
正则表达式指南

发布 3.9.20

Guido van Rossum
and the Python development team

九月 08, 2024

Python Software Foundation
Email: docs@python.org

Contents

1	概述	2
2	简单正则	2
2.1	匹配字符	2
2.2	重复匹配	3
3	正则表达式的使用	4
3.1	编译正则表达式	4
3.2	反斜杠灾难	5
3.3	应用匹配	5
3.4	模块级函数	7
3.5	编译标志	7
4	更多模式能力	9
4.1	更多元字符	9
4.2	分组	10
4.3	非捕获和命名组	11
4.4	前视断言	13
5	修改字符串	13
5.1	分割字符串	14
5.2	搜索和替换	14
6	常见问题	16
6.1	使用字符串方法	16
6.2	match() 和 search()	16
6.3	贪婪与非贪婪	17
6.4	使用 re.VERBOSE	17
7	反馈	18

作者 A.M. Kuchling <amk@amk.ca>

摘要

本文档是一个关于如何在 Python 中使用 `re` 模块进行正则表达式操作的入门教程。相较于 Python 标准库参考文档中的相关章节，本教程采用了更为浅显易懂的讲解方式。

1 概述

正则表达式 (Regular expression, 也称为 RE、regex 或 regex pattern)，本质上是一种微型的、高度专业化的编程语言。在 Python 中，它通过 `re` 模块嵌入并提供使用。利用这种迷你语言，你可以指定一套规则，用于匹配你想要的一系列可能的字符串。这些字符串可以是英文句子、邮箱地址、TeX 命令，或者是任何你想要的内容。有了正则表达式，你就可以提出诸如“这个字符串是否匹配该 pattern?”、“在字符串中是否存在与该 pattern 相匹配的部分?”之类的问题。此外，你还可以用正则来修改字符串，或以各种方式将其拆分。

正则表达式 pattern 会被编译成一系列字节码，然后由 C 语言编写的匹配引擎执行。对于高级应用场景，可能需要仔细考虑引擎如何执行给定的正则表达式，并以特定的方式编写正则表达式，以生成运行速度更快的字节码。然而，本文不会涉及此类优化技术，因为这要求读者对匹配引擎的内部机制有深入的理解。

正则表达式语言相对较小且受限，因此并非所有的字符串处理任务都能用正则表达式完成。有些任务尽管 * 可以用正则表达式来完成，但表达式会变得非常复杂。在这些情况下，最好通过编写 Python 代码来进行处理。也许 Python 代码的运行速度会比精心设计的正则表达式慢，但通常更容易理解。

2 简单正则

让我们从最简单的正则表达式开始学习。由于正则表达式是用于处理字符串的，我们将从最常见的任务入手：匹配字符。

如果你想深入了解正则表达式背后的计算机科学原理（确定性和非确定性有限自动机），可参阅任何一本编译原理教材。

2.1 匹配字符

大多数字符在正则表达式中会简单地匹配自身。例如，正则表达式 `test` 将会精确地匹配到 `test`。（你可以启用不区分大小写模式，让这个正则也匹配 `Test` 或 `TEST`；关于这一点，稍后会详细讨论。）

但该规则有例外。有些字符是特殊的元字符 (*metacharacters*)，这些字符并不会匹配自身，而是用于指示需要匹配某些特殊的内容，或者用于影响正则的某部分，如重复匹配或改变含义。本文的大部分篇幅将用于详细讨论各种元字符及其用途。

以下是元字符的完整列表。它们的含义将在本指南的后续部分进行详细讨论。

```
. ^ $ * + ? { } [ ] \ | ( )
```

我们首先要介绍的元字符是 `[` 和 `]`。这两个元字符用于定义字符类，也就是一组你想匹配的字符。在方括号中，这些字符可以逐个地列出，也可以通过提供两个字符并用 `-` 分隔来表示一个字符范围。例如，`[abc]` 将匹配 `a`、`b`、`c` 之中的任意一个字符；这与 `[a-c]` 是一样的，后者使用一个范围来表示同一组字符。如果你只想匹配小写字母，则正则表达式应该是 `[a-z]`。

在字符类定义中，除反斜杠 `\` 外的元字符都会失去其特殊功能。例如，`[akm$]` 将会匹配 `'a'`、`'k'`、`'m'`、`'$'` 之中的任意一个字符。通常情况下，`'$'` 是一个元字符，但在字符类内部，它的特殊性质被消除了。

你可以通过对集合取反来匹配字符类中未列出的字符。具体做法是在字符类的开头加入一个脱字符 `^`。例如，`[^5]` 将匹配除 `'5'` 之外的任意字符。如果脱字符出现在字符类的其他位置，它将不再具有特殊含义。例如：`[5^]` 将匹配 `'5'` 或 `'^'`。

反斜杠 “\” 可能是正则表达式中最重要的元字符。就像在 Python 字符串字面量中一样，反斜杠后面可以跟随不同的字符，表示各种特殊序列。同时，反斜杠还可以用来转义元字符，使你能在 `pattern` 中匹配元字符本身。例如，如果需要匹配一个 `[` 或 `\`，可以在它们前面加上一个反斜杠来消除它们的特殊含义：`\[` 或 `\\`。

一些以 `'\'` 开头的特殊序列，它们代表了预定义的字符集，这些字符集通常非常有用，例如数字集、字母集或非空白字符集。

让我们来看一个例子：`\w` 匹配任何字母数字字符。如果正则 `pattern` 以字节形式表示，这个字符类等同于 `[a-zA-Z0-9_]`。如果正则 `pattern` 是字符串形式，`\w` 将匹配 Unicode 数据库中被标记为字母的任何字符，该数据库由 `unicodedata` 模块提供。如果想要在字符串 `pattern` 中使用更受限的 `\w` 定义，可以在编译正则表达式时添加 `re.ASCII` 标志。

以下为特殊序列的不完全列表。如果需要完整的序列列表，以及 Unicode 字符串 `pattern` 的扩展类定义，请参阅标准库参考中正则表达式语法的最后一部分。一般来说，Unicode 版本的字符类会匹配 Unicode 数据库中属于相应类别的任何字符。

`\d` 匹配任何十进制数字，等价于字符类 `[0-9]`。

`\D` 匹配任何非数字字符，等价于字符类 `[^0-9]`。

`\s` 匹配任何空白字符，等价于字符类 `[\t\n\r\f\v]`。

`\S` 匹配任何非空白字符，等价于字符类 `[^\t\n\r\f\v]`。

`\w` 匹配任何字母与数字字符，等价于字符类 `[a-zA-Z0-9_]`。

`\W` 匹配任何非字母与数字字符，等价于字符类 `[^a-zA-Z0-9_]`。

这些序列可以包含在字符类中。例如，`[\s,.]` 是一个匹配任何空白字符、`,` 或 `.` 的字符类。

本节最后要介绍的元字符是 `.`。它可以匹配除换行符以外的任何字符，并且有一个可选模式 (`re.DOTALL`)，在该模式下它甚至可以匹配换行符。`.` 通常用于需要匹配“任何字符”的场景。

2.2 重复匹配

正则表达式的第一个优势在于它能够匹配各种各样的字符集合，这是普通字符串方法无法做到的。但是，如果这就是正则表达式的唯一额外功能，那么它也就谈不上有多重大的优势。除此之外，正则表达式还有一个强大的功能：你可以指定正则的某部分必须重复特定次数。

首先，我们来看一个用于重复匹配的元字符 `*`。注意，这里的 `*` 并不匹配字面意义上的星号 `'*'`，而是用来指定前面的字符可以重复出现零次或多次，而不是仅仅是恰好出现一次。

例如，正则表达式 `ca*t` 可以匹配 `'ct'` (0 个 `'a'`)、`'cat'` (1 个 `'a'`)、`'caaat'` (3 个 `'a'`) 等等。

像 `*` 这样的重复匹配是贪婪的。当重复匹配正则时，匹配引擎会尝试重复尽可能多的次数。如果 `pattern` 的后续部分无法匹配成功，匹配引擎会回退并尝试减少重复次。

我们可以通过一个逐步的例子来更好理解这一点。来看看这个正则表达式 `a[bcd]*b`。该表达式匹配的是，首先是一个字母 `'a'`，然后是零个或多个来自字符类 `[bcd]` 的字母，最后以一个 `'b'` 结尾。现在，让我们想象一下用这个正则去匹配字符串 `'abcbcd'` 的过程。

步骤	匹配	说明
1	a	正则中的 a 匹配成功。
2	abcbcd	引擎尽可能多地匹配 <code>[bcd]*</code> ，直至字符串末尾。
3	失败	引擎试图匹配正则中的 b，但是当前位置已到达字符串末尾，因此匹配失败。
4	abcb	回退，让 <code>[bcd]*</code> 少匹配一个字符。
5	失败	再次尝试匹配正则中的 b，但是当前位置上的字符是最后一个字符 <code>'d'</code> 。
6	abc	再次回退，让 <code>[bcd]*</code> 只匹配 <code>bc</code> 。
6	abcb	再次尝试匹配正则中的 b。这次，当前位置的字符是 <code>'b'</code> ，因此匹配成功。

此时，匹配引擎已经到达了正则表达式的尽头，并且成功匹配到了 `'abcb'`。这个例子表明了匹配引擎一开始会尽其所能地向前匹配，如果没有找到匹配，它会逐步回退，并重新尝试匹配正则的剩余部分，

如此往复，回退过程会一直持续到 `[bcd]*` 的匹配次数为零。如果随后的匹配还是失败了，那么引擎将得出结论：这个字符串与正则表达式不匹配。

正则表达式中另一个常用的重复元字符是 `+`，它表示匹配一次或更多次。要特别注意 `*` 与 `+` 之间的区别。`*` 匹配零次或更多次，也就是说它所重复的部分有可能是根本不存在的。而 `+` 则要求至少出现一次。举一个类似的例子，正则 `ca+t` 可以匹配 `'cat'`（1 个 `'a'`）或 `'caaat'`（3 个 `'a'`），但不能匹配 `'ct'`。

还有两个重复限定符。问号 `?` 表示匹配一次或零次。你可以认为它把内容变成了可选的。例如，`home-?brew` 可以匹配 `'homebrew'` 或 `'home-brew'`。

最复杂的重复限定符是 `{m,n}`，其中 `m` 和 `n` 是十进制整数。该限定符意味着必须至少重复 `m` 次，最多重复 `n` 次。例如，`a/{1,3}b` 可以匹配 `'a/b'`、`'a//b'` 或者 `'a///b'`，但不能匹配中间没有斜杆的 `'ab'`，或者四个斜杆的 `'a////b'`。

这个量词中的 `m` 和 `n` 都是可以省略的。在这种情况下，系统会为缺失的值赋予一个合理的默认值。如果省略 `m`，下限默认为 0；如果省略 `n`，则上限默认为无限大。

细心的读者也许注意到了，前面的三个重复限定符都可以用这种标记法来表示。`{0,}` 等同于 `*`，`{1,}` 等同于 `+`，`{0,1}` 等同于 `?`。如果可以的话，最好使用 `*`、`+`、或 `?`，因为后者更简洁易读。

3 正则表达式的使用

现在我们已经了解了一些简单的正则表达式，那么如何在 Python 中实际使用它们呢？`re` 模块提供了正则表达式引擎的接口，使你能够将正则表达式编译为对象，然后用使用这些对象进行匹配操作。

3.1 编译正则表达式

正则表达式会被编译成 `pattern` 对象，这些 `pattern` 对象具有各种操作方法，例如搜索模式匹配或执行字符串替换。：

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

`re.compile()` 也接受一个可选的 `flags` 参数，用于启用各种特殊功能和语法变体。我们稍后将介绍可用的设置，但现在先举一个简单的例子

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

在这个例子中，正则以字符串形式传递给了 `re.compile()`。之所将正则作为字符串处理，是因为正则表达式并非 Python 语言的核心组成部分，也没有为其创建专门的语法。（有些应用程序根本不需要正则，因此也就没必要将它纳入语言核心而使语言规范变得臃肿。）实际上，`re` 模块只是一个 Python 附带的 C 扩展模块，就像 `socket` 或 `zlib` 模块一样。

使用字符串存放正则表达式，的确可以让 Python 语言保持简洁，但也带来了一个缺点，这将是下一节讨论的主题。

3.2 反斜杠灾难

如前所述，正则表达式使用反斜杠字符 ('\\') 来表示特殊形式，或用于消除元字符的特殊含义来匹配字面上的字符。然而，反斜杠在 Python 字符串的字面量中也有相同的用法，两者之间存在冲突。

假设你想编写一个正则表达式，用于匹配字符串 `\section`，这个字符串可能出现在 LaTeX 文件中。要明确在程序代码中如何编写，首先从想要匹配的字符串开始。接着，你需要对其中的任何反斜杠和其他元字符进行转义，即在它们前面加上反斜杠，如此一来，得到了字符串 `\\section`。也就是说，最终传递给 `re.compile()` 的结果字符串必须是 `\\section`。然而，要将其表示为 Python 字符串字面量，两个反斜杠必须 * 再 * 进行一次转义。

字符	阶段
<code>\section</code>	需要匹配的文本字符串
<code>\\section</code>	为 <code>re.compile()</code> 而转义反斜杠
<code>"\\\\section"</code>	为字符串字面量而转义反斜杠

简而言之，要匹配一个字面上的反斜杠，必须将正则表达式字符串写为 `'\\\\'`。这是因为正则表达式本身必须是 `\\`，而在 Python 普通字符串字面量中，每个反斜杠都必须表示为 `\\`。在反复使用反斜杠的正则中，这会导致大量重复的反斜杠，使得最终的字符串难以理解。

要解决这个问题，可以使用 Python 的原始字符串表示法来书写正则表达式。在以 `'r'` 为前缀的字符串字面量中，反斜杠不需要作特殊处理，因此 `r"\n"` 是一个包含两个字符的字符串，分别为 `'\'` 和 `'n'`。而 `"\n"` 则是一个只包含单个换行符的字符串。在 Python 代码中，正则表达式通常会使用这种原始字符串表示法来书写。

此外，某些在正则表达式中有效，但在 Python 普通字符串字面量中无效的特殊转义序列，现在会引发 `DeprecationWarning` (弃用警告)，并且在未来可能会变为一个 `SyntaxError` (语法错误)。这意味着，如果没有使用原始字符串就直接书写正则本身，这种特殊序列将变得无效。

普通字符串	原始字符串
<code>"ab*"</code>	<code>r"ab*"</code>
<code>"\\\\section"</code>	<code>r"\\section"</code>
<code>"\\w+\\s+\\1"</code>	<code>r"\\w+\\s+\\1"</code>

3.3 应用匹配

当你已经得到一个编译好的正则表达式对象，接下来该怎么用呢？`Pattern` 对象有多个方法和属性，但这里我们只介绍最重要的几个。如果想要了解全部内容，请参阅 `re` 模块的官方文档。

方法 / 属性	目的
<code>match()</code>	检查正则表达式是否能在字符串开头找到匹配
<code>search()</code>	扫描整个字符串的所有位置，查找与正则表达式匹配的首个位置。
<code>findall()</code>	查找所有与正则表达式匹配的子串，并以列表形式返回。
<code>finditer()</code>	查找所有与正则表达式匹配的子串，并以迭代器形式 <code>iterator</code> 返回。

如果 `match()` 和 `search()` 方法没有找到匹配项，它们会返回 `None`。如果成功找到匹配项，则返回一个匹配对象 (`match object`) 实例。这个对象实例包含了关于匹配结果的详细信息，比如匹配项的起始和结束位置、匹配到的子字符串等等。

你可以通过与 `re` 模块进行交互式的试验来学习这个。如果你有 `tkinter`，你也可以看看 [Tools/demo/redemo.py](#)，这是一个 Python 发行版自带的演示程序。它允许你输入正则和字符串，并显示正则是否匹配还是失败。`redemo.py` 在调试复杂的正则时非常有用。

本教程使用标准 Python 解释器来演示例子。首先，运行 Python 解释器，导入 `re` 模块，然后编译一个正则表达式

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

现在，你可以尝试用正则 `[a-z]+` 去匹配不同的字符串。空字符串是无法匹配的，因为 `+` 表示“一次或多次重复”。在这种情况下，`match()` 会返回 `None`，导致解释器没有任何输出。为了更直观地了解这一点，你可以显式打印 `match()` 的返回结果。

```
>>> p.match("")
>>> print(p.match(""))
None
```

接下来，让我们尝试把这个正则应用到一个可以匹配的字符串，比如 `tempo`。在这个例子中，`match()` 会返回一个匹配对象，所以你应该将结果储存到一个变量中，以便后续使用。

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

现在你可以查询匹配对象来获取有关匹配字符串的信息。匹配对象实例也有多个方法和属性，其中比较最重要的是：

方法 / 属性	目的
<code>group()</code>	返回正则匹配的字符串
<code>start()</code>	返回匹配的开始位置
<code>end()</code>	返回匹配的结束位置
<code>span()</code>	返回包含匹配(起始, 结束)位置的元组

测一下这些方法，很快就能明白它们的含义：

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` 方法返回正则表达式匹配到的子字符串。`start()` 和 `end()` 方法分别返回匹配的起始和结束索引。`span()` 方法则以元组的形式返回起始和结束索引。由于 `match()` 方法只检查正则表达式能否在字符串的开头匹配，所以 `start()` 方法始终返回零。而 `search()` 方法会扫描字符串的所有位置，因此匹配的起始索引不一定是零。：

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

在实际编程中，最常用的方式是将匹配对象储存到一个变量中，然后检查这个变量是否为 `None`。这种做法通常如下所示：

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
```

(下页继续)

```
else:
    print('No match')
```

有两个 `pattern` 方法可以用来返回 `pattern` 的所有匹配项。`findall()` 方法会返回一个包含所有匹配字符串的列表：

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

在这个例子中，`r` 前缀用于将字面量标记为原始字符串字面量，这是必要的。因为在普通的“加工”字符串字面量中，如果出现 Python 无法识别的转义序列（尽管这个序列在正则表达式中是合法的），会引发一个 `DeprecationWarning`，并且在未来会变成 `SyntaxError`。请参阅[反斜杠灾难](#)。

`findall()` 方法必须在返回结果前生成一个完整的匹配列表。而 `finditer()` 方法则是以迭代器 `iterator` 的形式返回一系列匹配对象实例。

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

3.4 模块级函数

除了通过创建 `pattern` 对象来调用其方法外，`re` 模块还提供了一些顶层函数，例如 `match()`、`search()`、`findall()`、`sub()` 等。这些函数的参数与相应的 `pattern` 对象方法相同，只是需要将正则字符串作为第一个参数传入。它们同样会返回 `None` 或 匹配对象实例。

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From ' >
```

在底层实现上，这些函数其实就是为你创建一个 `pattern` 对象，然后在该对象上调用相应的方法。同时，它们还将编译后的对象存储在缓存中，这样在后续使用同一正则表达式时，就能避免重复解析 `pattern` 了。

我们应该使用这些模块级函数，还是创建 `pattern` 对象并调用其方法呢？这要看具体情况。如果你是在循环中多次使用同一个正则表达式，那么预编译正则可以减少一些函数调用。而在循环之外，由于内部缓存的存在，两种方式差别不大。

3.5 编译标志

通过编译标志，你能够调整正则表达式的某些匹配方式。在 `re` 模块中，这些标志有两种命名方式：一种是长名称，如 `IGNORECASE`；另一种是单字母的短名称，如 `I`。（如果你熟悉 Perl 的模式修饰符，就会发现这些单字母形式使用了相同的字母。例如，`re.VERBOSE` 的缩写形式为 `re.X`。）你可以通过按位或运算来同时指定多个标志。比如，`re.I | re.M` 就同时设置了 `I` 和 `M` 两个标志。

以下是可用标志的列表，以及每个标志的详细说明。

标志位	含意
ASCII, A	使得 \w、\b、\s 和 \d 等转义序列，仅匹配具有相应属性的 ASCII 字符。
DOTALL, S	使 . 匹配任何字符，包括换行符。
IGNORECASE, I	进行忽略大小写的匹配。
LOCALE, L	进行区域设置感知的匹配。
MULTILINE, M	多行匹配，影响 ^ 和 \$ 的匹配行为。
VERBOSE, X (代表 extended)	启用详细模式的正则，可以使表达式组织得更清晰，更易懂。

I

IGNORECASE

执行不区分大小写的匹配；字符类和字面字符串将通过忽略大小写来匹配字母。例如，[A-Z] 也匹配小写字母。除非使用 ASCII 标志来禁用非 ASCII 匹配，否则完全 Unicode 匹配也有效。当 Unicode 模式 [a-z] 或 [A-Z] 与 IGNORECASE 标志结合使用时，它们将匹配 52 个 ASCII 字母和 4 个额外的非 ASCII 字母：'İ' (U+0130，拉丁大写字母 I，带上面的点)，'ı' (U+0131，拉丁文小写字母无点 i)，'ş' (U+017F，拉丁文小写字母长 s) 和 'K' (U+212A，开尔文符号)。Spam 将匹配 'Spam'，'spam'，'sPAM' 或 'fpam' (后者仅在 Unicode 模式下匹配)。此小写不考虑当前区域设置；如果你还设置了 LOCALE 标志，则将考虑。

L

LOCALE

使 \w、\W、\b、\B 和大小写敏感匹配依赖于当前区域而不是 Unicode 数据库。

区域设置是 C 库的一个功能，旨在帮助编写考虑到语言差异的程序。例如，如果你正在处理编码的法语文本，那么你希望能够编写 \w+ 来匹配单词，但 \w 只匹配字符类 [A-Za-z] 字节模式；它不会匹配对应于 é 或 ç 的字节。如果你的系统配置正确并且选择了法语区域设置，某些 C 函数将告诉程序对应于 é 的字节也应该被视为字母。在编译正则表达式时设置 LOCALE 标志将导致生成的编译对象将这些 C 函数用于 \w；这比较慢，但也可以使 \w+ 匹配你所期望的法语单词。在 Python 3 中不鼓励使用此标志，因为语言环境机制非常不可靠，它一次只处理一个“文化”，它只适用于 8 位语言环境。默认情况下，Python 3 中已经为 Unicode (str) 模式启用了 Unicode 匹配，并且它能够处理不同的区域/语言。

M

MULTILINE

(^ 和 \$ 还没有解释；它们将在以下部分介绍更多元字符。)

通常 ^ 只匹配字符串的开头，而 \$ 只匹配字符串的结尾，紧接在字符串末尾的换行符（如果有的话）之前。当指定了这个标志时，^ 匹配字符串的开头和字符串中每一行的开头，紧跟在每个换行符之后。类似地，\$ 元字符匹配字符串的结尾和每行的结尾（紧接在每个换行符之前）。

S

DOTALL

使 '.' 特殊字符匹配任何字符，包括换行符；没有这个标志，'.' 将匹配任何字符除了换行符。

A

ASCII

使 \w、\W、\b、\B、\s 和 \S 执行仅 ASCII 匹配而不是完整匹配 Unicode 匹配。这仅对 Unicode 模式有意义，并且对于字节模式将被忽略。

X

VERBOSE

此标志允许你编写更易读的正则表达式，方法是为您提供更灵活的格式化方式。指定此标志后，将忽略正则字符串中的空格，除非空格位于字符类中或前面带有未转义的反斜杠；这使你可以更清楚地组织和缩进正则。此标志还允许你将注释放在正则中，引擎将忽略该注释；注释标记为 '#' 既不是在字符类中，也不是在未转义的反斜杠之前。

例如，这里的正则使用 re.VERBOSE；看看阅读有多容易？：

```
charref = re.compile(r"""
    &[#]          # Start of a numeric entity reference
```

(下页继续)

```
(
    0[0-7]+          # Octal form
    | [0-9]+         # Decimal form
    | x[0-9a-fA-F]+  # Hexadecimal form
)
;                  # Trailing semicolon
"""", re.VERBOSE)
```

如果没有详细设置，正则将如下所示：

```
charref = re.compile("&#(0[0-7]+"
```

```
    "[0-9]+"
```

```
    "|x[0-9a-fA-F]+);")
```

在上面的例子中，Python 的字符串文字的自动连接已被用于将正则分解为更小的部分，但它仍然比以下使用 `re.VERBOSE` 版本更难理解。

4 更多模式能力

到目前为止，我们只介绍了正则表达式的一部分功能。在本节中，我们将介绍一些新的元字符，以及如何使用组来检索匹配的文本部分。

4.1 更多元字符

我们还没有涉及到一些元字符。其中大部分内容将在本节中介绍。

要讨论的其余一些元字符是 零宽度断言。它们不会使解析引擎在字符串中前进一个字符；相反，它们根本不占用任何字符，只是成功或失败。例如，`\b` 是一个断言，指明当前位置位于字边界；这个位置根本不会被 `\b` 改变。这意味着永远不应重复零宽度断言，因为如果它们在给定位置匹配一次，它们显然可以无限次匹配。

| 或者 “or” 运算符。如果 *A* 和 *B* 是正则表达式，*A|B* 将匹配任何与 *A* 或 *B* 匹配的字符串。| 具有非常低的优先级，以便在交替使用多字符串时使其合理地工作。`Crow|Servo` 将匹配 'Crow' 或 'Servo'，而不是 'Cro'、'w' 或 'S' 和 'ervo'。

要匹配字面 '|', 请使用 `\|`，或将其括在字符类中，如 `[|]`。

^ 在行的开头匹配。除非设置了 `MULTILINE` 标志，否则只会在字符串的开头匹配。在 `MULTILINE` 模式下，这也在字符串中的每个换行符后立即匹配。

例如，如果你希望仅在行的开头匹配单词 `From`，则要使用的正则 `^From`。：

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

要匹配字面 '^', 使用 `\^`。

\$ 匹配行的末尾，定义为字符串的结尾，或者后跟换行符的任何位置。：

```
>>> print(re.search('{}$', '{block}'))
<re.Match object; span=(6, 7), match='{}'>
>>> print(re.search('{}$', '{block} '))
None
>>> print(re.search('{}$', '{block}\n'))
<re.Match object; span=(6, 7), match='{}'>
```

以匹配字面 '\$', 使用 `\$` 或者将其包裹在一个字符类中，例如 `[$]`。

\A 仅匹配字符串的开头。当不在 MULTILINE 模式时，\A 和 ^ 实际上是相同的。在 MULTILINE 模式中，它们是不同的：\A 仍然只在字符串的开头匹配，但 ^ 可以匹配在换行符之后的字符串内的任何位置。

\Z 只匹配字符串尾。

\b 字边界。这是一个零宽度断言，仅在单词的开头或结尾处匹配。单词被定义为一个字母数字字符序列，因此单词的结尾由空格或非字母数字字符表示。

以下示例仅当它是一个完整的单词时匹配 class；当它包含在另一个单词中时将不会匹配。

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

使用这个特殊序列时，你应该记住两个细微之处。首先，这是 Python 的字符串文字和正则表达式序列之间最严重的冲突。在 Python 的字符串文字中，\b 是退格字符，ASCII 值为 8。如果你没有使用原始字符串，那么 Python 会将 \b 转换为退格，你的正则不会按照你的预期匹配。以下示例与我们之前的正则看起来相同，但省略了正则字符串前面的 'r'。：

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

其次，在一个字符类中，这个断言没有用处，\b 表示退格字符，以便与 Python 的字符串文字兼容。

\B 另一个零宽度断言，这与 \b 相反，仅在当前位置不在字边界时才匹配。

4.2 分组

通常，你需要获取更多信息，而不仅仅是正则是否匹配。正则表达式通常用于通过将正则分成几个子组来解析字符串，这些子组匹配不同的感兴趣组件。例如，RFC-822 标题行分为标题名称和值，用 ':' 分隔，如下所示：

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

这可以通过编写与整个标题行匹配的正则表达式来处理，并且具有与标题名称匹配的一个组，以及与标题的值匹配的另一个组。

组由 '(', ')' 元字符标记。'(' 和 ')' 与数学表达式的含义大致相同；它们将包含在其中的表达式组合在一起，你可以使用重复限定符重复组的内容，例如 *，+，? 或 {m,n}。例如，(ab)* 将匹配 ab 的零次或多次重复。：

```
>>> p = re.compile('(ab)*')
>>> print(p.match('ababababab').span())
(0, 10)
```

用 '(', ')' 表示的组也捕获它们匹配的文本的起始和结束索引；这可以通过将参数传递给 group()、start()、end() 以及 span()。组从 0 开始编号。组 0 始终存在；它表示整个正则，所以匹配对象方法都将组 0 作为默认参数。稍后我们将看到如何表达不捕获它们匹配的文本范围的组。：

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

子组从左到右编号，从 1 向上编号。组可以嵌套；要确定编号，只需计算从左到右的左括号字符。：

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` 可以一次传递多个组号，在这种情况下，它将返回一个包含这些组的相应值的元组。：

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

`groups()` 方法返回一个元组，其中包含所有子组的字符串，从 1 到最后一个子组。：

```
>>> m.groups()
('abc', 'b')
```

模式中的后向引用允许你指定还必须在字符串中的当前位置找到先前捕获组的内容。例如，如果可以在当前位置找到组 1 的确切内容，则 `\1` 将成功，否则将失败。请记住，Python 的字符串文字也使用反斜杠后跟数字以允许在字符串中包含任意字符，因此正则中引入反向引用时务必使用原始字符串。

例如，以下正则检测字符串中重复的单词。：

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

像这样的后向引用通常不仅仅用于搜索字符串——很少有文本格式以这种方式重复数据——但是你很快就会发现它们在执行字符串替换时非常有用。

4.3 非捕获和命名组

精心设计的正则可以使用许多组，既可以捕获感兴趣的子串，也可以对正则本身进行分组和构建。在复杂的正则中，很难跟踪组号。有两个功能可以帮助解决这个问题。它们都使用常用语法进行正则表达式扩展，因此我们首先看一下。

Perl 5 以其对标准正则表达式的强大补充而闻名。对于这些新功能，Perl 开发人员无法选择新的单键击元字符或以 `\` 开头的新特殊序列，否则 Perl 的正则表达式与标准正则容易混淆。例如，如果他们选择 `&` 作为一个新的元字符，旧的表达式将假设 `&` 是一个普通字符，并且不会编写 `\&` 或 `[&]`。

Perl 开发人员选择的解决方案是使用 `(?...)` 作为扩展语法。括号后面紧跟 `?` 是一个语法错误，因为 `?` 没有什么可重复的，所以这样并不会带来任何兼容性问题。紧跟在 `?` 之后的字符表示正在使用的扩展语法，所以 `(?=foo)` 是一种语法（一个前视断言）和 `(?:foo)` 是另一种语法（包含子表达式 `foo` 的非捕获组）。

Python 支持一些 Perl 的扩展，并增加了新的扩展语法用于 Perl 的扩展语法。如果在问号之后的第一个字符为 `P`，即表明其为 Python 专属的扩展。

现在我们已经了解了一般的扩展语法，我们可以回到简化复杂正则中组处理的功能。

有时你会想要使用组来表示正则表达式的一部分，但是对检索组的内容不感兴趣。你可以通过使用非捕获组来显式表达这个事实：(?:...)，你可以用任何其他正则表达式替换 ...。：

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

除了你无法检索组匹配内容的事实外，非捕获组的行为与捕获组完全相同；你可以在里面放任何东西，用重复元字符重复它，比如 *，然后把它嵌入其他组（捕获或不捕获）。(?:...) 在修改现有模式时特别有用，因为你可以添加新组而不更改所有其他组的编号方式。值得一提的是，捕获和非捕获组之间的搜索没有性能差异；两种形式没有一种更快。

更重要的功能是命名组：不是通过数字引用它们，而是可以通过名称引用组。

命名组的语法是 Python 特定的扩展之一：(?P<name>...)。name 显然是该组的名称。命名组的行为与捕获组完全相同，并且还将名称与组关联。处理捕获组的匹配对象方法都接受按编号引用组的整数或包含所需组名的字符串。命名组仍然是给定的数字，因此你可以通过两种方式检索有关组的信息：

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search('((( Lots of punctuation )))')
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

此外，你可以通过 groupdict() 将命名分组提取为一个字典：

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

命名组很有用，因为它们允许你使用容易记住的名称，而不必记住数字。这是来自 imaplib 模块的示例正则

```
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
    r'(?P<year>[0-9][0-9][0-9][0-9])'
    r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
    r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
    r'")')
```

检索 m.group('zonem') 显然要容易得多，而不必记住检索第 9 组。

表达式中的后向引用语法，例如 (...) \1，指的是组的编号。当然有一种变体使用组名而不是数字。这是另一个 Python 扩展：(?P=name) 表示在当前点再次匹配名为 name 的组的内容。用于查找重复单词的正则表达式，\b(\w+)\s+\1\b 也可以写为 \b(?P<word>\w+)\s+(?P=word)\b:

```
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

4.4 前视断言

另一个零宽断言是前视断言。前视断言有肯定型和否定型两种形式，如下所示：

(?=...) 肯定型前视断言。如果内部的表达式（这里用...来表示）在当前位置可以匹配，则匹配成功，否则匹配失败。但是，内部表达式尝试匹配之后，正则引擎并不会向前推进；正则表达式的其余部分依然会在断言开始的地方尝试匹配。

(?!...) 否定型前视断言。与肯定型断言正好相反，如果内部表达式在字符串中的当前位置不匹配，则成功。

更具体一些，来看一个前视的实用案例。考虑用一个简单的表达式来匹配文件名并将其拆分为基本名称和扩展名，以.分隔。例如，在news.rc中，news是基本名称，rc是文件名的扩展名。

与此匹配的模式非常简单：

```
.*[.].*$
```

请注意，.需要特别处理，因为它是元字符，所以它在字符类中只能匹配特定字符。还要注意尾随的\$；添加此项以确保扩展名中的所有其余字符串都必须包含在扩展名中。这个正则表达式匹配foo.bar、autoexec.bat、sendmail.cf和printers.conf。

现在，考虑使更复杂一点的问题；如果你想匹配扩展名不是bat的文件名怎么办？一些错误的尝试：

.*[.][^b].*\$ 上面的第一次尝试试图通过要求扩展名的第一个字符不是b来排除bat。这是错误的，因为模式也与foo.bar不匹配。

```
.*[.]( [^b]... | .[^a]... | ...[^t] )$
```

当你尝试通过要求以下一种情况匹配来修补第一个解决方案时，表达式变得更加混乱：扩展的第一个字符不是b。第二个字符不是a；或者第三个字符不是t。这接受foo.bar并拒绝autoexec.bat，但它需要三个字母的扩展名，并且不接受带有两个字母扩展名的文件名，例如sendmail.cf。为了解决这个问题，我们会再次使模式复杂化。

```
.*[.]( [^b].??.? | .[^a]??.? | ...?[^t]? )$
```

在第三次尝试中，第二个和第三个字母都是可选的，以便允许匹配的扩展名短于三个字符，例如sendmail.cf。

模式现在变得非常复杂，这使得它难以阅读和理解。更糟糕的是，如果问题发生变化并且你想要将bat和exe排除为扩展，那么该模式将变得更加复杂和混乱。

否定型前视可以解决所有这些困扰：

.*[.](?!bat\$)[^.]*\$ 否定型前视意味着：如果表达式bat在当前位置不能匹配，则可以接着尝试正则表达式的其余部分；如果bat\$能匹配，则整个正则表达式将匹配失败。尾随的\$是必需的，以确保可以匹配到像sample.batch这样以bat开头的文件名。当文件名中有多个点号时，[^.]*可以确保表达式依然有效。

现在很容易排除另一个文件扩展名；只需在断言中添加它作为替代。以下模块排除以bat或exe：

```
.*[.](?!bat$|exe$)[^.]*$
```

5 修改字符串

到目前为止，我们只是针对静态字符串执行搜索。正则表达式通常也用于以各种方式修改字符串，使用以下模式方法：

方法 / 属性	目的
split()	将字符串拆分为一个列表，在正则匹配的任何地方将其拆分
sub()	找到正则匹配的所有子字符串，并用不同的字符串替换它们
subn()	与sub()相同，但返回新字符串和替换次数

5.1 分割字符串

模式的 `split()` 方法在正则匹配的任何地方拆分字符串，返回一个片段列表。它类似于 `split()` 字符串方法，但在分隔符的分隔符中提供了更多的通用性；字符串的 `split()` 仅支持按空格或固定字符串进行拆分。正如你所期望的那样，还有一个模块级 `re.split()` 函数。

.split(*string*[, *maxsplit*=0])

通过正则表达式的匹配拆分字符串。如果在正则中使用捕获括号，则它们的内容也将作为结果列表的一部分返回。如果 *maxsplit* 非零，则最多执行 *maxsplit* 次拆分。

你可以通过传递 *maxsplit* 的值来限制分割的数量。当 *maxsplit* 非零时，将最多进行 *maxsplit* 次拆分，并且字符串的其余部分将作为列表的最后一个元素返回。在以下示例中，分隔符是任何非字母数字字符序列。

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

有时你不仅对分隔符之间的文本感兴趣，而且还需要知道分隔符是什么。如果在正则中使用捕获括号，则它们的值也将作为列表的一部分返回。比较以下调用：

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', ' ', 'a', ' ', 'test', '.', '']
```

模块级函数 `re.split()` 添加要正则作为第一个参数，但在其他方面是相同的。：

```
>>> re.split(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ' ', ' ', 'words', ' ', ' ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 搜索和替换

另一个常见任务是找到模式的所有匹配项，并用不同的字符串替换它们。`sub()` 方法接受一个替换值，可以是字符串或函数，也可以是要处理的字符串。

.sub(*replacement*, *string*[, *count*=0])

返回通过替换 *replacement* 替换 *string* 中正则的最左边非重叠出现而获得的字符串。如果未找到模式，则 *string* 将保持不变。

可选参数 *count* 是要替换的模式最大的出现次数；*count* 必须是非负整数。默认值 0 表示替换所有。

这是一个使用 `sub()` 方法的简单示例。它用 `colour` 这个词取代颜色名称：

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

`subn()` 方法完成相同的工作，但返回一个包含新字符串值和已执行的替换次数的 2 元组：

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

仅当空匹配与前一个空匹配不相邻时，才会替换空匹配。：

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

如果 *replacement* 是一个字符串，则处理其中的任何反斜杠转义。也就是说，`\n` 被转换为单个换行符，`\r` 被转换为回车符，依此类推。诸如 `\&` 之类的未知转义是孤立的。后向引用，例如 `\6`，被替换为正则中相应组匹配的子字符串。这使你可以在生成的替换字符串中合并原始文本的部分内容。

这个例子匹配单词 `section` 后跟一个用 `{, }` 括起来的字符串，并将 `section` 改为 `subsection`

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First} section{second}')
'subsection{First} subsection{second}'
```

还有一种语法用于引用由 `(?P<name>...)` 语法定义的命名组。`\g<name>` 将使用名为 `name` 的组匹配的子字符串，`\g<number>` 使用相应的组号。因此 `\g<2>` 等同于 `\2`，但在诸如 `\g<2>0` 之类的替换字符串中并不模糊。（`\20` 将被解释为对组 20 的引用，而不是对组 2 的引用，后跟字面字符 `'0'`。）以下替换都是等效的，但使用所有三种变体替换字符串。：

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}', 'section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}', 'section{First}')
'subsection{First}'
```

replacement 也可以是一个函数，它可以为你提供更多信息。如果 *replacement* 是一个函数，则为 *pattern* 的每次非重叠出现将调用该函数。在每次调用时，函数都会传递一个匹配的匹配对象参数，并可以使用此信息计算所需的替换字符串并将其返回。

在以下示例中，替换函数将小数转换为十六进制：

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

使用模块级别 `re.sub()` 函数时，模式作为第一个参数传递。模式可以是对象或字符串；如果需要指定正则表达式标志，则必须使用模式对象作为第一个参数，或者在模式字符串中使用嵌入式修饰符，例如：`sub("(?i)b+", "x", "bbbb BBBB")` 返回 `'x x'`。

6 常见问题

正则表达式对于某些应用程序来说是一个强大的工具，但在某些方面，它们的行为并不直观，有时它们的行为方式与你的预期不同。本节将指出一些最常见的陷阱。

6.1 使用字符串方法

有时使用 `re` 模块是一个错误。如果你匹配固定字符串或单个字符类，并且你没有使用任何 `re` 功能，例如 `IGNORECASE` 标志，那么正则表达式的全部功能可能不是必需的。字符串有几种方法可以使用固定字符串执行操作，它们通常要快得多，因为实现是一个针对此目的而优化的单个小 C 循环，而不是大型、更通用的正则表达式引擎。

一个例子可能是用另一个固定字符串替换一个固定字符串；例如，你可以用 `deed` 替换 `word`。`re.sub()` 看起来像是用于此的函数，但请考虑 `replace()` 方法。注意 `replace()` 也会替换单词里面的 `word`，把 `swordfish` 变成 `sdeedfish`，但简单的正则 `word` 也会这样做。（为了避免对单词的部分进行替换，模式必须是 `\bword\b`，以便要求 `word` 在任何一方都有一个单词边界。这使得工作超出了 `replace()` 的能力。）

另一个常见任务是从字符串中删除单个字符的每个匹配项或将其替换为另一个字符。你可以用 `re.sub('\n', ' ', s)` 之类的东西来做这件事，但是 `translate()` 能够完成这两项任务，并且比任何正则表达式都快。

简而言之，在转向 `re` 模块之前，请考虑是否可以使用更快更简单的字符串方法解决问题。

6.2 `match()` 和 `search()`

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

另一方面，`search()` 将向前扫描字符串，报告它找到的第一个匹配项。：

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

有时你会被诱惑继续使用 `re.match()`，只需在你的正则前面添加 `.`。抵制这种诱惑并使用 `re.search()` 代替。正则表达式编译器对正则进行一些分析，以加快寻找匹配的过程。其中一个分析可以确定匹配的第二个特征必须是什么；例如，以 `Crow` 开头的模式必须与 `'C'` 匹配。分析让引擎快速扫描字符串，寻找起始字符，只在找到 `'C'` 时尝试完全匹配。

添加 `.` 会使这个优化失效，需要扫描到字符串的末尾，然后回溯以找到正则的其余部分的匹配。使用 `re.search()` 代替。

6.3 贪婪与非贪婪

当重复一个正则表达式时，就像在 `a*` 中一样，最终的动作就是消耗尽可能多的模式。当你尝试匹配一对对称分隔符，例如 `HTML` 标记周围的尖括号时，这个事实经常会让你感到困惑。因为 `.` 的贪婪性质，用于匹配单个 `HTML` 标记的简单模式不起作用。

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

正则匹配 `'<'` 中的 `'<html>'` 和 `.` 消耗字符串的其余部分。正则中还有更多的剩余东西，并且 `>` 在字符串的末尾不能匹配，所以正则表达式引擎必须逐个字符地回溯，直到它找到匹配 `>`。最终匹配从 `'<html>'` 中的 `'<'` 扩展到 `'</title>'` 中的 `'>'`，而这并不是你想要的结果。

在这种情况下，解决方案是使用非贪婪的限定符 `*?`、`+?`、`??` 或 `{m,n}?`，匹配为尽可能少的文字。在上面的例子中，在第一次 `'<'` 匹配后立即尝试 `'>'`，当它失败时，引擎一次前进一个字符，每一步都重试 `'>'`。这产生了正确的结果：

```
>>> print(re.match('<.*?>', s).group())
<html>
```

(请注意，使用正则表达式解析 `HTML` 或 `XML` 很痛苦。快而脏的模式将处理常见情况，但 `HTML` 和 `XML` 有特殊情况会破坏明显的正则表达式；当你编写正则表达式处理所有可能的情况时，模式将非常复杂。使用 `HTML` 或 `XML` 解析器模块来执行此类任务。)

6.4 使用 re.VERBOSE

到目前为止，你可能已经注意到正则表达式是一种非常紧凑的表示法，但它们并不是非常易读。具有中等复杂度的正则可能会成为反斜杠、括号和元字符的冗长集合，使其难以阅读和理解。

对于这样的正则，在编译正则表达式时指定 `re.VERBOSE` 标志可能会有所帮助，因为它允许你更清楚地格式化正则表达式。

`re.VERBOSE` 标志有几种效果。正则表达式中的 不是在字符类中的空格将被忽略。这意味着表达式如 `dog | cat` 等同于不太可读的 `dog|cat`，但 `[a b]` 仍将匹配字符 `'a'`、`'b'` 或空格。此外，你还可以在正则中放置注释；注释从 `#` 字符扩展到下一个换行符。当与三引号字符串一起使用时，这使正则的格式更加整齐：

```
pat = re.compile(r"""
\s*                # Skip leading whitespace
(?:P<header>[^\:]+) # Header name
\s* :              # Whitespace, and a colon
(?:P<value>.*?)     # The header's value -- *? used to
                    # lose the following trailing whitespace
\s*$              # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

这更具有可读性：

```
pat = re.compile(r"\s*(?:P<header>[^\:]+)\s*:(?:P<value>.*?)\s*$")
```

7 反馈

正则表达式是一个复杂的主题。这份文档是否有助于你理解它们？是否存在不清楚的部分，或者你遇到的问题未在此处涉及？如果是，请向作者发送改进建议。

关于正则表达式的最完整的书几乎肯定是由 O'Reilly 出版的 Jeffrey Friedl 的 *Mastering Regular Expressions*。不幸的是，它专注于 Perl 和 Java 的正则表达式，并且根本不包含任何 Python 材料，因此它不能用作 Python 编程的参考。（第一版涵盖了 Python 现在删除的 `regex` 模块，这对你没有多大帮助。）考虑从你的图书馆中查找它。