

正则表达式

正则表达式简介

概念

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑（可以用来做检索，截取或者替换操作）。

正则表达式用于搜索、替换和解析字符串。正则表达式遵循一定的语法规则，使用非常灵活，功能强大。使用正则表达式编写一些逻辑验证非常方便，例如电子邮件地址格式的验证。

正则表达式是对字符串（包括普通字符（例如，a 到 z 之间的字母）和特殊字符）操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑，正则表达式是一种文本模式，模式描述在搜索文本时要匹配一个或多个字符串。

作用

- 1.给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）。
- 2.可以通过正则表达式，从字符串中获取我们想要的特定部分。
- 3.还可以对目标字符串进行替换操作。

正则表达式的使用

Python 语言通过标准库中的 re 模块支持正则表达式。re 模块提供了一些根据正则表达式进行查找、替换、分隔字符串的函数，这些函数使用一个正则表达式作为第一个参数。re 模块常用的函数如下表所示。

re 模块常用的函数

函数	描述
match(pattern,string,flags=0)	根据 pattern 从 string 的头部开始匹配字符串，只返回第 1 次匹配成功的对象；否则，返回 None
findall(pattern,string,flags=0)	根据 pattern 在 string 中匹配字符串。如果匹配成功，返回包含匹配结果的列表；否则，返回空列表。当 pattern 中有分组时，返回包含多个元组的列表，每个元组对应 1 个分组。flags 表示规则选项，规则选项用于辅助匹配。
sub(pattern,repl,string,count=0)	根据指定的正则表达式，替换源字符串中的子串。pattern 是一个正则表达式，repl 是用于替换的字符串，string 是源字符串。如果 count 等于 0，则返回 string 中匹配的所有结果；如果 count 大于 0，则返回前 count 个匹配

	结果
subn(pattern,repl,string,count=0)	作用和 sub()相同，返回一个二元的元组。第 1 个元素是替换结果，第 2 个元素是替换的次数
search(pattern,string,flags=0)	根据 pattern 在 string 中匹配字符串，只返回第 1 次匹配成功的对象。如果匹配失败，返回 None
compile(pattern,flags=0)	编译正则表达式 pattern，返回 1 个 pattern 的对象
split(pattern,string,maxsplit=0)	根据 pattern 分隔 string，maxsplit 表示最大的分隔数
escape(pattern)	匹配字符串中的特殊字符，如*、+、?等

match 方法

re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回 None。语法格式如下：

```
re.match(pattern, string, flags=0)
```

函数参数说明如下表所示：

match 函数参数说明

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。如下表列出正则表达式修饰符 - 可选标志

正则表达式修饰符 - 可选标志

修饰符	描述
re.I	使匹配对大小写不敏感
re.L	做本地化识别（locale-aware）匹配
re.M	多行匹配，影响 ^ 和 \$
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据 Unicode 字符集解析字符。这个标志影响 \w, \W, \b, \B.
re.X	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。

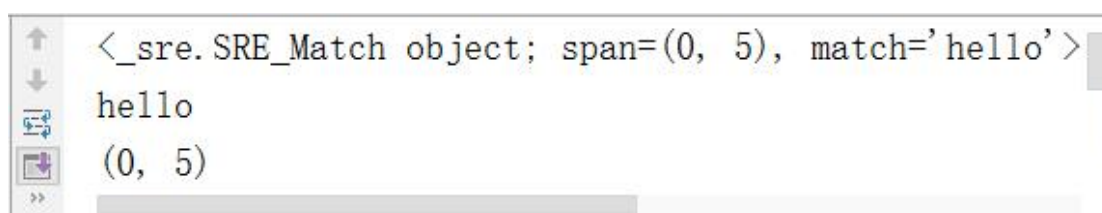
匹配字符串是正则表达式中最常用的一类应用。也就是设定一个文本模式，然后判断另外一个字符串是否符合这个文本模式。

如果文本模式只是一个普通的字符串，那么待匹配的字符串和文本模式字符串在完全相等的情况下，match 方法会认为匹配成功。如果匹配成功，则 match 方法返回匹配的对象，然后可以调用对象中的 group 方法获取匹配成功的字符串，如果文本模式就是一个普通的字符串，那么 group 方法返回的就是文本模式字符串本身。

【示例】match 方法的使用

```
import re
s='hello python'
pattern='hello'
v=re.match(pattern,s)
print(v)
print(v.group())
print(v.span())
```

执行结果如图所示：



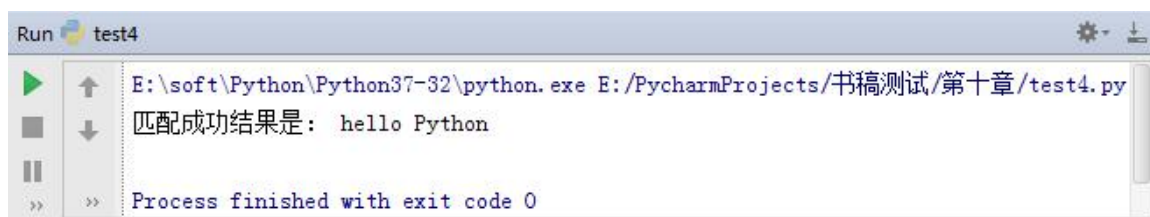
```
<_sre.SRE_Match object; span=(0, 5), match='hello'>
hello
(0, 5)
```

从上面的代码可以看出，进行文本模式匹配时，只要待匹配的字符串开始部分可以匹配文本模式，就算匹配成功。

【示例】match 方法中 flag 可选标志的使用

```
import re
s = 'hello Python!'
m=re.match('hello python',s,re.I) #忽略大小写
if m is not None:
    print('匹配成功结果是：',m.group())
else:
    print('匹配失败')
```

执行结果如下图所示：



```
Run test4
E:\soft\Python\Python37-32\python.exe E:/PycharmProjects/书稿测试/第十章/test4.py
匹配成功结果是： hello Python
Process finished with exit code 0
```

常用匹配符

正则表达式中常用的字符

符号	描述
.	匹配任意一个字符（除了\n）
[]	匹配列表中的字符

\w	匹配字母、数字、下划线, 即 a-z, A-Z, 0-9, _
\W	匹配不是字母、数字、下划线
\s	匹配空白字符, 即空格 (\n, \t)
\S	匹配不是空白的字符
\d	匹配数字, 即 0-9
\D	匹配非数字的字符

【示例】常用匹配符的使用

```
import re
print('-----.的使用-----')
pattern='.' #不能匹配换行符\n
# s='a'
# s='C'
# s='_'
s='\n'
v=re.match(pattern,s)
print(v)

print('-----\d 的使用-----')
pattern='\d'
# s='9'
# s='4'
# s='a'
# s='_'
v=re.match(pattern,s)
print(v)

print('-----\D 的使用-----')
pattern='\D'
s='9'
# s='4'
# s='a'
s='_'
v=re.match(pattern,s)
print(v)
```

```
print('-----\s 的使用-----')
pattern='\s'
s=' '
s='\t'
s='\n'
s='_ '
v=re.match(pattern,s)
print(v)

print('-----\S 空白-----')
pattern='\S'
s=' '
s='\t'
s='\n'
s='_ '
v=re.match(pattern,s)
print(v)

print('-----\w 的使用-----')
# pattern='\w'
pattern='\W'
s='a'
# s='_ '
# s='5'
# s='A'
s='#'
v=re.match(pattern,s)
print(v)

print('-----[]的使用-----')
pattern='[2468]'
s='1'
s='2'
s='3'
s='4'
```

```
s='#'
v=re.match(pattern,s)
print(v)
print('-----电话号码-----')
# pattern='d\d\d\d\d\d\d\d\d'#匹配手机号
pattern='1[35789]\d\d\d\d\d\d\d'#匹配手机号
s='13456788789'
v=re.match(pattern,s)
print(v)
```

其中，匹配符“[]”可以指定一个范围，例如：“[ok]”将匹配包含“o”或“k”的字符。同时“[]”可以与 \w、\s、\d 等标记等价。例如，[0-9a-zA-Z]等价于 \w,[^0-9] 等价于 \D。

表示数量（匹配多个字符）

【示例】匹配手机号码

```
import re
print('-----手机号码-----')
# pattern='d\d\d\d\d\d\d\d\d'#匹配手机号
pattern='1[35789]\d\d\d\d\d\d\d'#匹配手机号
s='13456788789'
v=re.match(pattern,s)
print(v)
```

如果要匹配电话号码，需要形如“\d\d\d\d\d\d\d\d\d”这样的正则表达式。其中表现了 11 次“d”，表达方式烦琐。而且某些地区的电话号码是 8 位数字，区号也有可能是 3 位或 4 位数字，因此这个正则表达式就不能满足要求了。正则表达式作为一门小型的语言，还提供了对表达式的一部分进行重复处理的功能。例如，“*”可以对正则表达式的某个部分重复匹配多次。这种匹配符号称为限定符。下表列出了正则表达式中常用的限定符。

正则表达式中常用的限定符

符号	描述	符号	描述
*	匹配零次或多次	{m}	重复 m 次
+	匹配一次或多次	{m, n}	重复 m 到 n 次，其中 n 可以省略，表示 m 到任意次
?	匹配一次或零次	{m, }	至少 m 次

利用 {} 可以控制符号重复的次数。

【示例】数量的使用

```
import re
```

```

print('-----*的使用-----')
pattern='d*'  #0 次或多次
s='123abc'
s='abc'  #这时候不是 None 而是",因为 abc 前面默认有空
v=re.match(pattern,s)
print(v)
print('-----+的使用-----')
pattern='d+'  #1 次或多次
s='123abc'
s='abc'  #这时候是 None
v=re.match(pattern,s)
print(v)

print('-----?的使用-----')
pattern='d?'  #0 次或 1 次
# s='123abc'
s='abc'  #这时候是空
v=re.match(pattern,s)
print(v)

print('-----{m}的使用-----')
pattern='d{3}'  #出现 m 次
pattern='d{2}'  #出现 m 次
pattern='d{4}'  #出现 m 次
s='123abc'
v=re.match(pattern,s)
print(v)

print('-----{m,}的使用-----')
# pattern='d{3,}'  #出现大于 m 次 尽可能满足的都返回
pattern='d{2,4}'  #出现 m 到 n 次
s='1234567abc'
v=re.match(pattern,s)
print(v)

```

【示例】匹配出一个字符串首字母为大写字母，后边都是小写字母，这些小写字母可有可

无

```
pattern='[A-Z][a-z]*'
s='Hello world'
s='HEllo world'
v=re.match(pattern,s)
print(v)
```

【示例】匹配出有效的变量名

```
pattern='[A-Za-z_][0-9A-Za-z_]*'
pattern='[A-Za-z_]\w*'
# s='a'
s='ab'
s='_ab'
s='2ab'
v=re.match(pattern,s)
print(v)
```

【示例】匹配出 1-99 直接的数字

```
pattern='[1-9]\d?'
s='1'
s='55'
s='99'
s='199'
v=re.match(pattern,s)
print(v)
```

【示例】匹配出一个随机密码 8-20 位以内 (大写字母 小写字母 下划线 数字)

```
pattern='\w{8,20}'
s='12345678'
s='123__456'
v=re.match(pattern,s)
print(v)
```

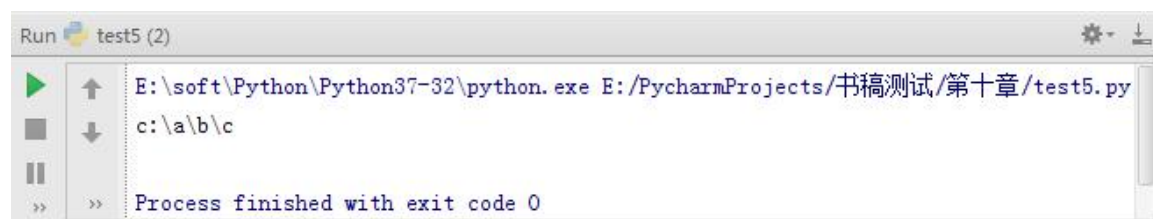
原生字符串

在大多数编程语言相同，正则表达式里使用 “\” 作为转义字符，这就可以能造成反斜杠困扰。示例如下：

【示例】“\” 作为转义字符


```
s='c:\\a\\b\\c'
print(s)
```

执行结果如下图所示：

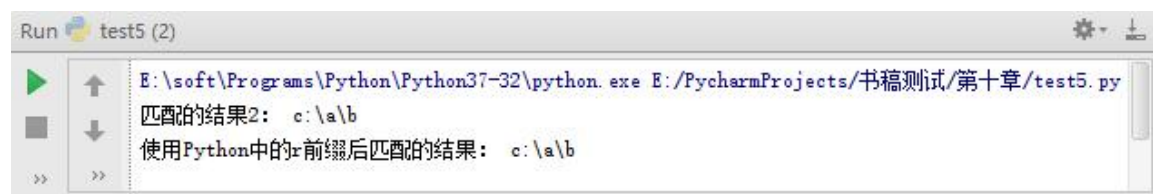


假如你需要匹配文本中的字符“\”，那么使用编程语言表示的正则表达式里将需要 4 个反斜杠“\\”：前面两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。Python 里的原生字符串很好地解决了这个问题，使用 Python 的 r 前缀。例如匹配一个数字的“\d”可以写成 r“\d”。有了原生字符串，再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

【示例】Python 中的 r 前缀的使用

```
import re
s='c:\\a\\b\\c'
m=re.match('c:\\a\\b',s)
if m is not None:
    print('匹配的结果 1: ',m.group())
m=re.match('c:\\\\a\\\\b',s)
if m is not None:
    print('匹配的结果 2: ',m.group())
#使用 Python 中的 r 前缀
m=re.match(r'c:\\a\\b',s)
if m is not None:
    print('使用 Python 中的 r 前缀后匹配的结果: ',m.group())
```

执行结果如下图所示：



边界字符

字符	功能
^	匹配字符串开头
\$	匹配字符串结尾。

\b	匹配一个单词的边界
\B	匹配非单词的边界

注意:

- ^与[^m]中的“^”的含义并不相同，后者“^”表示“除了....”的意思

【示例】匹配符\$的使用

```
#匹配 qq 邮箱， 5-10 位
pattern = '[\d]{5,10}@qq.com'
#必须限制结尾的
# pattern = '[1-9]\d{4,9}@qq.com$'
#正确的地址
v = re.match(pattern,'12345@qq.com')
#未限制结尾的前提下使用不正确的地址
# v = re.match(pattern,'12345@qq.comabc')
print(v)
```

【示例】匹配符^的使用

```
#匹配 qq 邮箱， 5-10 位
pattern = '[\d]{5,10}@qq.com'
#必须限制结尾的
# pattern = '[1-9]\d{4,9}@qq.com$'
#正确的地址
v = re.match(pattern,'12345@qq.com')
#未限制结尾的前提下使用不正确的地址
# v = re.match(pattern,'12345@qq.comabc')
print(v)
```

【示例】\b 匹配单词边界

```
pattern = r'.*\bab'
#ab 左边界的情况
v = re.match(pattern,'123 abr')
print(v)

pattern = r'.*ab\b'
#ab 为右边界的情况
```

```
v = re.match(pattern,'wab')
print(v)
```

【示例】\B 匹配非单词边界

```
#ab 不为左边界
pattern = r'.*\Bab'
v = re.match(pattern,'123 abr')
print(v)

#ab 不为右边界
pattern = r'.*ab\B'
v = re.match(pattern,'wab')
print(v)
```

search 方法

search 在一个字符串中搜索满足文本模式的字符串。语法格式如下：

```
re.search(pattern, string, flags=0)
```

函数参数与 match 方法类似，如下表所示：

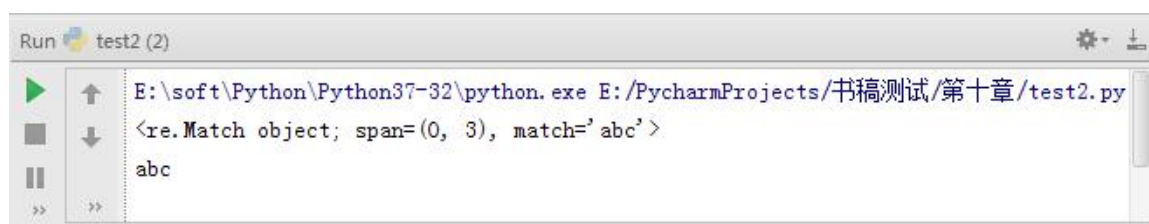
search 函数参数说明

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。 如下表列出正则表达式修饰符 - 可选标志

【示例】search 方法的使用

```
import re
m=re.search('abc','abcdefg')
print(m)
print(m.group())
```

执行结果如下图所示：



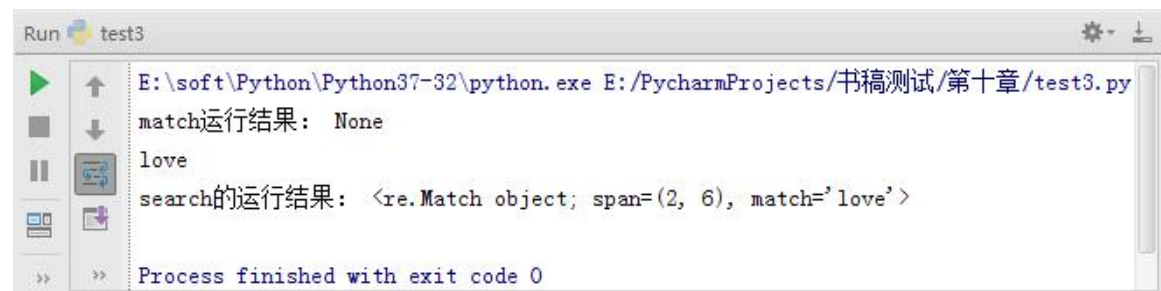
match 与 search 的区别

re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None；而 re.search 匹配整个字符串，直到找到一个匹配。

【示例】match 方法与 search 方法的使用对比

```
import re
#进行文本模式匹配，匹配失败，match 方法返回 None
m=re.match('love','I love you')
if m is not None:
    print(m.group())
print('match 运行结果: ',m)
#进行文本模式搜索，
m=re.search('love','I love you')
if m is not None:
    print(m.group())
print('search 的运行结果: ',m)
```

执行结果如下图所示：



匹配多个字符串

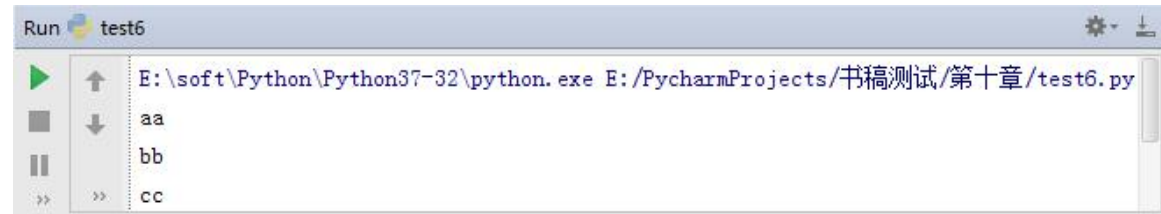
search 方法搜索一个字符串，要想搜索多个字符串，如搜索 aa、bb 和 cc，最简单的方法是在文本模式字符串中使用择一匹配符号（|）。择一匹配符号和逻辑或类似，只要满足任何一个，就算匹配成功。

【示例】择一匹配符号（|）的使用

```
import re
s='aa|bb|cc'
#match 进行匹配
m=re.match(s,'aa')    #aa 满足要求，匹配成功
print(m.group())
m=re.match(s,'bb')    #bb 满足要求，匹配成功
print(m.group())
```

```
#search 查找
m=re.search(s,'Where is cc')
print(m.group())
```

执行结果如下图所示：



从上面的代码可以看出，待匹配的字符串只要是 aa、bb 和 cc 中的任何一个就会匹配成功。

【示例】匹配 0-100 之间所有的数字

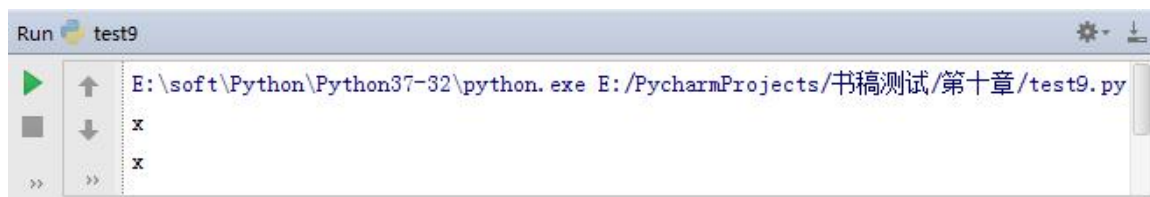
```
pattern = '[1-9]?d$|100$'
v = re.match(pattern,'0')
print(v)
v = re.match(pattern,'10')
print(v)
v = re.match(pattern,'100')
print(v)
v = re.match(pattern,'99')
print(v)
v = re.match(pattern,'200')
print(v)
```

如果待匹配的字符串中，某些字符可以有多个选择，就需要使用字符集（[]），也就是一对中括号括起来的字符串。例如，[xyz]表示 x、y、z 三个字符可以取其中任何一个，相当于“x|y|z”，所以对单个字符使用或关系时，字符集和择一匹配符的效果是一样的。示例如下：

【示例】字符集（[]）和择一匹配符（|）完成相同的效果

```
import re
m=re.match('[xyz]','x') #匹配成功
print(m.group())
m=re.match('x|y|z','x') #匹配成功
print(m.group())
```

执行结果如下图所示：

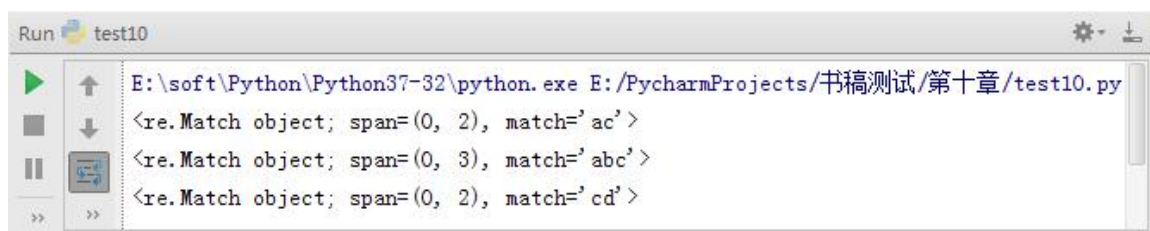


```
Run test9
E:\soft\Python\Python37-32\python.exe E:/PycharmProjects/书稿测试/第十章/test9.py
x
x
```

【示例】字符集（[]）和择一匹配符（|）的用法，及它们的差异

```
import re
#匹配以第 1 个字母是 a 或者 b，第 2 个字母是 c 或者 d，如 ac、bc、ad、bd
m=re.match('[ab][cd]','aceg')
print(m)
#匹配以 ab 开头，第 3 个字母是 c 或者 d，如 abc、abd
m=re.match('ab[cd]','abcd')
print(m)
#匹配 ab 或者 cd
m=re.match('ab|cd','cd')
print(m)
```

执行结果如下图所示：



```
Run test10
E:\soft\Python\Python37-32\python.exe E:/PycharmProjects/书稿测试/第十章/test10.py
<re.Match object; span=(0, 2), match='ac'>
<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(0, 2), match='cd'>
```

分组

如果一个模式字符串中有用一对圆括号括起来的部分，那么这部分就会作为一组，可以通过 `group` 方法的参数获取指定的组匹配的字符串。当然，如果模式字符串中没有任何用圆括号括起来的部分，那么就不会对待匹配的字符串进行分组。

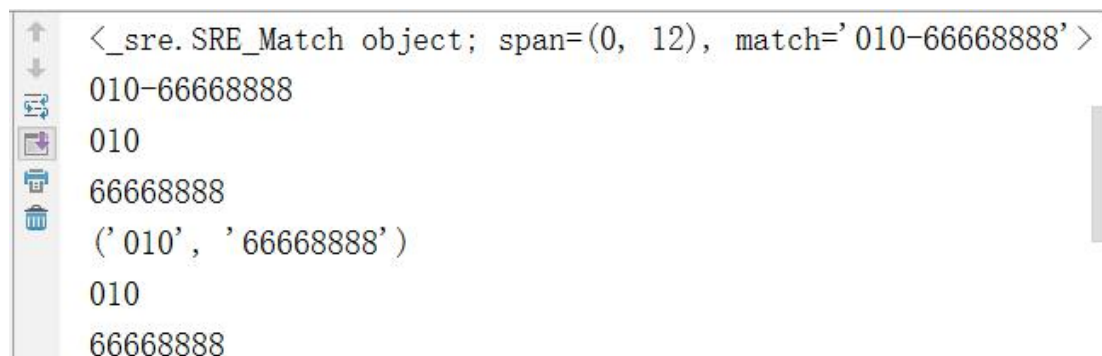
字符	功能
(ab)	将括号中的字符作为一个分组
\num	引用分组 num 匹配到的字符串
(?p<name>)	分别起组名
(?p=name)	引用别名为 name 分组匹配到的字符串

【示例】匹配座机号码

```
pattern = r'(\d+)-(\d{5,8}$)'
v = re.match(pattern,'010-66668888')
print(v)
```

```
print(v.group())
print(v.group(1))
print(v.group(2))
print(v.groups())
print(v.groups()[0])
print(v.groups()[1])
```

在上执行结果如下图所示：



```
<_sre.SRE_Match object; span=(0, 12), match='010-66668888'>
010-66668888
010
66668888
('010', '66668888')
010
66668888
```

【示例】\num 的使用

#匹配出网页标签内的数据

```
s = '<html><title>我是标题</title></html>'
```

#优化前

```
# pattern = r'<.+><.+>.</.+></.+>'
```

#优化后 可以使用分组 \2 表示引用第 2 个分组 \1 表示引用第 1 个分组

```
pattern = r'<(.)><(.)>.</\2></\1>'
```

```
v = re.match(pattern,s)
```

```
print(v)
```

【示例】?P<要起的别名> (?P=起好的别名)

```
s = '<html><h1>我是一号字体</h1></html>'
```

```
# pattern = r'<(.)><(.)>.</\2></\1>'
```

#如果分组比较多的话，数起来比较麻烦，可以使用起别名的方法?P<要起的名字> 以及使用别名(?P=之前起的别名)

```
pattern = r'<(P<key1>.)><(P<key2>.)>.</(P=key2)></(P=key1)>'
```

```
v = re.match(pattern,s)
```

```
print(v)
```

使用分组要了解如下几点：

- 只有圆括号括起来的部分才算一组，如果模式字符串中既有圆括号括起来的部分，

也有没有被圆括号括起来的部分，那么只会将被圆括号括起来的部分算作一组，其它的部分忽略。

- 用 `group` 方法获取指定组的值时，组从 1 开始，也就是说，`group(1)` 获取第 1 组的值，`group(2)` 获取第 2 组的值，以此类推。
- `groups` 方法用于获取所有组的值，以元组形式返回。所以除了使用 `group(1)` 获取第 1 组的值外，还可以使用 `groups()[0]` 获取第 1 组的值。获取第 2 组以及其它组的值的方式类似。

re 模块中其他常用的函数

sub 和 subn 搜索与替换

`sub` 函数和 `subn` 函数用于实现搜索和替换功能。这两个函数的功能几乎完全相同，都是将某个字符串中所有匹配正则表达式的一部分替换成其他字符串。用来替换的部分可能是一个字符串，也可以是一个函数，该函数返回一个用来替换的字符串。`sub` 函数返回替换后的结果，`subn` 函数返回一个元组，元组的第 1 个元素是替换后的结果，第 2 个元素是替换的总数。语法格式如下：

```
re.sub(pattern, repl, string, count=0, flags=0)
```

参数说明：

表 sub 函数参数说明

参数	描述
pattern	匹配的正则表达式
repl	替换的字符串，也可为一个函数
string	要被查找替换的原始字符串。
count	模式匹配后替换的最大次数，默认 0 表示替换所有的匹配

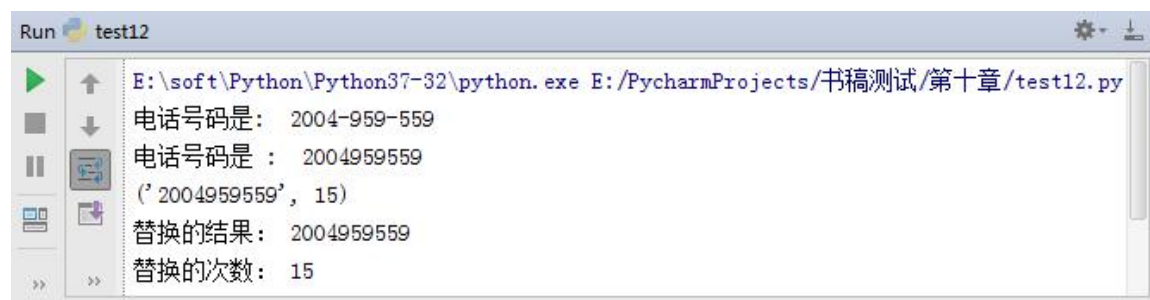
【示例】sub 和 subn 方法的使用

```
import re
phone = "2004-959-559 # 这是一个国外电话号码"
# 删除字符串中的 Python 注释
num = re.sub(r'#. *$', "", phone)
print("电话号码是: ", num)
# 删除非数字(-)的字符串
num = re.sub(r'\D', "", phone)
print("电话号码是 : ", num)
#subn 函数的使用
result=re.subn(r'\D', "", phone)
print(result)
```



```
print('替换的结果: ',result[0])
print('替换的次数: ',result[1])
```

在上执行结果如下图所示:



```
Run test12
E:\soft\Python\Python37-32\python.exe E:/PycharmProjects/书稿测试/第十章/test12.py
电话号码是: 2004-959-559
电话号码是: 2004959559
('2004959559', 15)
替换的结果: 2004959559
替换的次数: 15
```

compile 函数

compile 函数用于编译正则表达式, 生成一个正则表达式 (Pattern) 对象, 供 match() 和 search() 这两个函数使用。语法格式为:

```
re.compile(pattern[, flags])
```

参数说明:

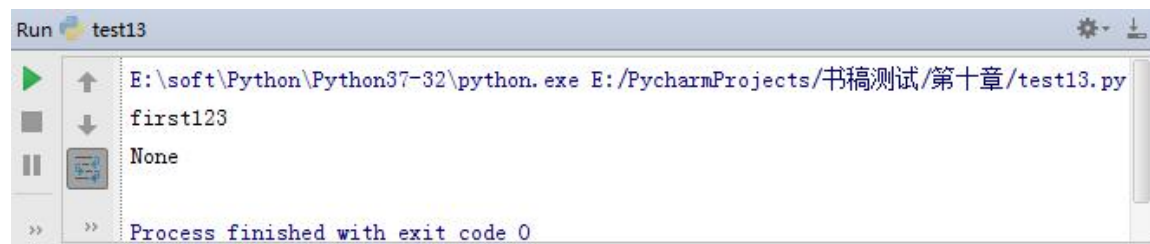
表 compile 函数参数说明

参数	描述
pattern	一个字符串形式的正则表达式
flags	可选, 表示匹配模式, 比如忽略大小写, 多行模式等,

【示例】compile 函数的使用

```
import re
s='first123 line'
regex=re.compile(r'\w+') #匹配至少一个字母或数字
m=regex.match(s)
print(m.group())
# s 的开头是 "f", 但正则中限制了开始为 i 所以匹配失败
regex = re.compile("^i\w+")
print(regex.match(s))
```

在上执行结果如图所示:



```
Run test13
E:\soft\Python\Python37-32\python.exe E:/PycharmProjects/书稿测试/第十章/test13.py
first123
None
Process finished with exit code 0
```

findall 函数

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回空列表。语法格式如下：

```
findall(pattern, string, flags=0)
```

参数说明：

表 findall 函数参数说明

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。 如下表列出正则表达式修饰符 - 可选标志

【示例】findall 函数的使用

```
pattern=r'\w+'
s='first 1 second 2 third 3'
o=re.findall(pattern,s)
print(o)
```

在上执行结果如图所示：

```
['first', '1', 'second', '2', 'third', '3']
```

注意：

- match 和 search 是匹配一次 findall 匹配所有

finditer 函数

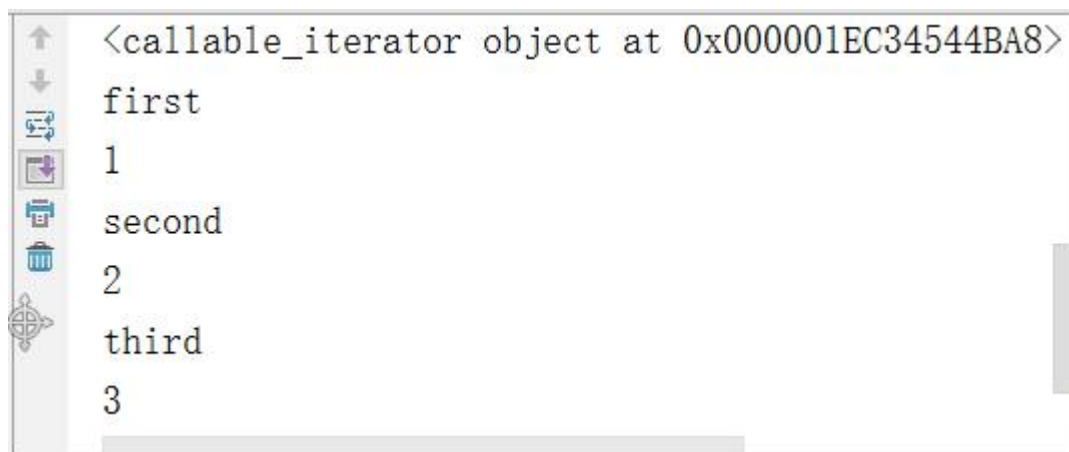
和 findall 类似，在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

【示例】finditer 函数的使用

```
pattern=r'\w+'
s='first 1 second 2 third 3'
o=re.finditer(pattern,s)
print(o)
for i in o:
```

```
print(i.group())
```

在上执行结果如图所示：



split 函数

split 函数用于根据正则表达式分隔字符串，也就是说，将字符串与模式匹配的子字符串都作为分隔符来分隔这个字符串。split 函数返回一个列表形式的分隔结果，每一个列表元素都是分隔的子字符串。语法格式如下：

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

参数说明：

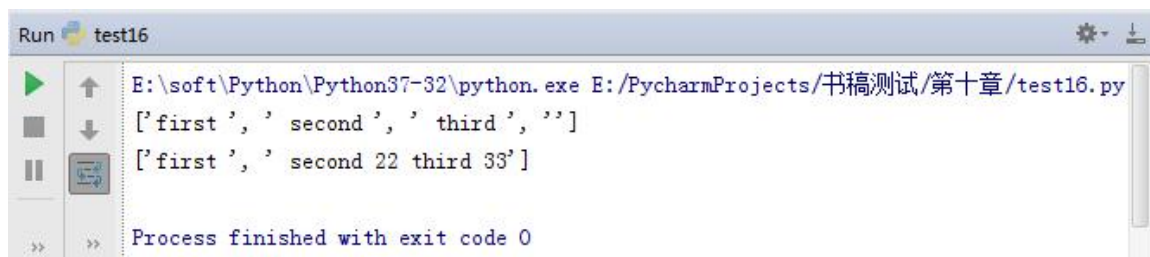
表 split 函数参数说明

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
maxsplit	分隔次数，maxsplit=1 分隔一次，默认为 0，不限制次数。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

【示例】split 函数的使用

```
import re
s='first 11 second 22 third 33'
#按数字切分
print(re.split(r'\d+',s))
# maxsplit 参数限定分隔的次数，这里限定为 1，也就是只分隔一次
print(re.split(r'\d+',s,1))
```

在上执行结果如图所示：



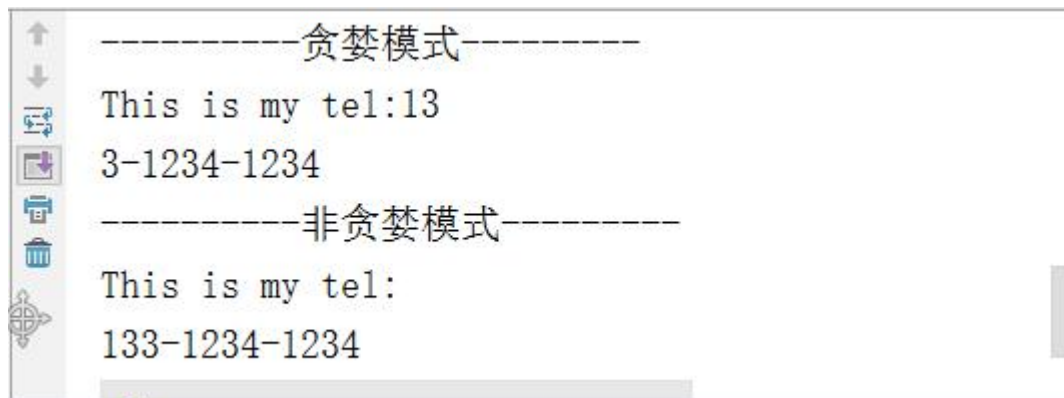
贪婪模式和非贪婪

贪婪模式指 Python 里数量词默认是贪婪的，总是尝试匹配尽可能多的字符。非贪婪模式与贪婪相反，总是尝试匹配尽可能少的字符，可以使用 "*"、"?", "+", "{m,n}" 后面加上 "?"，使贪婪变成非贪婪

【示例】贪婪模式，.+ 中的 '.' 会尽量多的匹配

```
v = re.match(r'(.+)(\d+-\d+-\d+)', 'This is my tel:133-1234-1234')
print('-----贪婪模式-----')
print(v.group(1))
print(v.group(2))
print('-----非贪婪模式-----')
v = re.match(r'(.+?)(\d+-\d+-\d+)', 'This is my tel:133-1234-1234')
print(v.group(1))
print(v.group(2))
```

执行结果如下图：



【示例】贪婪模式非贪婪模式测试

```
print('贪婪模式')
v = re.match(r'abc(\d+)', 'abc123')
print(v.group(1))
#非贪婪模式
print('非贪婪模式')
v = re.match(r'abc(\d+?)', 'abc123')
```

```
print(v.group(1))
```

执行结果如下图：

