作为一个概念而言，正则表达式对于Python来说并不是独有的。但是，Python中的正则表达式在实际使用过程中还是有一些细小的差别。

本文是一系列关于Python正则表达式文章的其中一部分。在这个系列的第一篇文章中，我们将重点讨论如何使用Python中的正则表达式并突出Python中一些独有的特性。

我们将介绍Python中对字符串进行搜索和查找的一些方法。然后我们讲讨论如何使用分组来处理我们查找到的匹配对象的子项。

我们有兴趣使用的Python中正则表达式的模块通常叫做‘re’。

Python

|  |  |
| --- | --- |
| 1 | >>> import re |

**1. Python中的原始类型字符串**

Python编译器用‘\’（反斜杠）来表示字符串常量中的转义字符。

如果反斜杠后面跟着一串编译器能够识别的特殊字符，那么整个转义序列将被替换成对应的特殊字符（例如，‘\n’将被编译器替换成换行符）。

但这给在Python中使用正则表达式带来了一个问题，因为在‘re’模块中也使用反斜杠来转义正则表达式中的特殊字符（比如\*和+）。

这两种方式的混合意味着有时候你不得不转义转义字符本身（当特殊字符能同时被Python和正则表达式的编译器识别的时候），但在其他时候你不必这么做（如果特殊字符只能被Python编译器识别）。

与其将我们的心思放在去弄懂到底需要多少个反斜杠，我们可以使用原始字符串来替代。

原始类型字符串可以简单的通过在普通字符串的双引号前面加一个字符‘r’来创建。当一个字符串是原始类型时，Python编译器不会对其尝试做任何的替换。本质上来讲，你在告诉编译器完全不要去干涉你的字符串。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> string = 'This is a\nnormal string'  >>> rawString = r'and this is a\nraw string'  >>> print string  这是一个普通字符串  >>> print rawString  and this is a\nraw string  这是一个原始类型字符串。 |

**在Python中使用正则表达式进行查找**

‘re’模块提供了几个方法对输入的字符串进行确切的查询。我们将会要讨论的方法有：

* re.match()
* re.search()
* re.findall()

每一个方法都接收一个正则表达式和一个待查找匹配的字符串。让我们更详细的查看这每一个方法从而弄明白他们是如何工作的以及他们各有什么不同。

**2. 使用re.match查找 – 匹配开始**

让我们先来看一下match()方法。match()方法的工作方式是只有当被搜索字符串的开头匹配模式的时候它才能查找到匹配对象。

举个例子，对字符串‘dog cat dog’调用mathch()方法，查找模式‘dog’将会匹配：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> re.match(r'dog', 'dog cat dog')  <\_sre.SRE\_Match object at 0xb743e720<  >>> match = re.match(r'dog', 'dog cat dog')  >>> match.group(0)  'dog' |

我们稍后将更多的讨论group()方法。现在，我们只需要知道我们用0作为它的参数调用了它，group()方法返回查找到的匹配的模式。

我还暂且略过了返回的SRE\_Match对象，我们很快也将会讨论到它。

但是，如果我们对同一个字符串调用match()方法，查找模式‘cat’，则不会找到匹配。

Python

|  |  |
| --- | --- |
| 1  2 | >>> re.match(r'cat', 'dog cat dog')  >>> |

**3. 使用re.search查找 – 匹配任意位置**

search()方法和match()类似，不过search()方法不会限制我们只从字符串的开头查找匹配，因此在我们的示例字符串中查找‘cat’会查找到一个匹配：

Python

|  |  |
| --- | --- |
| 1  2  3 | search(r'cat', 'dog cat dog')  >>> match.group(0)  'cat' |

然而search()方法会在它查找到一个匹配项之后停止继续查找，因此在我们的示例字符串中用match()方法查找‘dog’只找到其首次出现的位置。

Python

|  |  |
| --- | --- |
| 1  2  3 | >>> match = re.search(r'dog', 'dog cat dog')  >>> match.group(0)  'dog' |

**4. 使用 re.findall – 所有匹配对象**

目前为止在Python中我使用的最多的查找方法是findall()方法。当我们调用findall()方法，我们可以非常简单的得到一个所有匹配模式的列表，而不是得到match的对象（我们会在接下来更多的讨论match对象）。对我而言这更加简单。对示例字符串调用findall()方法我们得到：

Python

|  |  |
| --- | --- |
| 1  2  3 | ['dog', 'dog']  >>> re.findall(r'cat', 'dog cat dog')  ['cat'] |

**5. 使用 match.start 和 match.end 方法**

那么，先前search()和match()方法先前返回给我们的‘match’对象”到底是什么呢？

和只简单的返回字符串的匹配部分不同，search()和match()返回的“匹配对象”，实际上是一个关于匹配子串的包装类。

先前你看到我可以通过调用group()方法得到匹配的子串，（我们将在下一个部分看到，事实上匹配对象在处理分组问题时非常有用），但是匹配对象还包含了更多关于匹配子串的信息。

例如，match对象可以告诉我们匹配的内容在原始字符串中的开始和结束位置：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> match = re.search(r'dog', 'dog cat dog')  >>> match.start()  0  >>> match.end()  3 |

知道这些信息有时候非常有用。

**6. 使用 mathch.group 通过数字分组**

就像我之前提到的，匹配对象在处理分组时非常得心应手。

分组是对整个正则表达式的特定子串进行定位的能力。我们可以定义一个分组做为整个正则表达式的一部分，然后单独的对这部分对应匹配到的内容定位。

让我们来看一下它是怎么工作的：

Python

|  |  |
| --- | --- |
| 1 | >>> contactInfo = 'Doe, John: 555-1212' |

我刚才创建的字符串类似一个从某人的地址本里取出来的一个片段。我们可以通过这样一个正则表达式来匹配这一行：

Python

|  |  |
| --- | --- |
| 1  2 | >>> re.search(r'\w+, \w+: \S+', contactInfo)  <\_sre.SRE\_Match object at 0xb74e1ad8< |

通过用圆括号来（字符‘(’和‘)’）包围正则表达式的特定部分，我们可以对内容进行分组然后对这些子组做单独处理。

Python

|  |  |
| --- | --- |
| 1 | >>> match = re.search(r'(\w+), (\w+): (\S+)', contactInfo) |

这些分组可以通过用分组对象的group()方法得到。它们可以通过其在正则表达式中从左到右出现的数字顺序来定位（从1开始）：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> match.group(1)  'Doe'  >>> match.group(2)  'John'  >>> match.group(3)  '555-1212' |

组的序数从1开始的原因是因为第0个组被预留来存放所有匹配对象（我们在之前学习match()方法和search()方法到时候看到过）。

Python

|  |  |
| --- | --- |
| 1  2 | >>> match.group(0)  'Doe, John: 555-1212' |

**7. 使用 match.group 通过别名来分组**

有时候，特别是当一个正则表达式有很多分组的时候，通过组的出现次序来定位就会变的不现实。Python还允许你通过下面的语句来指定一个组名：

Python

|  |  |
| --- | --- |
| 1 | >>> match = re.search(r'(?P<last>\w+), (?P<first>\w+): (?P<phone>\S+)', contactInfo) |

我们还是可以用group()方法获取分组的内容，但这时候我们要用我们所指定的组名而不是之前所使用的组的所在位数。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> match.group('last')  'Doe'  >>> match.group('first')  'John'  >>> match.group('phone')  '555-1212' |

这大大加强了代码的明确性和可读性。你可以想像当正则表达式变得越来越复杂，去弄懂一个分组到捕获了什么内容将会变得越来越困难。给你的分组命名将明确的告诉了你和你的读者你的意图。

尽管findall()方法不返回分组对象，它也可以使用分组。类似的，findall()方法将返回一个元组的集合，其中每个元组中的第N个元素对应了正则表达式中的第N个分组。

Python

|  |  |
| --- | --- |
| 1  2 | >>> re.findall(r'(\w+), (\w+): (\S+)', contactInfo)  [('Doe', 'John', '555-1212')] |

但是，给分组命名并不适用于findall()方法。

在本文中我们介绍了Python中使用正则表达式的一些基础。我们学习了原始字符串类型（还有它能帮你解决的在使用正则表达式中一些头痛的问题）。我们还学习了如何适使用match(), search(), and findall()方法进行基本的查询，以及如何使用分组来处理匹配对象的子组件。

和往常一样，如果想查看更多关于这个主题的内容，re模块的Python官方文档是一个非常好的资源。