串，同时保证其后的字符串不匹配 exp2。

例 30：查找所有不在', '之前的字符串。

pattern = '\w+(?!,)'

output = re.findall(pattern, string)

输出为：['Grammar', 'O', 'relating', 't', 'or', 'being', 'a', 'noun', 'or', 'pronoun', 'case', 'that', 'indicates', 'possession']

“(?<=exp1)exp2” 找到匹配 exp2 的字符串，同时要求其前面的字符串也匹配 exp1。

例 31：查找所有在','之后的字符串，注意','之后可能有空格。

pattern = '(?<=,\s{1})\w*'

output = re.findall(pattern, string)

输出为：['relating', 'or']

“(?<!exp1)exp2” 找到匹配 exp2 的字符串，同时要求其前的字符串不匹配 exp1。

例 32：查找所有不在','之后的字符串。

pattern = '(?<!,\s)\w+'

output = re.findall(pattern, string)

输出为：['Grammar', 'Of', 'elating', 'to', 'r', 'being', 'a', 'noun', 'or', 'pronoun', 'case', 'that', 'indicates', 'possession']

3.3 分组匹配

任何的正则表达式都可以用圆括号括起来作为一个分组，如**“(exp)”中 exp 匹配字符将作为一个分组（group）**。与括号内的正则表达式相匹配的字符串会被记录下来，根据圆括号出现的顺序，依次编号，并可以使用 **“\N”** 来引用第 N 个括号内匹配的字符串，如正则表达式**“(exp1)(exp2)(exp3)”**中与 exp1，exp2，exp3 匹配的字符就会被编号为 1，2，3，并可以用 **“\1”“\2”“\3”** 来分别引用与第一、二、三个括号内相匹配的字符串。如果有正则表达式“(exp1)(exp2)(exp3)(exp4)...(expN)”，每个括号将产生一个 group，并按照括号顺序编号，即 1, 2, 3, …, N。利用“\N”可以引用正则表达式中第 N 组的字符串，例如\1 引用第一个 group。如果参数值是 0,那么返回整个匹配结果的字符串。另一个返回所有匹配结果的方式是 groups，它将以元组的形式返回所有 group。需要注意只有 match 对象才有 group 或者 groups，findall 生成的是 list，不能直接使用 group。下面通过例子进行具体展开。

例 33：一个简单的分组匹配例子

m = re.match("(\w+) (\w+)", "金融计算 教师： 蒋老师")

print(m.group(0)) #group(0)就是匹配的整个结果

print(m.group(1)) #group(1)是第一个 group 的值

print(m.group(2)) #group(2)是第二个 group 的值

print(m.groups()) #groups 以元组的形式返回所有的 group

输出为：

金融计算 教师

金融计算

教师

('金融计算', '教师')

例 34：在文本中同一非空字符连续两次出现的情况，如“aa”，“bb”等。我们可以用**“（\S）”**查找任意的非空白字符，并作为匹配的第一个 group，再用**“\2”** 匹配分组 number 是 2 的分组，因为最外层有圆括号，所以 number 是 2 的分组就是前面\S 匹配的字符。

string = 'Grammar Of, relating to, or being a noun or pronoun case that indicates possession.'

pattern = r'((\S)\2)'

output = re.findall(pattern, string) # 匹配了元组形式的所有 group

output_0 = [i[0] for i in output] # 匹配了最外层圆括号

output_1 = [i[1] for i in output] # 匹配了内层圆括号

output_coordinates = [coordinate.span() for coordinate in re.finditer(pattern, string)]

print(output)

print(output_0)

print(output_1)

print(output_coordinates)

输出为：

[('mm', 'm'), ('ss', 's'), ('ss', 's')]

['mm', 'ss', 'ss']

['m', 's', 's']

[(3, 5), (74, 76), (77, 79)]

例 35：查找 html 语句中类似<a>abc的部分。

**string = '<!comment><tr nam="7507"></tr><table>Default</table>
'**

pattern = '<(\w+).*?>.*?<\1>'

output = re.findall(pattern, string)

print(output)

输出为：

['tr', 'table']

例 36：将匹配到的第一个 group 和第二个 group 的位置互换。

string = 'Norma Jean Baker'

pattern = '(\w+\s\w+)\s(\w+)'

output = re.sub(pattern, lambda x: x[2] + ' ' + x[1], string)

print(output)

输出为：Baker Norma Jean

分析：可尝试输出 lambda 中的 x[0]，看看 x[0]匹配了什么。

4 综合应用实例

问题 1：查找包含某个字串的串。要在字符串中 apple_food, chocolates_food, ipod_electronics, dvd_player_electronics, water_melon_food 中检查是否包含子字符串 food。

string=['apple_food','chocolates_food','ipod_electronics','dvd_player_electronics', 'water_melon_food']

pattern = 'food'

output = [(re.search(pattern, single_string) is not None) for single_string in string]

问题 2：如何将 Python 中的^转换成 C 语言？如将 a^b 转换成 a**b，或者 power(a, b)。已知计算公式为：

1/2*w/(1+Pf^2*Pc-Pf^2*Pc*w1-w1*Pf^2-Pf*Pc-Pf^2*w^2+2*w1*Pf-2*Pf)

string='1/2*w/(1+Pf^2*Pc-Pf^2*Pc*w1-w1*Pf^2-Pf*Pc-Pf^2*w^2+2*w1*Pf-2*Pf)'

pattern = '(\w{1,2})\^(\d{1})'

temp = re.findall(pattern, string, re.I)

output = re.sub(pattern, lambda x: 'power(' + x[1] + ',' + x[2] + ')', string)

问题 3：删掉<，/>及两符号之间的部分。

处理前：Hello world. 2 < 5

处理后：Hello world. 2 < 5

string = 'Hello world. 2 < 5'

pattern ='<.*?>'

string_replace = re.sub(pattern, '', string)

问题 4：游程平滑算法：将连续的且个数小于某个阈值的 0 全部替换成 1。

平滑前：1111100000111100011

平滑后：1111100000111111111

a = '1111100000111100011'

T = 4

b = re.sub(r'(?<!0)0{1,' + str(T - 1) + r'}(?!0)', lambda x: '1' * len(x[0]), a)

5. 网页数据抓取案例

科创板上市公司董事、高管、监事数据抓取

提示：从 http://guba.eastmoney.com/remenba.aspx?type=1&tab=7 获取科创板所有上市公司的名单与股票代码，构造 URL（如：http://f10.eastmoney.com/CompanyManagement/CompanyManagementAjax?code=SH688003）并进行访问。

代码如下：

```
import requests
import re
import time
import random
import pandas as pd
from tqdm import tqdm
from bs4 import BeautifulSoup
import json

def get_company_list(df_name):
    '''
    输入你的 xlsx 文件名，爬取科创板企业的代码与名称并保存到本地
    '''
    company_df = pd.DataFrame()
    URL = 'http://guba.eastmoney.com/remenba.aspx?type=1&tab=7'
    response = requests.get(URL, timeout=10)

    # bs
    soup = BeautifulSoup(response.content, 'lxml')
    data = soup.select('.ngblistul2 li')
```

```
        company_df['code'] = [re.search('\d{6}', i.text).group() for i in data]
        company_df['name'] = [i.text[8:] for i in data]
        company_df.to_excel(df_name, index=False)

    if __name__ == '__main__':
        df_name = '科创板上市公司名单.xlsx'
        get_company_list(df_name)
        company_df = pd.read_excel(df_name)
        company_code_list = company_df['code']
        company_name_list = company_df['name']
        manager_info_data = pd.DataFrame()
        for i in tqdm(range(len(company_code_list))):
            code = company_code_list[i]
            name = company_name_list[i]
            manager_info = pd.DataFrame()
            while 1:
                error_num = 0
                try:
                    URL = 'http://f10.eastmoney.com/CompanyManagement/CompanyManagementAjax?code=SH' + str(code)
                    response = requests.get(URL, timeout=10)
                    string = response.text
                    #  正则
                    manager_info['姓名'] = re.findall('"xm":"(.*?)"', string)
                    manager_info['性别'] = re.findall('"xb":"(.*?)"', string)
                    manager_info['年龄'] = re.findall('"nl":"(.*?)"', string)
                    manager_info['学历'] = re.findall('"xl":"(.*?)"', string)
                    manager_info['职位'] = re.findall('"zw":"(.*?)"', string)
                    manager_info['入职时间'] = re.findall('"rzsj":"(.*?)"', string)
                    manager_info['简介'] = re.findall('"jj":"(.*?)"', string)
                    # # json
                    # manager_info = pd.DataFrame(json.loads(response.text)['RptManagerList'])
                    # manager_info.rename(
```

```
                        #         columns={'xm': '姓名', 'xb': '性别', 'nl': '年龄', 'xl': '学
历', 'zw': '职位',
                        #                         'rzsj': '入职时间', 'jj': '简介'}, inplace=True)
                        manager_info['股票代码'] = code
                        manager_info['公司名称'] = name
                        manager_info_data                                                    =
manager_info_data.append(manager_info, ignore_index=True)
                        # time.sleep(random.randint(1, 4))
                        break
                    except Exception as e:
                        print(e)
                        error_num += 1
                        if error_num == 3:
                            print(str(code) + str(name) + '爬取有误')
                        break
        manager_info_data.to_excel('科创板高管信息.xlsx', index=False)
```