关于作者

廖雪峰 V 北京 都职区
★ 编程王

正则表达式

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取 `@` 前后的子串，再分别判断是否是单词和域名，但这样都不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为"匹配了"，否则，该字符串就是不合法的。

所以现在判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符串描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用 `\d` 可以匹配一个数字，`\w` 可以匹配一个字母或数字，所以：

- `'00\d'` 可以匹配 `'007'`，但无法匹配 `'00A'`；
- `'\d\d\d'` 可以匹配 `'010'`；
- `'\w\w\d'` 可以匹配 `'py3'`；
- `.` 可以匹配任意字符，所以：
- `'py.'` 可以匹配 `'pyc'`、`'py#'`、`'py!'` 等等。

要匹配变长的字符，在正则表达式中，用 `*` 表示任意个字符（包括0个），用 `+` 表示至少一个字符，用 `?` 表示0个或1个字符，用 `{n}` 表示n个字符，用 `{n,m}` 表示n-m个字符：

来看一个复杂的例子：`\d{3}\s+\d{3,8}`。

我们来从左到右解读一下：

1. `\d{3}` 表示匹配3个数字，例如 `'010'`；
2. `\s` 可以匹配一个空格（也包括Tab等空白符），所以 `\s+` 表示至少有一个空格，例如匹配 `' '`，`'  '` 等；
3. `\d{3,8}` 表示3-8个数字，例如 `'1234567'`。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配 `'010-12345'` 这样的号码呢？由于 `'-'` 是特殊字符，在正则表达式中，要用 `\` 转义，所以，上面的正则是 `\d{3}-\d{3,8}`。

但是，仍然无法匹配 `'010 - 12345'`，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用 `[]` 表示范围，比如：

- `[0-9a-zA-Z_]` 可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z_]+` 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如：`'a100'`，`'0_Z'`，`'Py3000'` 等等；
- `[a-zA-Z_][0-9a-zA-Z_]*` 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- `[a-zA-Z_][0-9a-zA-Z_]{0, 19}` 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B` 可以匹配A或B，所以 `(P|p)ython` 可以匹配 `'Python'` 或者 `'python'`。

`^` 表示行的开头，`^\d` 表示必须以数字开头。

`$` 表示行的结束，`\d$` 表示必须以数字结束。

你可能注意到了，`py` 也可以匹配 `'python'`，但是加上 `^py$` 就全成了整行匹配，就只能匹配 `'py'` 了。

re模块

有了准备知识，我们就可以在Python中使用正则表达式了。Python提供 `re` 模块，包含所有正则表达式的功能。由于Python的字符串本身也用 `\` 转义，所以要特别注意：

```
s = 'ABC\\-001' # Python的字符串
# 对应的正则表达式字符串变成：
# 'ABC\-001'
```

因此我们强烈建议使用Python的 `r` 前缀，就不用考虑转义的问题了：

```
s = r'ABC\-001' # Python的字符串
# 对应的正则表达式字符串不变：
# 'ABC\-001'
```

先看看如何判断正则表达式是否匹配：

```
>>> import re
>>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> re.match(r'^\d{3}\-\d{3,8}$', '010 12345')
>>>
```

`match()` 方法判断是否匹配，如果匹配成功，返回一个 `Match` 对象，否则返回 `None`。常见的判断方法就是：

```
test = '用户输入的字符串'
if re.match(r'正则表达式', test):
    print('ok')
else:
    print('failed')
```

切分字符串

用正则表达式切分字符串比用固定的字符要灵活，请看正常的切分代码：

```
>>> 'a b   c'.split(' ')
['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
>>> re.split(r'\s', 'a b   c')
['a', 'b', '', '', 'c']
```

无论多少个空格都可以正常分割。加入 `,` 试试：

```
>>> re.split(r'[\s,]', 'a,b, c  d')
['a', 'b', '', 'c', '', 'd']
```

再加入 `;` 试试：

```
>>> re.split(r'[\s,;]', 'a,b;; c  d')
['a', 'b', '', '', 'c', '', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用 `()` 表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$` 分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
>>> m = re.match(r'^(\d{3})-(\d{3,8})$', '010-12345')
>>> m
<_sre.SRE_Match object; span=(0, 9), match='010-12345'>
>>> m.group(0)
'010-12345'
>>> m.group(1)
'010'
>>> m.group(2)
'12345'
```

如果正则表达式中定义了组，就可以在 `Match` 对象上用 `group()` 方法提取出子串来。

注意到 `group(0)` 永远是原始字符串，`group(1)`、`group(2)`……表示第1、2、……个子串。

提取子串非常有用。来看一个更复杂的例子：

```
>>> t = '19:05:30'
>>> m = re.match(r'^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9])$', t)
>>> m.groups()
('19', '05', '30')
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
'^(0[1-9]|[12][0-9]|3[01])-(0[1-9]|1[0-2])$'
```

对于 `'2-30'`，`'4-31'` 这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

最后需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的 `0`：

```
>>> re.match(r'^(\d+)(0*)$', '102300').groups()
('102300', '')
```

由于 `\d+` 采用贪婪匹配，直接把后面的 `0` 全部匹配了，结果 `0*` 只能匹配空字符串了。

必须让 `\d+` 采用非贪婪匹配（也就是尽可能少匹配），才能把后面的 `0` 匹配出来，加个 `?` 就可以让 `\d+` 采用非贪婪匹配：

```
>>> re.match(r'^(\d+?)(0*)$', '102300').groups()
('1023', '00')
```

编译

当我们在Python中使用正则表达式时，re模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；
2. 用编译后的正则表达式去匹配字符串。

如果一个正则表达式要重复使用几千次，出于效率的考虑，我们可以预编译该正则表达式，接下来重复使用时就不需要编译这个步骤了，直接匹配：

```
>>> import re
# 编译:
>>> re_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')
# 使用：
>>> re_telephone.match('010-12345').groups()
('010', '12345')
>>> re_telephone.match('010-8086').groups()
('010', '8086')
```

编译后生成Regular Expression对象，由于该对象自己包含了正则表达式，所以调用对应的方法时不用给出正则字符串。

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要详细地了解正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

- someone@gmail.com
- bill.gates@microsoft.com

```
# -*- coding: utf-8 -*-
import re

def is_valid_email(addr):
    return True

# 测试:
assert is_valid_email('someone@gmail.com')
assert is_valid_email('bill.gates@microsoft.com')
assert not is_valid_email('bob@example.com')
assert not is_valid_email('mr-bob@example.com')
print('ok')
```

▶ Run

版本二可以提取出带名字的Email地址：

- <Tom Paris> tom@voyager.org => Tom Paris
- bob@example.com => bob

```
# -*- coding: utf-8 -*-
import re

def name_of_email(addr):
    return None

# 测试:
assert name_of_email('<Tom Paris> tom@voyager.org') == 'Tom Paris'
assert name_of_email('tom@voyager.org') == 'tom'
print('ok')
```

▶ Run

参考源码

regex.py

评论

第二期
1 不错哦！ created at 2 days ago, Last updated at 2 days ago

```
def name_of_email(addrss):
    return re.match(r'^([^<]+)?(<)?(\w+@\w+\.\w+)(>)?', addrss)[1]
```

编辑 全部评论 回复

lw 匹配 内容需要更正
刚刚回 created at 2 days ago, Last updated at 2 days ago