正则表达式 HOWTO

发布 *2.7.18*

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

作者 A.M. Kuchling <amk@amk.ca>

摘要

本文档是在 Python 中使用 re 模块使用正则表达式的入门教程。它提供了比“标准库参考”中相应部分更平和的介绍。

1 概述

The re module was added in Python 1.5, and provides Perl-style regular expression patterns. Earlier versions of Python came with the regex module, which provided Emacs-style patterns. The regex module was removed completely in Python 2.5.

正则表达式（称为 RE，或正则，或正则表达式模式）本质上是嵌入在 Python 中的一种微小的、高度专业化的编程语言，可通过 re 模块获得。使用这种小语言，你可以为要匹配的可能字符串集指定规则；此集可能包含英语句子，电子邮件地址，TeX 命令或你喜欢的任何内容。然后，您可以询问诸如“此字符串是否与模式匹配?”或“此字符串中的模式是否匹配?”等问题。你还可以使用正则修改字符串或以各种方式将其拆分。

正则表达式模式被编译成一系列字节码，然后由用 C 编写的匹配引擎执行。对于高级用途，可能需要特别注意引擎如何执行给定的正则，并将正则写入以某种方式生成运行速度更快的字节码。本文档未涉及优化，因为它要求你充分了解匹配引擎的内部结构。

正则表达式语言相对较小且受限制，因此并非所有可能的字符串处理任务都可以使用正则表达式完成。还有一些任务 可以用正则表达式完成，但表达式变得非常复杂。在这些情况下，你最好编写 Python 代码来进行处理；虽然 Python 代码比精心设计的正则表达式慢，但它也可能更容易理解。

2 简单模式

我们首先要了解最简单的正则表达式。由于正则表达式用于对字符串进行操作，因此我们将从最常见的任务开始：匹配字符。

有关正则表达式（确定性和非确定性有限自动机）的计算机科学的详细解释，你可以参考几乎所有有关编写编译器的教科书。

2.1 匹配字符

大多数字母和字符只会匹配自己。例如，正则表达式 test 将完全匹配字符串 test 。（你可以启用一个不区分大小写的模式，让这个正则匹配 Test 或 TEST，稍后会详细介绍。）

这条规则有例外；一些字符是特殊的 *metacharacters* ，并且不匹配自己。相反，它们表示应该匹配一些与众不同的东西，或者通过重复它们或改变它们的含义来影响正则的其他部分。本文档的大部分内容都致力于讨论各种元字符及其功能。

这是元字符的完整列表；它们的意思将在本 HOWTO 的其余部分讨论。

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

我们将看到的第一个元字符是 [`` 和]“。它们用于指定字符类，它是你希望匹配的一组字符。可以单独列出字符，也可以通过给出两个字符并用 '-' 标记将它们分开来表示一系列字符。例如，[abc] 将匹配任何字符 a、b 或 c; 这与 [a-c] 相同，它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，你的正则是 [a-z]。

字符类中的元字符不生效。例如，`[akm$]` 将匹配“’a’,’k’、’m’ 或’$’“ 中的任意字符；`'$'` 通常是一个元字符，但在一个字符类中它被剥夺了特殊性。

你可以通过以下方式匹配 *complementing* 设置的字符类中未列出的字符。这通过包含一个 `'^'` 作为该类的第一个字符来表示。例如，`[^5]` 将匹配除 `'5'` 之外的任何字符。如果插入符出现在字符类的其他位置，则它没有特殊含义。例如：`[5^]` 将匹配 `'5'` 或 `'^'`。

也许最重要的元字符是反斜杠，`\`。与 Python 字符串文字一样，反斜杠后面可以跟各种字符，以指示各种特殊序列。它也用于转义所有元字符，因此您仍然可以在模式中匹配它们；例如，如果你需要匹配 `[` 或 `\`，你可以在它们前面加一个反斜杠来移除它们的特殊含义：`\[` 或 `\\`。

Some of the special sequences beginning with `'\'` represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn’t whitespace. The following predefined special sequences are a subset of those available. The equivalent classes are for byte string patterns. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax.

`\d` 匹配任何十进制数字；这等价于类 `[0-9]`。

`\D` 匹配任何非数字字符；这等价于类 `[^0-9]`。

`\s` 匹配任何空白字符；这等价于类 `[ \t\n\r\f\v]`。

`\S` 匹配任何非空白字符；这相当于类 `[^ \t\n\r\f\v]`。

`\w` 匹配任何字母与数字字符；这相当于类 `[a-zA-Z0-9_]`。

`\W` 匹配任何非字母与数字字符；这相当于类 `[^a-zA-Z0-9_]`。

这些序列可以包含在字符类中。例如，`[\s,.]` 是一个匹配任何空格字符的字符类或者 `','`，或 `'.'`。

The final metacharacter in this section is `.`. It matches anything except a newline character, and there’s an alternate mode (`re.DOTALL`) where it will match even a newline. `'.'` is often used where you want to match “any character”.

2.2 重复

能够匹配不同的字符集合是正则表达式可以做的第一件事，这对于字符串可用方法来说是不可能的。但是，如果这是正则表达式的唯一额外功能，那么它们就不会有太大的优势。另一个功能是你可以指定正则的某些部分必须重复一定次数。

The first metacharacter for repeating things that we’ll look at is `*`. `*` doesn’t match the literal character `*`; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, `ca*t` will match `ct` (0 `a` characters), `cat` (1 `a`), `caaat` (3 `a` characters), and so forth. The RE engine has various internal limitations stemming from the size of C’s `int` type that will prevent it from matching over 2 billion `a` characters; you probably don’t have enough memory to construct a string that large, so you shouldn’t run into that limit.

类似 `*` 这样的重复是 *贪婪的*；当重复正则时，匹配引擎将尝试尽可能多地重复它。如果模式的后续部分不匹配，则匹配引擎将回退并以较少的重复次数再次尝试。

A step-by-step example will make this more obvious. Let’s consider the expression `a[bcd]*b`. This matches the letter `'a'`, zero or more letters from the class `[bcd]`, and finally ends with a `'b'`. Now imagine matching this RE against the string `abcbd`.

步骤	匹配	解释
1	a	正则中的 a 匹配。
2	abcbd	引擎尽可能多地匹配 [bcd]*，直到字符串结束。
3	失败	引擎尝试匹配 b，但是当前位置位于字符串结束，所以匹配失败。
4	abcb	回退一次，[bcd]* 少匹配一个字符。
5	失败	再次尝试匹配 b，但是当前位置是最后一个字符 'd'。
6	abc	再次回退，所以 [bcd]* 只匹配 bc。
6	abcb	再试一次 b。这次当前位置的字符是 'b'，所以它成功了。

The end of the RE has now been reached, and it has matched `abcb`. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for `[bcd]*`, and if that subsequently fails, the engine will conclude that the string doesn't match the RE at all.

Another repeating metacharacter is +, which matches one or more times. Pay careful attention to the difference between `*` and +; `*` matches *zero* or more times, so whatever's being repeated may not be present at all, while + requires at least *one* occurrence. To use a similar example, `ca+t` will match `cat` (1 `a`), `caaat` (3 `a`'s), but won't match `ct`.

There are two more repeating qualifiers. The question mark character, `?`, matches either once or zero times; you can think of it as marking something as being optional. For example, `home-?brew` matches either `homebrew` or `home-brew`.

The most complicated repeated qualifier is `{m,n}`, where *m* and *n* are decimal integers. This qualifier means there must be at least *m* repetitions, and at most *n*. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, or `a////b`, which has four.

You can omit either *m* or *n*; in that case, a reasonable value is assumed for the missing value. Omitting *m* is interpreted as a lower limit of 0, while omitting *n* results in an upper bound of infinity —actually, the upper bound is the 2-billion limit mentioned earlier, but that might as well be infinity.

还原论者的读者可能会注意到其他三个限定符都可以用这种表示法表达。`{0,}``与 ``*``相同，``{1,}``相当于 ``+，{0,1}``和¿'相同。最好使用 "*"，+``或¿'，只要因为它们更短更容易阅读。

3 使用正则表达式

现在我们已经看了一些简单的正则表达式，我们如何在 Python 中实际使用它们？`re` 模块提供了正则表达式引擎的接口，允许你将正则编译为对象，然后用它们进行匹配。

3.1 编译正则表达式

正则表达式被编译成模式对象，模式对象具有各种操作的方法，例如搜索模式匹配或执行字符串替换。:

```
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

`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，但现在只需一个例子

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

正则作为字符串传递给 `re.compile()` 。正则被处理为字符串，因为正则表达式不是核心 Python 语言的一部分，并且没有创建用于表达它们的特殊语法。（有些应用程序根本不需要正则，因此不需要通过包含它们来扩展语言规范。）相反，`re` 模块只是 Python 附带的 C 扩展模块，就类似于 `socket` 或 `zlib` 模块。

将正则放在字符串中可以使 Python 语言更简单，但有一个缺点是下一节的主题。

3.2 反斜杠灾难

如前所述，正则表达式使用反斜杠字符 (`'\'`) 来表示特殊形式或允许使用特殊字符而不调用它们的特殊含义。这与 Python 在字符串文字中用于相同目的的相同字符的使用相冲突。

假设你想要编写一个与字符串 `\section` 相匹配的正则，它可以在 LaTeX 文件中找到。要找出在程序代码中写入的内容，请从要匹配的字符串开始。接下来，您必须通过在反斜杠前面添加反斜杠和其他元字符，从而产生字符串 `\\section`。必须传递给 `re.compile()` 的结果字符串必须是 `\\section`。但是，要将其表示为 Python 字符串文字，必须 再次转义两个反斜杠。

字符	阶段
`\section`	被匹配的字符串
`\\section`	为 `re.compile()` 转义的反斜杠
`"\\\\section"`	为字符串字面转义的反斜杠

简而言之，要匹配文字反斜杠，必须将 `'\\\\'` 写为正则字符串，因为正则表达式必须是 `\\`，并且每个反斜杠必须表示为 `\\` 在常规 Python 字符串字面中。在反复使用反斜杠的正则中，这会导致大量重复的反斜杠，并使得生成的字符串难以理解。

解决方案是使用 Python 的原始字符串表示法来表示正则表达式；反斜杠不以任何特殊的方式处理前缀为 `'r'` 的字符串字面，因此 `r"\n"` 是一个包含 `'\'` 和 `'n'` 的双字符字符串，而 `"\n"` 是一个包含换行符的单字符字符串。正则表达式通常使用这种原始字符串表示法用 Python 代码编写。

常规字符串	原始字符串
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

3.3 应用匹配

一旦你有一个表示编译正则表达式的对象，你用它做什么？模式对象有几种方法和属性。这里只介绍最重要的内容；请参阅 `re` 文档获取完整列表。

方法 / 属性	目的
`match()`	确定正则是否从字符串的开头匹配。
`search()`	扫描字符串，查找此正则匹配的任何位置。
`findall()`	找到正则匹配的所有子字符串，并将它们作为列表返回。
`finditer()`	找到正则匹配的所有子字符串，并将它们返回为一个 iterator。

`match()` and `search()` return `None` if no match can be found. If they’re successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the `re` module. If you have Tkinter available, you may also want to look at Tools/scripts/redemo.py, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE. Phil Schwartz’s Kodos is also an interactive tool for developing and testing RE patterns.

本 HOWTO 使用标准 Python 解释器作为示例。首先，运行 Python 解释器，导入 re 模块，然后编译一个正则

```
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p  #doctest: +ELLIPSIS
<_sre.SRE_Pattern object at 0x...>
```

Now, you can try matching various strings against the RE `[a-z]+`. An empty string shouldn't match at all, since + means 'one or more repetitions'. `match()` should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of `match()` to make this clear.

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

Now, let's try it on a string that it should match, such as `tempo`. In this case, `match()` will return a match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object at 0x...>
```

Now you can query the match object for information about the matching string. match object instances also have several methods and attributes; the most important ones are:

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

尝试这些方法很快就会清楚它们的含义:

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

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
<_sre.SRE_Match object at 0x...>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

在实际程序中，最常见的样式是在变量中存储 匹配对象，然后检查它是否为 None。这通常看起来像:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings:

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

`findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator.[1]

```
>>> 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 模块级别函数

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as the corresponding pattern method, with the RE string added as the first argument, and still return either `None` or a match object instance.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<_sre.SRE_Match object at 0x...>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE are faster.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? That choice depends on how frequently the RE will be used, and on your personal coding style. If the RE is being used at only one point in the code, then the module functions are probably more convenient. If a program contains a lot of regular expressions, or re-uses the same ones in several locations, then it might be worthwhile to collect all the definitions in one place, in a section of code that compiles all the REs ahead of time. To take an example from the standard library, here's an extract from the deprecated `xmllib` module:

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

[1] Introduced in Python 2.2.2.

I generally prefer to work with the compiled object, even for one-time uses, but few people will be as much of a purist about this as I am.

3.5 编译标志

编译标志允许你修改正则表达式的工作方式。标志在 `re` 模块中有两个名称，长名称如 `IGNORECASE` 和一个简短的单字母形式，例如 `I`。（如果你熟悉 Perl 的模式修饰符，则单字母形式使用和其相同的字母；例如，`re.VERBOSE` 的缩写形式为 `re.X`。）多个标志可以通过按位或运算来指定它们；例如，`re.I | re.M` 设置 `I` 和 `M` 标志。

这是一个可用标志表，以及每个标志的更详细说明。

标志	含义
`DOTALL`, `S`	Make `.` match any character, including newlines
`IGNORECASE`, `I`	Do case-insensitive matches
`LOCALE`, `L`	Do a locale-aware match
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`
`VERBOSE`, `X`	启用详细的正则，可以更清晰，更容易理解。
`UNICODE`, `U`	Makes several escapes like `\w`, `\b`, `\s` and `\d` dependent on the Unicode character database.

I
IGNORECASE
Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match `Spam`, `spam`, or `spAM`. This lowercasing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

L
LOCALE
Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]`; it won't match `'é'` or `'ç'`. If your system is configured properly and a French locale is selected, certain C functions will tell the program that `'é'` should also be considered a letter. Setting the `LOCALE` flag when compiling a regular expression will cause the resulting compiled object to use these C functions for `\w`; this is slower, but also enables `\w+` to match French words as you'd expect.

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。

U
UNICODE
Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` dependent on the Unicode character properties database.

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
 (
     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。

| Alternation, or the "or" operator. If A and B are regular expressions, A|B will match any string that matches either A or B. | has very low precedence in order to make it work reasonably when you're alternating multi-character strings. Crow|Servo will match either Crow or Servo, not Cro, a 'w' or an 'S', and 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')
<_sre.SRE_Match object at 0x...>
>>> print re.search('^From', 'Reciting From Memory')
None
```

$ 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('}$', '{block}')
<_sre.SRE_Match object at 0x...>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<_sre.SRE_Match object at 0x...>
```

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')
<_sre.SRE_Match object at 0x...>
>>> 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')
<_sre.SRE_Match object at 0x...>
```

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

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a ':', like this:

```
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('abababababab').span()
(0, 10)
```

Groups indicated with '(', ')' also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to group(), start(), end(), and span(). Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> 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() can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

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

The groups() method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> 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的双字。：

```
>>> 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 added several additional features to standard regular expressions, and the Python `re` module supports most of them. It would have been difficult to choose new single-keystroke metacharacters or new special sequences beginning with `\` to represent the new features without making Perl's regular expressions confusingly different from standard REs. If you chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn't have escaped it by writing `\&` or `[&]`.

Perl 开发人员选择的解决方案是使用 `(?...)` 作为扩展语法。括号后面的 `?` 是一个语法错误，因为 `?` 没有什么可重复的，所以这并没有引入任何兼容性问题。紧跟在 `?` 之后的字符表示正在使用什么扩展名，所以 `(?=foo)` 是一个东西（一个正向的先行断言）和 `(?:foo)` 是其它东西（包含子表达式 `foo` 的非捕获组）。

Python adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a `P`, you know that it's an extension that's specific to Python. Currently there are two such extensions: `(?P<name>...)` defines a named group, and `(?P=name)` is a backreference to a named group. If future versions of Perl 5 add similar features using a different syntax, the `re` module will be changed to support the new syntax, while preserving the Python-specific syntax for compatibility's sake.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Since groups are numbered from left to right and a complex expression may use many groups, it can become difficult to keep track of the correct numbering. Modifying such a complex RE is annoying, too: insert a new group near the beginning and you change the numbers of everything that follows it.

Sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can replace the `...` with any other regular expression.

```
>>> 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。

The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. *name* is, obviously, the name of the group. Named groups also behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

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

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。用于查找双字的正则表达式，\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 前向断言

另一个零宽度断言是前向断言。前向断言以正面和负面形式提供，如下所示：

(?=…) 正向前向断言。如果包含的正则表达式，由 ... 表示，在当前位置成功匹配，则成功，否则失败。但是，一旦尝试了包含的表达式，匹配的引擎就不会前进；模式其余的部分会在在断言开始的地方尝试。

(?!…) 负向前向断言。这与积正向断言相反；如果包含的表达式在字符串中的当前位置 不匹配，则成功。

更具体一些，让我们看看前向是有用的情况。考虑一个简单的模式来匹配文件名并将其拆分为基本名称和扩展名，用 . 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：

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

Notice that the . needs to be treated specially because it's a metacharacter; I've put it inside a character class. Also notice the trailing $; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches foo.bar and autoexec.bat and sendmail.cf and 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 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。

负面前向消除了所有这些困扰：

`.*[.](?!bat$)[^.]*$` 负向前向意味着：如果表达式 `bat` 此时不匹配，请尝试其余的模式；如果 `bat$` 匹配，整个模式将失败。尾随的 `$` 是必需的，以确保允许像 `sample.batch` 这样的扩展只以 `bat` 开头的文件能通过。`[^.]*` 确保当文件名中有多个点时，模式有效。

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()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

5.1 分割字符串

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; `split()` only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level `re.split()` function, too.

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

5.2 搜索和替换

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

.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。

Here's a simple example of using the `sub()` method. It replaces colour names with the word `colour`:

```
>>> 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'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> 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)
```

Empty matches are replaced only when they're not adjacent to a previous match.

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

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\j` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

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控制。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。图案可以作为对象或字符串提供；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

Sometimes using the `re` module is a mistake. If you’re matching a fixed string, or a single character class, and you’re not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they’re usually much faster, because the implementation is a single small C loop that’s been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`’s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

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
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> 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一个特征必须是什么；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>
```

The RE matches the `'<'` in `<html>`, and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the `'<'` in `<html>` to the `'>'` in `</title>`, which isn't what you want.

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。

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

`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。

The most complete book on regular expressions is almost certainly Jeffrey Friedl’s Mastering Regular Expressions, published by O’Reilly. Unfortunately, it exclusively concentrates on Perl and Java’s flavours of regular expressions, and doesn’t contain any Python material at all, so it won’t be useful as a reference for programming in Python. (The first edition covered Python’s now-removed `regex` module, which won’t help you much.) Consider checking it out from your library.