正则表达式教程

零基础入门系列教程

作者: 魏明择

2025 年版

https://weimingze.com

目录

正则表达式（Python版本）

前言

正则表达式（Regular Expression）是一种用于描述文字规则的字符串。

正则表达式是一门非常通用的技术，无论你是在编写 Word 或者 Excel 文档时进行查找替换，还是用爬虫爬取页面提取数据，再或者解析网站的 URL 地址都离不开它。他是每个计算机使用者或开发者都必须了解的技术。

下面的课程当中，我们将使用 Python 语言来讲解正则表达式的规则。正则表达式不仅可以用在 Python 语言中，其他的编程语言和一些应用工具软件也可以使用正则表达式技术。

正则表达式的用途

1. 验证文本（如邮箱、电话号码格式）。
2. 搜索与替换（批量修改代码或文档）。
3. 提取数据（从日志或网页中抓取特定内容等）。

为什么学习正则表达式?

1. 文本数据处理已经成为大数据时代下的日常工作，而文本处理离不开正则表达式。
2. 对文本内容的查找，定位，提取的逻辑比较复杂的工作需要正则表达式。

正则表达式是由普通字符和特殊字符（也称为元字符：Meta Characters）组成的字符串，用以描述一定的字符串规则，比如：重复，位置等，来表达某类特定的字符串，进而匹配。

所谓特殊字符是指由有特殊含义的字符，用于构建复杂的匹配模式。

例如:

字符串 `"1[35678]\d{9}"` 就可以表示以 1 开头，第二个数字是3、5、6、7、8中的一个，第三个和之后的八个数字都是0~9之间的数字组成的11位移动电话号码。

这就是正则表达式的写法，其中的 `1` 是普通字符，`[35678]` 、`\d` 和 `{9}` 都是有语法含义的特殊字符。

另外字符串 `"^\w+@\w+\.\w+$"` 表示邮箱的命名规则，如：author@weimingze.com；weimz@gitee.com等。

正则表达式的优缺点

正则表达式的优点在于其灵活性，缺点是正则难以阅读，调试困难。

本课程目标

1. 掌握正则表达式各种特殊字符的用法。
2. 能够读懂常用正则表达式并能够编辑正则表达式来完成日常工作。
3. 能在 Word 和 Excel 等文档编写工具中使用正则表达式进行查找和替换。
4. 能在 Visual Studio Code 或 PyCharm 中使用正则表达式进行批量查找和替换。
5. 能在爬虫项目中解析 HTML 中的信息，并用正则表达式提取其中的信息。
6. 能在 Django 或 Flask 后端项目中正确匹配 URL 并提取信息。

前置课程

学习本课程之前，请先学习 《Python编程语言（基础篇）》 的内容。

本课程使用 Python 语言的进行效果演示。

下面我们来讲解正则表达式，来和我一起开始学习吧！

1. python 的 re 模块

python 的 re 模块是专门处理正则表达式的内置模块。本模块提供了正则表达式匹配操作。

这里面我们学习两个 re 模块中常用的函数：`re.findall()` 和 `re.sub()`。格式如下:

函数	说明
re.findall(pattern, string, flags=0)	返回 pattern 在 string 中的所有非重叠匹配，以字符串列表或字符串元组列表的形式。
re.sub(pattern, repl, string, count=0, flags=0)	返回通过使用 repl 替换在 string 最左边非重叠出现的 pattern 而获得的字符串。 如果样式没有找到，则不加改变地返回 string。

总结

1. re.findall() 函数用来从字符串 string 提取符合格式 pattern 的内容，并返回。

2. re.sub() 函数是替换字符串 string 将符合格式 pattern 的内容，替换成 repl 并返回替换后的字符串。

示例

```
>>> import re
>>> html = '''<h1>咏鹅</h1>
... <p>鹅，鹅，鹅，曲项向天歌。</p>
... <p>白毛浮绿水，红掌拨清波。</p>'''
>>> re.findall("鹅", html)
['鹅', '鹅', '鹅', '鹅']
>>> re.findall("鹅，", html)
['鹅，', '鹅，', '鹅，']
>>> re.sub("鹅", '鸭',  html)
'<h1>咏鸭</h1>\n<p>鸭，鸭，鸭，曲项向天歌。</p>\n<p>白毛浮绿水，红掌拨清波。</p>'
>>> r = re.sub("鹅", '鸭',  html)
>>> print(r)
<h1>咏鸭</h1>
<p>鸭，鸭，鸭，曲项向天歌。</p>
<p>白毛浮绿水，红掌拨清波。</p>
```

可见:

`re.findall("鹅", html)` 是将所有含有 `"鹅"` 这一个字的内容都提取出来，形成列表 `['鹅', '鹅', '鹅', '鹅']` 并返回。

`re.findall("鹅，", html)` 是将所有含有 `"鹅，"` 这一个字的内容都提取出来，形成列表 `['鹅，', '鹅，', '鹅，']` 并返回。

`re.sub("鹅", '鸭', html)` 是将 html 中所有的 `"鹅"` 都替换成了 `"鸭"` 并返回替换后的字符串。

2. 匹配重复次数的特殊字符

这一小节我们先来学习正则表达式当中用于匹配重复次数的特殊字符。这可以让我们尽快了解什么是正则表达式。

用于匹配重复次数的特殊字符如下：

特殊字符	说明
`*`	匹配前一个字符 **0 次、1 次或多次**。
`+`	匹配前一个字符 **1 次或多次**。
`?`	匹配前一个字符 **0 次或 1 次**。
`{n}`	匹配前一个字符 **固定 n 次**。
`{m,}`	匹配前一个字符 **至少 m 次**。
`{,n}`	匹配前一个字符 **最多 n 次**。
`{m,n}`	匹配前一个字符 **m 到 n 次**。

下面用示例说明上述特殊字符的用法。

先来创建一个字符串 s1 用来绑定一个字符串如下：。

```
>>> import re  # 导入 python 的 re 模块
>>> s1 = 'weewiweiweeiweeeiweeeeimingze.com'
```

请注意观察 s1 绑定的字符串。

使用普通字符匹配

目标1:

找出以 `'w'` 开头，后面跟有两个 `'e'` 的全部的字符串，并返回结果。

结果如下：

```
>>> re.findall(r'wee', s1)
['wee', 'wee', 'wee', 'wee']
```

在 s1 字符串中一共找到四个，并以列表的形式返回结果。

使用特殊字符匹配重复次数

目标2:

找出以 `'w'` 开头，以 `'i'` 结尾，中间有0个、1个或多个 `'e'` 的全部的字符串，并返回结果。

这时候我们可以使用 `'*'` 这个特殊字符。`'*'` 匹配前面的字符出现0次、1次或多次。

结果如下：

```
>>> re.findall(r'we*i', s1)
['wi', 'wei', 'weei', 'weeei', 'weeeei']
```

在 s1 字符串中一共找到五个。第一个 `'wi'` 中间没有 `'e'` 也可以匹配。

目标3:

找出以 `'w'` 开头，以 `'i'` 结尾，中间有1个或多个 `'e'` 的全部的字符串，并返回结果。

这时候我们可以使用 `'+'` 这个特殊字符。`'+'` 匹配前面的字符出现1次或多次。

结果如下：

```
>>> re.findall(r'we+i', s1)
['wei', 'weei', 'weeei', 'weeeei']
```

在 s1 字符串中一共找到四个。第一个是 `'wei'`。此示例中`'wee'` 和 `'wi'`没有匹配成功。

目标4:

找出以 `'w'` 开头，以 `'i'` 结尾，中间有0个或1个 `'e'` 的全部的字符串，并返回结果。

这时候我们可以使用 `'?'` 这个特殊字符。`'?'` 匹配前面的字符出现0次或1次。

结果如下：

```
>>> re.findall(r'we?i', s1)
['wi', 'wei']
```

在 s1 字符串中一共找到两个。第一个是 `'wi'`，第二个是 `'wei'`。其他没有匹配成功。

目标5:

找出以 `'w'` 开头，以 `'i'` 结尾，中间有3个 `'e'` 的全部的字符串，并返回结果。

这时候我们可以使用 `'{n}'` 这个特殊字符的组合。`'{n}'` 匹配前面的字符出现固定的n次。

结果如下：

```
>>> re.findall(r'we{3}i', s1)
['weeei']
```

在 s1 字符串中一共找到1个`'weeei'`。

当让上述匹配样式`'we{3}i'`也可以写成`r'weeei'`，都是同样的效果。当使用`'{n}'`这种方式会更灵活，并可以匹配字符集合（下一节讲解）。

目标6:

找出以`'w'`开头，以`'i'`结尾，中间有1个到3个`'e'`的全部的字符串，并返回结果。

这时候我们可以使用`'{m,n}'`这个特殊字符的组合。`'{m,n}'`匹配前面的字符出现最少n次，最多m次的匹配（包含n和m）。

> 注意: 表达式`'{m,n}'`中间逗号后面没有空格。这个语法非常严格。

结果如下:

```
>>> re.findall(r'we{1,3}i', s1)
['wei', 'weei', 'weeei']
```

在 s1 字符串中一共找到3个符合条件的结果。

目标7:

找出以`'w'`开头，以`'i'`结尾，中间有2个或以上个`'e'`的全部的字符串，并返回结果。

这时候我们可以使用`'{m,}'`这个特殊字符的组合。`'{m,}'`匹配前面的字符出现最少n次，最多不限制次数的匹配。

结果如下:

```
>>> re.findall(r'we{2,}i', s1)
['weei', 'weeei', 'weeeei']
```

在 s1 字符串中一共找到3个符合条件的结果。

目标8:

找出以`'w'`开头，以`'i'`结尾，中间有0个到3个`'e'`的全部的字符串，并返回结果。

这时候我们可以使用`'{,n}'`这个特殊字符的组合。`'{,n}'`匹配前面的字符出现最少0次，最多n次的匹配（包含n）。

结果如下:

```
>>> re.findall(r'we{,3}i', s1)
['wi', 'wei', 'weei', 'weeei']
```

在 s1 字符串中一共找到4个符合条件的结果。 其中 `'wi'` 也能成功匹配并找到了。

3. 匹配一个字符集合的特殊字符

用于匹配一个字符集合的特殊字符如下：

特殊字符	说明
`[abc]`	匹配 `a`、`b` 或 `c` 中的任意一个字符。
`[^abc]`	匹配**不在** `a`、`b`、`c` 中的任意字符（否定匹配）。
`[a-z]`	匹配任意小写字母（范围匹配）。
`[0-9A-F]`	匹配十六进制字符（组合范围）。
`.`	匹配**任意单个字符**（除换行符 `'\n'` 外，除非启用 `DOTALL` 模式）。
`\d`	匹配**数字**（等价于 `[0-9]`）。
`\D`	匹配**非数字**（等价于 `[^0-9]`）。
`\w`	匹配**单词字符**（字母、数字、下划线以及中文，等价于 `[a-zA-Z0-9_]`）。
`\W`	匹配**非单词字符**（如标点、空格等）。
`\s`	匹配**空白字符**（空格、换行符 `'\n'`、回车符`'\r'`、水平制表符 `'\t'`、垂直制表符 `'\v'`、 换页符 `'\f'` 等不可见的控制字符）。
`\S`	匹配**非空白字符**。

下面用示例说明上述特殊字符的用法。

先来创建一个字符串 s2 用来绑定一个字符串如下：

```
>>> import re
>>> s2 = 'wazwbzwcz wdz wmz w1z w8z w666z w z wAz'
```

请注意观察 s2 绑定的字符串。

目标1:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 `'a'` 或 `'d'` 的全部的字符串，并返回结果。

这时候我们可以使用 `[字符集]` 这个特殊字符的组合。字符集可以是任意的普通字符。表示当前原字符串中要匹配的字符是字符集中的字符之一则匹配成功。

结果如下：

```
>>> re.findall(r'w[ad]z', s2)
['waz', 'wdz']
```

`[ad]` 表示 字符 `'a'` 或者 字符 `'d'`，所以匹配的结果是两个：'waz' 和 'wdz'。

目标2:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 `'a'`、`'b'`、`'c'` 或 `'d'` 的全部的字符串，并返回结果。

这时候我们可以使用 `[abcd]` 这个字符集进行匹配。

结果如下：

```
>>> re.findall(r'w[abcd]z', s2)
['waz', 'wbz', 'wcz', 'wdz']
```

`[abcd]` 表示 字符 `'a'`、`'b'`、`'c'` 或 `'d'`，这种情况可以在中括号中使用 `-` 号，表示从左侧字符开始到侧字符结束。如下：

```
>>> re.findall(r'w[a-d]z', s2)
['waz', 'wbz', 'wcz', 'wdz']
```

> 注意：`[a-]` 表示 `'a'` 或 `'-'` 两个字符；`[-d]` 表示 `'-'` 或`'d'` 两个字符；必须 `-` 减号两侧都有字符，且左侧的字符编码小于右侧字符编码值时 `-` 才表示区间的特殊字符。

目标3:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 数字 或 空格 的全部的字符串，并返回结果。

这时候我们可以使用 `[0-9 ]` 这个字符集进行匹配。

结果如下：

```
>>> re.findall(r'w[0-9 ]z', s2)
['w1z', 'w8z', 'w z']
```

匹配到三个结果。

目标4:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 数字 或 小写英文字母 的全部的字符串，并返回结果。

这时候我们可以使用 `[0-9a-z]` 这个字符集进行匹配。

结果如下：

```
>>> re.findall(r'w[0-9a-z]z', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w1z', 'w8z']
```

目标5:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是不是 数字 也不是 小写英文字母 的全部的字符串，并返回结果。

这时候我们可以使用 `[^0-9a-z]` 这个字符集进行匹配。 在 中括号 `[]` 中，如果字符集的第一个是 `^` 字符则表示不包括内部的字符集的其他字符集。

结果如下：

```
>>> re.findall(r'w[^0-9a-z]z', s2)
['w z', 'wAz']
```

目标6:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是不是 数字 也不是 小写英文字母 的全部的字符串，并返回结果。

这时候我们可以使用 `[^0-9a-z]` 这个字符集进行匹配。 在 中括号 `[]` 中，如果字符集的第一个是 `^` 字符则表示不包括内部的字符集的其他字符集。

结果如下：

```
>>> re.findall(r'w[^0-9a-z]z', s2)
['w z', 'wAz']
```

目标7:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 任意字符 的全部的字符串，并返回结果。

这时候我们可以使用 `.` 这个特殊字符进行匹配。 `.` 用来匹配不包括 换行符 `'\n'` 在内的全部字符，如果要包含 换行符需要使用 `DOTALL` 匹配模式。

结果如下：

```
>>> re.findall(r'w.z', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w1z', 'w8z', 'w z', 'wAz']
```

再看一个用 `.` 匹配全部字符的例子

```
>>> re.findall(r'w.z', 'waz wbz wcz w\nz w z wmz')
['waz', 'wbz', 'wcz', 'w z', 'wmz']
>>> re.findall(r'w.z', 'waz wbz wcz w\nz w z wmz', flags=re.DOTALL)
['waz', 'wbz', 'wcz', 'w\nz', 'w z', 'wmz']
```

当 re.findall() 函数的第三个参数传入 `re.DOTALL` 或 `re.S` 时，此时的 `.` 可以匹配任意一个字符。

此时 `'w\nz'` 能够被成功匹配。

目标8:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 数字 的全部的字符串，并返回结果。

这时候我们可以使用 `[0-9]` 进行匹配，也可以使用 `\d` 来匹配,或者使用 `[\d]` 进行匹配。

结果如下：

```
>>> re.findall(r'w\dz', s2)
['w1z', 'w8z']
>>> re.findall(r'w[0-9]z', s2)
['w1z', 'w8z']
>>> re.findall(r'w[\d]z', s2)
['w1z', 'w8z']
```

三者结果相同。

目标9:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符不是 数字 的全部的字符串，并返回结果。

这时候我们可以使用 `[^0-9]` 进行匹配，也可以使用 `\D` 来匹配，或者使用 `[\D]` 进行匹配。

结果如下：

```
>>> re.findall(r'w[^0-9]z', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w z', 'wAz']
>>> re.findall(r'w\Dz', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w z', 'wAz']
>>> re.findall(r'w[\D]z', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w z', 'wAz']
```

目标10:

找出字符串 s2 中的全部数字的字符串，并返回结果。

这时候我们可以使用 `[0-9]+` 进行匹配，也可以使用 `\d+` 进行匹配。

结果如下：

```
>>> re.findall(r'[0-9]+', s2)
['1', '8', '666']
>>> re.findall(r'\d+', s2)
['1', '8', '666']
>>> re.findall(r'[\d]+', s2)
['1', '8', '666']
```

目标11:

找出字符串 s2 中以 `'w'` 开头，以 `'z'` 结尾，中间的一个字符是 空白字符 的全部的字符串，并返回结果。

这时候我们可以使用 `[ \r\n\t\v\f]` 进行匹配，也可以使用 `\s` 来匹配。`\s`用来匹配空白字符，包括`[ \r\n\t\v\f]`但不限于这些。

结果如下：

```
>>> re.findall(r'w\sz', s2)
['w z']
>>> re.findall(r'w[ \r\n\t\v\f]z', s2)
['w z']
```

特殊字符 `\S` 匹配非空白字符，等同于 [^\s]， 就是空白字符以外的字符。

特殊字符 `\w` 匹配英文字母（大小写）、数字、中文字符等不包括标点符合等的字符。

特殊字符 `\W` 等同于 `[^\w]`。

4. 贪婪匹配和非贪婪匹配

贪婪匹配(默认)

默认的 `*`、 `+`、 `?` 和 `{m,n}` 数量限定符都是 贪婪的，所谓贪婪就是在整个表达式匹配成功的前提下，尽可能多的匹配。

非贪婪匹配

在原有的 `*`、 `+`、 `?` 和 `{m,n}` 数量限定符的后面加一个问号 `?`，则为非贪婪的匹配方式。所谓非贪婪就是尽可能少的匹配。

非贪婪匹配的 数量限定符表示为: `*?`、 `+?`、 `??` 和 `{m,n}?`。

示例

先准备一个字符串 s3，如下：

```
>>> import re
>>> s3 = '<h1>咏鹅</h1><p>鹅，鹅，鹅，曲项向天歌。</p><p>白毛浮绿水，红掌拨清波。</p>'
```

使用贪婪匹配方式进行匹配

```
>>> re.findall(r'<p>.*</p>', s3)
['<p>鹅，鹅，鹅，曲项向天歌。</p><p>白毛浮绿水，红掌拨清波。</p>']
```

可见贪婪匹配方式 只匹配到一个结果为：`'<p>鹅，鹅，鹅，曲项向天歌。</p><p>白毛浮绿水，红掌拨清波。</p>'`。

下面再使用非贪婪匹配方式进行匹配

```
>>> re.findall(r'<p>.*?</p>', s3)
['<p>鹅，鹅，鹅，曲项向天歌。</p>', '<p>白毛浮绿水，红掌拨清波。</p>']
```

可见非贪婪匹配方式 匹配到了两个结果为：`'<p>鹅，鹅，鹅，曲项向天歌。</p>'` 和 `'<p>白毛浮绿水，红掌拨清波。</p>'`。

5. 正则表达式分组

正则表达式的分组（Grouping）是指将一部分子模式用括号 () 包围起来，使其成为一个独立的单元。

分组示例

先来创建一个字符串 s2 用来绑定一个字符串如下：

```
>>> import re
>>> s2 = 'wazwbzwcz wdz wmz w1z w8z w666z w z wAz'
```

使用分组提取 'w' 开头，'z' 结尾，中间是一个任意字符的字符串中间的那个一个字符。

```
>>> re.findall(r'w.z', s2)
['waz', 'wbz', 'wcz', 'wdz', 'wmz', 'w1z', 'w8z', 'w z', 'wAz']
>>> re.findall(r'w(.)z', s2)
['a', 'b', 'c', 'd', 'm', '1', '8', ' ', 'A']
```

可见 使用 `w.z` 时，没有分组返回的是包含 `'w'` 和 `'z'` 在内的所有匹配的三个字符。而使用 `w(.)z` 分组时返回的是组内匹配的1个字符。

接下来我们使用分组提取 `'w'` 开头，`'z'` 结尾，中间是一个任意字符的字符串首字母w和中间的那个一个字符组成的所有的字符串。

```
>>> re.findall(r'(w.)z', s2)
['wa', 'wb', 'wc', 'wd', 'wm', 'w1', 'w8', 'w ', 'wA']
```

此时 组内的内容 `(w.)` 匹配的内容都提取出来了。

当有两个或两个以上分组的情况

当然一个正则表达式中用两个或两个以上分组时，re.findall() 返回类的列表内是元组，每一个元组对应着一次成功匹配的数据，数据按分组的先后顺序摆放。

示例

```
>>> import re
>>> s3 = '<h1>咏鹅</h1><p>鹅，鹅，鹅，曲项向天歌。</p><p>白毛浮绿水，红掌拨清波。</p>'
```

我们提取 HTML 标签开始（如`<h1>`）,标签内部的内容（如:咏鹅）和标签的结束标志（如：`</h1>`）

```
>>> re.findall(r'<(.*?)>(.*?)<(.*?)>', s3)
[('h1', '咏鹅', '/h1'), ('p', '鹅，鹅，鹅，曲项向天歌。', '/p'), ('p', '白毛浮绿水，红掌拨清波。', '/p')]
```

上述正则表达式 `r'<(.*?)>(.*?)<(.*?)>'` 中有三个分组, 而返回的列表中也不再是字符串，而是元组，每个元组内有三个数据，对应分组的先后顺序。

反向引用

在一个正则表达式当中，如果实现了分组，那么每一组将有一个编号，这个编号是从1开始的，第一个组匹配的内容是1，第二个组匹配的内容是2，我们可以用这个编号反向引用原来的内容。

下列正则表达式中，正则表达式 `r'<(.*?)>(.*?)<(.*?)>'` 有三个组，则第一个组匹配的内容是 `\1`, 第二个组的内容为 `\2`, 以此类推。

```
>>> s3 = '<h1>咏鹅</h1><p>鹅，鹅，鹅，曲项向天歌。</p><p>白毛浮绿水，红掌拨清波。</p>'
>>> re.findall(r'<(.*?)>(.*?)<(.*?)>', s3)
[('h1', '咏鹅', '/h1'), ('p', '鹅，鹅，鹅，曲项向天歌。', '/p'), ('p', '白毛浮绿水，红掌拨清波。', '/p')]
```

下面我们使用反向引用，将 s3 字符串的 `<h1>` 标签和 `<p>` 标签都替换成 `<h4>` 标签，其他原标签内的内容不变。

注意：内容部分对应第二个分组，即 `\2`对应的内容

替换后结果如下：

```
>>> re.sub(r'<(.*?)>(.*?)<(.*?)>', r'<h4>\2<h4>', s3)
'<h4>咏鹅<h4><h4>鹅，鹅，鹅，曲项向天歌。<h4><h4>白毛浮绿水，红掌拨清波。<h4>'
```

注意: `r'<h4>\2<h4>'` 中的 `\2` 对应原来的内容。`\1` 和 `\3` 在此处没有使用。

命名分组

用 `(?P<name>...)` 定义分组，通过名称（而非数字）引用分组，提高可读性。

```
>>> re.findall(r'<(.*?)>(?P<content>.*?)<(.*?)>', s3)
[('h1', '咏鹅', '/h1'), ('p', '鹅，鹅，鹅，曲项向天歌。', '/p'), ('p', '白毛浮绿水，红掌拨清波。', '/p')]
```

上述正则 `r'<(.*?)>(?P<content>.*?)<(.*?)>` 中，我们将第二个分组取了名字为 `content`, 后面可以使用这个名称进行反向引用。

反向引用时的语法格式是 `\g<分组名称>`， 如 `'\g<content>'`。

我们将 s3 字符串的 `<h1>` 标签和 `<p>` 标签都替换成 `<h4>` 标签，其他原标签内的内容不变。这次我们使用命名分组。结果如下：

```
>>> re.sub(r'<(.*?)>(?P<content>.*?)<(.*?)>', r'<h4>\g<content></h4>', s3)
'<h4>咏鹅</h4><h4>鹅，鹅，鹅，曲项向天歌。</h4><h4>白毛浮绿水，红掌拨清波。</h4>'
```

至此，你已经学习了正则表达式最实用的语法了，下一节我们再学习一下附加的语法规则。

6. 转义字符

在正则表达式中会用到很多特殊字符，当我们将这些特殊字符当成普通字符来用的时候就需要转义。

正则中的转义符号和 Python 中的转义符号相同，都是反斜杠 `\`

比如当我们匹配一个网站，他的名字是weimingze.com 和 weimingze_com 这时候要能正确的匹配到 域名中的 `.` 此时正则中要使用 `\.` 来将特殊字符 `.` 转义成为普通的 点`.`

示例:

```
>>> s4 = 'weimingze.com; weimingze_com'
>>> re.findall(r'\w+.\w+', s4)
['weimingze.com', 'weimingze_com']
>>>
>>> re.findall(r'\w+\.\w+', s4)
['weimingze.com']
```

可见 在正则表达式 `r'\w+.\w+'` 中，`.` 可以匹配任意字符， 在正则表达式 `r'\w+\.\w+'` 中，`.` 只能匹配 `.` 这个字符。

需要转义的字符

正则中的常用的特殊字符在当成普通字符使用时都需要转义。这些字符有

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

部分需要转义的符号（要看应用的场景）

```
- ,
```

部分特殊字符

```
\s  \S
\w  \W
\d  \D
```

```
\b  \B
\n  \t  \v  \f  \r  \\(反斜杠自身)
```

7. 边界匹配

边界匹配用于精确控制匹配的位置，而不仅仅是内容。

它的作用是避免误匹配，确保正则表达式只在特定位置（如字符串开头、结尾或单词边界）生效，从而提升匹配的准确性。

边界匹配的特殊字符

特殊字符	说明
`^`	匹配字符串的**开头**（多行模式下匹配行首）。
`$`	匹配字符串的**结尾**（多行模式下匹配行尾）。
`\b`	匹配**单词边界**（即 \w 和 \W 之间的位置）。
`\B`	匹配**非单词边界**（如单词内部的连续字母）。

示例

```
>>> s5 = '''line 1
... this is line 2
... line 3
... line4
... last line 5'''
>>> re.findall(r'^line.*', s5, flags=re.MULTILINE)
['line 1', 'line 3', 'line4']
```

上述示例中S5绑定的字符串内部有换行。我们使用多行模式 `re.MULTILINE` 进行匹配。可见 `r'^line.*'` 只匹配了 line开头的行。

```
>>> re.findall(r'^line.*[1-3]$', s5, flags=re.MULTILINE)
['line 1', 'line 3']
```

上述示例中可见 `r'^line.*[1-3]$'` 只匹配了 line开头，并且以 数字 1-3结尾的行。`$` 匹配行尾，`^` 匹配行首。

单词边界示例

```
>>> s5 = '''line 1
... this is line 2
... line 3
... line4
... last line 5'''
>>> re.findall(r'.*line\b.*', s5, flags=re.MULTILINE)
['line 1', 'this is line 2', 'line 3', 'last line 5']
```

上述示例 正则表达式 `r'.*line\b.*'` 匹配 line 后是单词边界的行，其中第四行 `line4` 因为 `4` 和 `line` 相连，因此匹配失败。

```
>>> re.findall(r'.*line\B.*', s5, flags=re.MULTILINE)
['line4']
```

上述示例 正则表达式 `'.*line\B.*'` 匹配 line 后不是单词边界的行， `\B` 匹配非单词边界。

祝各位学员学习愉快!