Python 全栈文档

第三章 正则表达式

在开发中会有大量的字符串处理工作，其中经常会涉及到字符串格式的校验。

思考1

场景：如何判断一个字符串是手机号呢?

测试文件

```
aesdf
13811011234
aa1a3hi233rhi3
87156340
affa124564531346546
afa19454132135
```

思考2

场景：在一个文件中，查找出msb开头的语句

测试文件

```
msb hello python
msb c++
itheima ios
itheima php
```

1、正则表达式概述

正则表达式，又称正规表示式、正规表示法、正规表达式、规则表达式、常规表示法（英语：Regular Expression，在代码中常简写为regex、regexp或RE），是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列匹配某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本。

2、re模块

一个正则表达式（或RE）指定了一集与之匹配的字符串；模块内的函数可以让你检查某个字符串是否跟给定的正则表达式匹配

模块定义了几个函数，常量，和一个例外。有些函数是编译后的正则表达式方法的简化版本（少了一些特性）。绝大部分重要的应用，总是会先将正则表达式编译，之后在进行操作

那现在我们先熟悉re模块的一简单的方法，来解决上面的思考题

compile方法

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

# 作用：把正则表达式语法转化成正则表达式对象

pattern：正则表达式语法

flags定义匹配模式包括：{

re.I：忽略大小写

re.L：表示特殊字符集 \w,\W,\b,\B,\s,\S 依赖于当前环境

re.M：多行模式

re.S：' . '并且包括换行符在内的任意字符（注意：' . '匹配任意字符但不包括换行符）

re.U：表示特殊字符集 \w,\d,\D,\S 依赖于 Unicode 字符属性数据库

}
```

search方法

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

# 作用：扫描整个字符串，并返回第一个成功的匹配。如果匹配失败，则返回None。

pattern ： 正则表达式对象

string ： 要被查找替换的原始字符串。

flags定义匹配模式包括：{

re.I：忽略大小写

re.L：表示特殊字符集 \w,\W,\b,\B,\s,\S 依赖于当前环境

re.M：多行模式

re.S：' . '并且包括换行符在内的任意字符（注意：' . '匹配任意字符但不包括换行符）

re.U：表示特殊字符集 \w,\d,\D,\S 依赖于 Unicode 字符属性数据库

}
```

match方法

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

# 作用：从起始位置开始匹配，匹配成功返回一个对象，未匹配成功返回None
```

```
pattern : 正则表达式对象

string : 需要匹配的字符串

flags定义匹配模式包括：{

re.I：忽略大小写

re.L：表示特殊字符集 \w,\W,\b,\B,\s,\S 依赖于当前环境

re.M：多行模式

re.S：' . '并且包括换行符在内的任意字符（注意：' . '匹配任意字符但不包括换行符）

re.U：表示特殊字符集 \w,\d,\D,\S 依赖于 Unicode 字符属性数据库

}
```

这里我们分别简单了解一下这些模块，

1、compile方法是将正则表达式转换成对象，

2、search和match方法是根据compile对象转换生成好的规则，进行匹配。

那我们将上方的思考题拿下来，先看思考一下

```
# 导入模块
>>> import re

#  如何判断一个字符串是手机号呢
>>> tel_1 = '''
    aesdf13811011234
    aa1a3hi233rhi3
    87156340
    affa124564531346546

afa19454132135
    '''
# 这里我们不说的过于复杂了，也不说特殊号码了,简单了解一下规则
# 1、由11位正整数字组成
# 2、第一位数字必须由1开头，第二位数字由3-9组成
# 根据这些条件这里即可以生成一个匹配式    1[3,9]\d{9}，下面我们会讲解上匹配语法与规则
>>> pattern = re.compile(r'1[3,9]\d{9}')

# 使用search方法，匹配到一个电话号码
>>> print(re.search(pattern, tel_1))
<re.Match object; span=(7, 18), match='13811011234'>

# 再使用match方法，输出了None
>>> print(re.match(pattern, tel_1))
None

# ------------------------------------------------
```

```
# 那我们换一个例子再看，
>>> tel_2 = '''19454132135abuw'''
# 再将匹配表达式转换成匹配对象
>>> pattern = re.compile(r'1[3,9]\d{9}')

# 先使用search方法
>>> print(re.search(pattern, tel_2))
<re.Match object; span=(0, 11), match='19454132135'>

# 再使用match方法，看到对象响应的结果，出现了需要匹配的号码
>>> print(re.match(pattern, tel_2))
<re.Match object; span=(0, 11), match='19454132135'>
```

我们可以发现，更多的时候，我们在字符串或者是一堆单词内容里面不确定范围搜索，**可以使用search来进行匹配第一个最先匹配到的正常的电话号码**，而**match用来匹配第一个注意是第一个字符的，这里的第一个是在被搜索的这串字符的第一位索引上的**

这里面我们根据返回回来的内容，看到有三部分

```
1、re.Match object 对象

2、span=()搜索结果在文本索引位置

3、match匹配结果
```

这里我们现在就需要急切了解到两个问题

```
# 第一、得到的匹配对象re.Match object该如何处理，以及re模块是否存在一些其他的方法

# 第二、正则表达式的写法
```

3、Match对象

我们来看一下，Match对象，Match对象是一次匹配的结果，包含匹配的很多信息

Match匹配对象的属性

属性与方法	描述
pos	搜索的开始位置
endpos	搜索的结束位置
string	搜索的字符串
re	当前使用的正则表达式对象
lastindex	最后匹配的组索引
lastgroup	最后匹配的组名
group(index)	某个组匹配的结果
groups()	所有分组的匹配结果，每个分组组成的结果以列表返回
groupdict()	返回组名作为key，每个分组的匹配结果作为value的字典
start([group])	获取组的开始位置
end([group])	获取组的结束位置
span([group])	获取组的开始和结束位置
expand(template)	使用组的匹配结果来替换template中的内容，并把替换后的字符串返回

这里我们来看一些例子

思考题的例子来看看

```
>>> import re

>>> tel_1 = '''
    aesdf13811011234
    aa1a3hi233rhi3
    87156340
    affa124564531346546

afa19454132135
    '''
>>> pattern = re.compile(r'1[3,9]\d{9}')
>>> results = re.search(pattern, tel_1)

# pos表示搜索的开始的位置，endpos搜索结束的位置
>>> print(results.pos, results.endpos)
0 83

# group表示匹配的结果
>>> print(results.group())
13811011234


>>> tel_2 = '''19454132135abuw'''
```

```
>>> results = re.search(pattern, tel_2)

# pos表示搜索的开始的位置，endpos搜索结束的位置
>>> print(results.pos, results.endpos)
0 15

# group表示匹配的结果
>>> print(results.group())
19454132135
```

4、正则表达式

构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与运算符可以将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

正则表达式是由普通字符（例如字符 a 到 z）以及特殊字符（称为"元字符"）组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配

元字符 （参见 python 模块 re 文档）

字符	功能
.	匹配任意字符（不包括换行符）
^	匹配开始位置，多行模式下匹配每一行的开始，(也有取反的意思，区分应用场景)
$	匹配结束位置，多行模式下匹配每一行的结束
*	匹配前一个元字符0到多次
+	匹配前一个元字符1到多次
?	匹配前一个元字符0到1次
{m,n}	匹配前一个元字符m到n
\ \	转义字符，跟在其后的字符将失去作为特殊元字符的含义， 例如\.只能匹配.，不能再匹配任意字符
[ ]	字符集，一个字符的集合，可匹配其中任意一个字符
|	逻辑表达式 或 ，比如 a
(...)	分组，默认为捕获，即被分组的内容可以被单独取出， 默认每个分组有个索引，从 1 开始，按照"("的顺序决定索引值
(?iLmsux)	分组中可以设置模式，iLmsux之中的每个字符代表一个模式
(?:...)	分组的不捕获模式，计算索引时会跳过这个分组
(?P...)	分组的命名模式，取此分组中的内容时可以使用索引也可以使用name
(?P=name)	分组的引用模式，可在同一个正则表达式用引用前面命名过的正则
(?#...)	注释，不影响正则表达式其它部分
(?=...)	顺序肯定环视，表示所在位置右侧能够匹配括号内正则
(?!...)	顺序否定环视，表示所在位置右侧不能匹配括号内正则
(?<=...)	逆序肯定环视，表示所在位置左侧能够匹配括号内正则
(?<!...)	逆序否定环视，表示所在位置左侧不能匹配括号内正则
(?(id/name)yes|no)	若前面指定id或name的分区匹配成功则执行yes处的正则，否则执行no处的正则
\number	匹配和前面索引为number的分组捕获到的内容一样的字符串
\A	匹配字符串开始位置，忽略多行模式
\Z	匹配字符串结束位置，忽略多行模式
\b	匹配位于单词开始或结束位置的空字符串
\B	匹配不位于单词开始或结束位置的空字符串
\d	匹配一个数字， 相当于 [0-9]

字符	功能
\D	匹配非数字,相当于 [^0-9]
\s	匹配任意空白字符， 相当于 [\t\n\r\f\v]
\S	匹配非空白字符，相当于 [^ \t\n\r\f\v]
\w	匹配数字、字母、下划线中任意一个字符， 相当于 [a-zA-Z0-9_]
\W	匹配非数字、字母、下划线中的任意字符，相当于[^\w]

以上是所涉及到的字符，我们分类来讲解

5、表示字符

字符	功能
.	匹配任意单个字符（不包括换行符）
[ ]	匹配字符集，区间中的集合，可匹配其中任意一个字符
\d	匹配数字，即0-9,可以表示为[0-9]
\D	匹配非数字，即不是数字，相当于 [^0-9]
\s	匹配空白字符，即 空格，tab键，相当于 [\t\n\r\f\v]
\S	匹配非空白字符，相当于 [^ \t\n\r\f\v]
\w	匹配单词字符，即a-z、A-Z、0-9、_
\W	匹配非单词字符，即非数字、字母、下划线中的任意字符，相当于[^\w]

我们来看一下用法，

'.'用法

```
# 导入模块
>>> import re
# 测试匹配任意字符（不包括换行符）使用的re中的match方法
>>> print(re.match(".","a").group())
'a'

>>> print(re.match(".","1").group())
'1'

>>> print(re.match(".","_").group())
'_'

>>> print(re.match(".","0").group())
'0'

# 只有匹配到'\n'时候提示None，说明，匹配任意字符（不包括换行符）
```

```
>>> print(re.match(".","\n"))
None
```

'[]'用法

```
# 导入模块
>>> import re

# 匹配字符集，区间中的集合，可匹配其中任意一个字符，使用的re中的match方法

# 如果hello的首字符小写，那么正则表达式需要小写的h
>>> print(re.match("h","hello Python").group())
'h'

# 如果hello的首字符大写，那么正则表达式需要大写的H
>>> print(re.match("H","Hello Python").group())
'H'

# 大小写h都可以的情况
>>> print(re.match("[hH]","hello Python").group())
'h'

>>> print(re.match("[hH]","Hello Python").group())
'H'

# 匹配0到9第一种写法
>>> print(re.match("[0123456789]","7Hello Python").group())
'7'

# 匹配0到9第二种写法
>>> print(re.match("[0-9]","7Hello Python").group())
'7'

# 这里我们看到匹配的字符串里面没有 0到9的数字，返回来None，这里没有group，是因为空对象没有这个属性
>>> print(re.match("[0-9]","Hello Python"))
None
```

'\d'用法

```
# 导入模块
>>> import re

# 普通的匹配方式
>>> print(re.match("嫦娥1号","嫦娥1号发射成功"))
<re.Match object; span=(0, 4), match='嫦娥1号'>

>>> print(re.match("嫦娥8号","嫦娥8号发射成功"))
<re.Match object; span=(0, 4), match='嫦娥8号'>

# 使用\d进行匹配
>>> print(re.match("嫦娥\d号","嫦娥1号发射成功"))
```

```
<re.Match object; span=(0, 4), match='嫦娥1号'>

>>> print(re.match("嫦娥\d号","嫦娥8号发射成功"))
<re.Match object; span=(0, 4), match='嫦娥8号'>

# 这里匹配中文的数字，非数字类型，返回了None,这里之所以没有
>>> print(re.match("嫦娥\d号","嫦娥十号发射成功"))
None
```

6、转义字符

'\'在正则表达式中，使用反斜杠进行转义，与其他的方法类似相同

转义特殊字符（允许你匹配 `'*'`, `'?'`,或者此类其他字符），或者表示一个特殊序列

我们知道在Python中也存在反斜杠的转义字符，其中我们还有更简便的方法，就是原生字符

如果没有使用原始字符串（ `r'raw'` ）来表达样式，要牢记Python也使用反斜杠作为转义序列；只有转义序列没有被Python的分析器识别，反斜杠和字符才能出现在字符串中。如果Python可以识别这个序列，那么反斜杠就应该重复两次。这会导致理解上非常的麻烦，所以高度推荐使用原始字符串，就算是最简单的表达式，也要使用原始字符串

代码：

```
# 导包
>>> import re
# 定义变量
>>> path = "c:\\a\\b\\c"
>>> path
'c:\\a\\b\\c'
>>> print(path)
c:\a\b\c

# 经过正则表达式的转移，表达的就不是转义字符，四个反斜杠，其中前一个反斜杠对后一个反斜杠进行转移，代表只有两个反斜杠
>>> re.match("c:\\\\",path).group()
'c:\\'
# 经过一次反斜杠的转以后，在进行python解释器的打印函数，再次会通过字符进行转义打印，那这里我们的结果就生成一个反斜杠
>>> ret = re.match("c:\\\\",path).group()
>>> print(ret)
c:\
>>> ret = re.match("c:\\\\a",path).group()
>>> print(ret)
c:\a

# 这里直接只有一个反斜杠，查询不被转移，所以只保留一个反斜杠，打印结果反斜杠没有转义，即为字符打印结果，只打印一个反斜杠
>>> ret = re.match(r"c:\\a",path).group()
>>> print(ret)
c:\a

#  这里采用python的原生字符操作，实际上我们调用的操作没有问题，也能实现我们想要的内容
>>> ret = re.match(r"c:\\a",path)
```

```
>>> ret.group()
'c:\\a'
# 打印这里我们没有限定转移含义，直接打印出来
>>> print(ret)
c:\a
# 实际上变量ret还是实际我们想要匹配的字符串
>>> ret
'c:\\a'
```

我们可以看到，'\'反斜杠转义字符在实际python字符串中是没有被转义的，但是打印出来的结果中被进行了转义，由我们的python解释器进行转义字符，我们了解到正则表达式中的转义字符也与python中的转义字符类似的规则

7、表示数量

匹配多个字符的相关格式

字符	功能
*	匹配前一个字符出现0次或者无限次，即可有可无
+	匹配前一个字符出现1次或者无限次，即至少有1次
?	匹配前一个字符出现1次或者0次，即要么有1次，要么没有
{m}	匹配前一个字符出现m次
{m,}	匹配前一个字符至少出现m次
{m,n}	匹配前一个字符出现从m到n次

这里我们根据实际例子来操作：

'*'用法

需求：匹配出，一个字符串第一个字母为大写字母，后面都是小写字母并且这些小写字母可有可无

```
>>> import re

>>> ret = re.match("[A-Z][a-z]*","Mm")
>>> ret.group()
'Mm'

>>> ret = re.match("[A-Z][a-z]*","Aabcdef")
>>> ret.group()
'Aabcdef'

>>> ret = re.match("[A-Z][a-z]*","C")
>>> ret.group()
'C'
```

我们来解析这段正则表达式[A-Z][a-z]*

首先两个中括号，这里中括号代表用法我们还记得是表示字符集，可以匹配集合中任意字符，

我们这里的[A-Z]就是匹配26个大写的英文字母，

后面这段[a-z]是匹配26个小写的英文字母，

其中*号代表匹配前一个字符出现0次或者0次以上，

此时我们的定义内容就符合规则，其中第一个位置上的数是任意大写字母，后面可有可无

看到我们的例子验证发现*是匹配0次或者是0次以上

'+'用法

需求：匹配出，变量名是否有效

```
import re

>>> ret = re.match("[a-zA-Z_]+[\w]*","name1")
>>> ret.group()
'name1'

>>> ret = re.match("[a-zA-Z_]+[\w]*","_name")
>>> ret.group()
'_name'

>>> ret = re.match("[a-zA-Z_]+[\w]*","2_n_ame")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

变量是需要一些条件的，当然现在这里条件不会太过严禁，满足标准标识符结构即可, [a-zA-Z_]+[\w]*

变量第一个字符可以是字母下划线，那我们就可以使用我们的中括号来做取值范围，接着我们在里面写一些取值字母与下划线

我们的第一个字符产生后，其实可以很明显的看出来这个不仅没有问题，还可以多次列出来，这里我们可以产生 + 作为匹配前一个字符出现1次或者无限次，即至少有1次，后面我们可以加上，字母数字下划线，也是用中括号，将\w表示的匹配字母数字下划线，即a-z、A-Z、0-9、_,接着我们使用*，表示这个字符可有可无，我们明白，并不是一定需要第二个字符，有时候我们定义一个变量a也是可以的

[a-zA-Z_]+[\w]*所以这段里面，表示着一个或一个以上的字母下划线，和0或0个以上的字母数字下划线，组合的变量

'？'用法

需求：匹配出，0到99之间的数字，当然写法有很多，还有更加简单的写法，但是我们这里按照练习这些元字符

分析：0到99的字符，首先也是两位数字，第一位包含的是1-9，第二位包含0-9，但是第一位可以没有，所以我们这里采用'*'与'+'字符就不可取了，这里就可以使用上'？'，它匹配前一个元字符0到1次

```
# 导包
>>> import re

>>> ret = re.match("[1-9]?[0-9]","7")
>>> ret.group()

'7'
>>> ret = re.match("[1-9]?[0-9]","33")
>>> ret.group()
'33'

>>> ret = re.match("[1-9]?[0-9]","09")
>>> ret.group()
'0'
```

我们来分析这串正则表达式：[1-9]?[0-9]

首先中括号中的1-9随机可匹配一个1到9的任意正整数，但是我们第一位数是不一定存在的，那我们可以使用'？'来进行匹配操作，表示前一个字符可有0个或者1个，第二个字符表示的是中括号中的0-9任意一位字符，但是需要了解，匹配 02只会匹配0，匹配 08也是只会匹配0。

'{m}'用法

需求：匹配出，8到20位的密码，可以是大小写英文字母、数字、下划线

分析：我们知道我们常用的密码就是大小写英文字母、数字、下划线当然这里没有包含点之类的一些其他字符，这个可以暂时不用严谨，我们先测试大括号的功能，｛m｝匹配前一个字符出现m次，｛m, n｝匹配前一个字符出现从m到n次,但是注意这里的m需要小于n

```
>>> import re

>>> ret = re.match("[a-zA-Z0-9_]{6}","12a3g45678")
>>> ret.group()
'12a3g4'

>>> ret = re.match("[a-zA-Z0-9_]{8,20}","1ad12f23s34455ff66")
>>> ret.group()
'1ad12f23s34455ff66'
```

既然我们知道｛m, n｝匹配前一个字符出现从m到n次，[a-zA-Z0-9_]{6}我们来解析一下这串字符，首先我们想办法来限定一下能生成任意的大小写英文字母、数字、下划线字符，我们想到，首先中括号，可以任意选择，其次我们可以加入\w_就可以解决问题

综合上述，可以来一题练习匹配出163的邮箱地址，例如hello_123@163.com,当然这里我们暂时不算上VIP邮箱

6～18个字符，可使用字母、数字、下划线，需要以字母开头

首先分析一下这里，限定，字母开头 后面可跟字母、数字、下划线，那我们可以写出 [a-zA-Z][\w_]

其次名称至少是有6至18个，我们可以去除必须要的第一个首字母，那还剩余5至17个可填，这里我们将后余可以出现的字符表示出来，进行位数限定， [a-zA-Z][\w_]{5,17}

至此可以写出完整的正则，[a-zA-Z][\w_]{5,17}@163\.com,后缀不变，因为我们只匹配后缀为163.com的域名。但是注意'' . ''的含义是0个或者0个以上的哦，需要用到转义字符进行只是表示点的意思

```
# 导包
>>> import re
>>> tel = '''
    awhahlf@163.com
    affafafafaaaaaaaaaaaaaaaa@163.com
    afa_@163.com
    225afafaf@163.com
    aaaa____@qq.com
    aaaa____@163.com
    '''
# 这里我们使用新的匹配方法re.search，匹配第一个返回成功的
>>> ret = re.search("[a-zA-Z][\w_]{5,17}@163\.com", tel)
>>> ret.group()

'awhahlf@163.com'
```

8、表示边界

限定开始与结尾的匹配

字符	功能
^	匹配开始位置，多行模式下匹配每一行的开始， (也有取反的意思，区分应用场景)
$	匹配字符串结尾
\b	匹配一个单词的边界
\B	匹配非单词边界

'^'用法

用法一: 限定开头

匹配字符串的开头， 并且在 `MULTILINE` 模式也匹配换行后的首个符号

re.MULTILINE 设置以后，样式字符 `'^'` 匹配字符串的开始，和每一行的开始（换行符后面紧跟的符号）；样式字符 `'$'` 匹配字符串尾，和每一行的结尾（换行符前面那个符号）。默认情况下， `'^'` 匹配字符串头， `'$'` 匹配字符串尾，

具体实现例子

```
# 导包
>>> import re

# 定义字符串，害死邮箱的例子
>>> tel = '''
225afafaf@163.com
   awhahlf@163.com
affafafafaaaaaaaaaaaaaaaa@163.com
afa_@163.com
225afafaf@163.com
```

```
aaaa____@qq.com
aaaa____@163.com
'''
# 生成正则表达式，一个是带 ^ ,一个不带
>>> pattern1 = r"[a-zA-Z][\w_]{5,17}@163.com"
>>> pattern2 = r"^[a-zA-Z][\w_]{5,17}@163.com"

# 生成三个正则对象，第一个是 正常写入的正则表达式，第二个是带 ^ 限定的表达式，第三个是带 ^ 限定的表达式且支持换行匹配
>>> single_line1 = re.compile(pattern1)
>>> single_line2 = re.compile(pattern2)
>>> multiline = re.compile(pattern2, re.MULTILINE)

# 使用三个已经生成好的匹配字符
>>> ret1 = re.search(single_line1, tel)
>>> ret2 = re.search(single_line2, tel)
>>> ret3 = re.search(multiline, tel)

# 第一种不带 ^ 匹配发现立即匹配到，第二行的邮箱
>>> print(ret1.group())
afafaf@163.com

# 第二种带 ^ 匹配发现没有匹配到，我们看到匹配限定，不换行第一行匹配
>>> print(ret2)
None

# 第二种带 ^ 匹配发现没有匹配到，我们看到匹配限定支持换行使用了re.MULTILINE，换行匹配查到了afa_@163.com
# re.MULTILINE支持换行后接字符，不能再接空格字符等
>>> print(ret3.group())
aaaa____@163.com
```

上面的例子我们看到了 ^ 与re.MULTILINE用法，但是我们仔细来看 ^ 应用模式

```
# 导包
>>> import re

# 新生成一个例子
>>> tel1 = 'awhahlf@163.com, affafafafaaaaaaaaaaaaaaaa@163.com, afa_@163.com'
>>> tel2 = '  awhahlf@163.com, affafafafaaaaaaaaaaaaaaaa@163.com, afa_@163.com'

# 第一行不存在空格，我们使用相同的表达式，匹配到了第一行，第一个字符
>>> ret = re.search("^[a-zA-Z][\w_]{5,17}@163\.com", tel1)
>>> print(ret.group())
'awhahlf@163.com'

# 但是当我们第一行存在空格，我们使用相同的表达式，匹配不到对应的内容，返回来一个空
>>> ret = re.search("^[a-zA-Z][\w_]{5,17}@163\.com", tel2)
>>> print(ret)
None
```

被限定字符，也会受到空格换行等符号的影响，所以在边界相关的内容中，空格与换行等等这些会间接影响到结果的判断，

用法二：取反

在某些特定的条件下，^ 有存在取反的作用，例如当我们使用[^0-9],这就是取反一个非数字的例子

```
# 导包
>>> import re

# 需要筛选的字符
>>> tel = '123789'

# 使用字符,直接取反，可以很直接的看到
>>> ret = re.search("[^0-9]", tel)
>>> print(ret)
None

# 但是将 ^ 从中括号中拿出来就是取出对应的开头匹配的值
>>> ret = re.search("^[0-9]", tel)
>>> print(ret.group())
1
```

很明显，在对应的环境中 ^ 所显示的内容是取反的含义，不过需要注意：是在[ ]中括号字符集限定中。

'$'用法

匹配字符串尾或者换行符的前一个字符，当然换行符是在 `MULTILINE` 模式匹配换行符的前一个字符，

例如：`foo` 可以匹配 `'foo'` 和 `'foobar'`，但正则 `foo$` 只匹配 `'foo'`。更有趣的是， 在 `'foo1\nfoo2\n'` 搜索 `foo.$` ，通常匹配 `'foo2'` ，但在 `MULTILINE` 模式，可以匹配到 `'foo1'` ；在 `'foo\n'` 搜索 `$` 会找到两个空串：一个在换行前，一个在字符串最后

```
# 导包
import re

# 我们将匹配后缀信息，此时我们没有进行限定结尾
>>> ret = re.match("[a-zA-Z][\w_]{5,17}@163\.com", "xiaoWang@163.com")
>>> ret.group()
'xiaoWang@163.com'

# 继续匹配后缀为163.comheihei，这个如果是后缀增加，那我们在筛选的时候就会存在一些问题，因为实际上域名就已经不对了。
>>> ret = re.match("[a-zA-Z][\w_]{5,17}@163\.com", "xiaoWang@163.comheihei")
>>> ret.group()
'xiaoWang@163.com'

# 通过$来确定末尾
>>> ret = re.match("[a-zA-Z][\w_]{5,17}@163\.com$", "xiaoWang@163.comheihei")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    ret.group()
```

```
AttributeError: 'NoneType' object has no attribute 'group'
```

'\b' 用法

'\b' 匹配空字符串，但只在单词开始或结尾的位置。一个单词被定义为一个单词字符的序列。注意，通常 `\b` 定义为 `\w` 和 `\W` 字符之间，或者 `\w` 和字符串开始/结尾的边界， 意思就是 `r'\bfoo\b'` 匹配 `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` 但不匹配 `'foobar'` 或者 `'foo3'`

这里就产生了一个概念，单词与字符，单词是一组词组，而字符是包含单字符与多字符。

```
# 导入模块
import re

# 匹配 bver开头结尾的单词，当然这里面在字符前面还包含了  '.*'号
>>> re.match(r".*\bver\b", "ho ver abc").group()
'ho ver'

# bver开头结尾的单词，但是这里单词为 verabc 不符合筛选错误
>>> re.match(r".*\bver\b", "ho verabc").group()
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    re.match(r".*\bver\b", "ho verabc").group()
AttributeError: 'NoneType' object has no attribute 'group'

# bver开头结尾的单词，但是这里单词为 hover 不符合筛选错误
>>> re.match(r".*\bver\b", "hover abc").group()
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    re.match(r".*\bver\b", "hover abc").group()
AttributeError: 'NoneType' object has no attribute 'group'
```

'\B' 用法

匹配空字符串，但不能在词的开头或者结尾。意思就是 `r'py\B'` 匹配 `'python'`, `'py3'`, `'py2'`, 但不匹配 `'py'`, `'py.'`, 或者 `'py!'`. `\B` 是 `\b` 的取非，所以Unicode样式的词语是由Unicode字母，数字或下划线构成的，虽然可以用 `ASCII` 标志来改变。如果使用了 `LOCALE` 标志，则词的边界由当前语言区域设置

```
        ASCII：让 `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` 和 `\S` 只匹配ASCII，而不是Unicode。这只对Unicode样式有效，会被byte样式忽略

        LOCALE：由当前语言区域决定 `\w`, `\W`, `\b`, `\B` 和大小写敏感匹配。这个标记只能对byte样式有效。这个标记不推荐使用，因为语言区域机制很不可靠，它一次只能处理一个 "习惯"，而且只对8位字节有效。Unicode匹配在Python 3 里默认启用，并可以处理不同语言
```

\B是区分于\b，与之相对，进行取反操作的

```
# 导包
>>> import re

# 相同的内容，使用\B进行取反操作， 匹配 ver 非包含开头结尾的单词，当然这里面在字符前面还包含了  '.*'号
# 如果未加'.*'号也会报错，注意思路取值是筛选单词与边界相关。
>>> re.match(r".*\Bver\B", "hoverabc").group()
```

```
'hover'


>>> re.match(r".*\Bver\B", "ho verabc").group()
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    re.match(r".*\Bver\B", "ho verabc").group()
AttributeError: 'NoneType' object has no attribute 'group'

# 相同的内容，使用\B进行取反操作， 匹配 bver 开头结尾的单词
>>> re.match(r".*\Bver\B", "hover abc").group()
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    re.match(r".*\Bver\B", "hover abc").group()
AttributeError: 'NoneType' object has no attribute 'group'

# 相同的内容，使用\B进行取反操作， 匹配 bver 开头结尾的单词
>>> re.match(r".*\Bver\B", "ho ver abc").group()
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    re.match(r".*\Bver\B", "ho ver abc").group()
AttributeError: 'NoneType' object has no attribute 'group'
```

\b与\B之间的区别是相互的，互相的对立面，一个单词，让我们对这个单词进行头尾做对应的限定时，

\b更多的是针对整个被正则\b限定包含的单词在搜索的环境中是否存在对应的精准值，再上方看到，是使用了空格将独立的单词进行隔离包裹出完整的内容，作为最后的取值结果标准。

\B相反，是以非空格形式，将我们的被正则\B限定包含的单词在搜索的环境中是否存在对应的，再上方看到，是使用了非空格将独立的单词进行隔离包裹出完整的内容，作为最后的取值结果标准。

9、表示分组

分组组合操作

字符	功能
|	匹配左右任意一个表达式
(ab)	将括号中字符作为一个分组
\number	匹配和前面索引为number的分组捕获到的内容一样的字符串
`(?P<name>)`	分组起别名
(?P=name)	引用别名为name分组匹配到的字符串

'|'用法

A|B，A 和 B 可以是任意正则表达式，创建一个正则表达式，匹配 A 和 B. 任意个正则表达式可以用 `'|'` 连接。(它也可以在组合内使用)。扫描目标字符串时， `'|'` 分隔开的正则样式从左到右进行匹配。当一个样式完全匹配时，这个分支就被接受。意思就是，一旦 *A* 匹配成功， *B* 就不再进行匹配，即便它能产生一个更好的匹配。或者说， `'|'` 操作符绝不贪婪。 如果要匹配 `'|'` 字符，使用 `\|`， 或者把它包含在字符集里，比如 [|]

可以看一个例子：匹配出0-100之间的数字

```
# 导包
>>> import re

# 我们通过中括号，[限定十位上为1到9之间的正整数]，使用? 号限定前面的数是0或1个，\d表示0到9之间任意的数字
# 通过判断筛选数字为'8'
>>> ret = re.match("[1-9]?\d","8")
>>> ret.group()
'8'

# 通过判断筛选数字为'78'
>>> ret = re.match("[1-9]?\d","78")
>>> ret.group()
'78'

# 通过判断筛选数字为'0'，这里我们十位上的判断失效，使用\d判断值为'0'
>>> ret = re.match("[1-9]?\d","08")
>>> ret.group()
'0'

# 那这个时候我们通过$作为限定文本的结尾发现以\d作为结尾，那此时我们的但是数字结尾前数字0不匹配前一位
# 其实我们数字 输出 08 肯定不是想要的，这里只是针对测试
>>> ret = re.match("[1-9]?\d$","08")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'

# 这种错误存在相同类型的错误，我们的数字限定与实际的0至99的精准值，
>>> ret = re.match("[1-9]?\d$","100")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'

# 但是这里我们就是使用上了，限定本文结尾，并使用 | 进行分支匹配，我们看是是可以成功的，他选择了左方的表达式
>>> ret = re.match("[1-9]?\d$|100","8")
>>> ret.group()
'8'

>>> ret = re.match("[1-9]?\d$|100","78")
>>> ret.group()
'78'

# 当继续匹配100时 依旧使用 | 进行分支匹配，我们看是可以成功的，通过右方的表达式进行匹配结果
>>> ret = re.match("[1-9]?\d$|100","100")
>>> ret.group()
'100'
```

```
# 当我们再去匹配这个 "08" 字符是依然是存在问题。
>>> ret = re.match("[1-9]?\d$|100","08")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

'()' 用法

（组合），匹配括号内的任意正则表达式，并标识出组合的开始和结尾。匹配完成后，组合的内容可以被获取，并可以在之后用 `\number` 转义序列进行再次匹配，之后进行详细说明。要匹配字符 `'('` 或者 `')'`，用 `\(` 或 `\)`，或者把它们包含在字符集合里: `[(]`，`[)]`.

直接来看代码

```
# 导包
>>> import re

# 这里常规测试邮箱，无边界限定，匹配正常
>>> ret = re.match("\w{4,20}@163\.com", "test@163.com")
>>> ret.group()
'test@163.com'

# 在括弧中加入了分支进行判断，根据结果判断，也匹配到结果，
>>> ret = re.match("\w{4,20}@(163|126|qq)\.com", "test@126.com")
>>> ret.group()
'test@126.com'

# 如法炮制在匹配对应的分支判断下，在小括号中的分支，留下对应的其中一组结果。
>>> ret = re.match("\w{4,20}@(163|126|qq)\.com", "test@qq.com")
>>> ret.group()
'test@qq.com'

# 只有当不在对应的值内匹配即存在无法识别的问题
>>> ret = re.match("\w{4,20}@(163|126|qq)\.com", "test@gmail.com")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

'\number' 用法

上面说到在（）分组中，可以只用 \number 进行转义操作

了解 \ 可以调节转义字符，但是当后面加上数字，则存在特殊序列，由 `'\'` 和一个字符组成的特殊序列。

匹配数字代表的组合。每个括号是一个组合，组合从1开始编号。比如 `(.+) \1` 匹配 `'the the'` 或者 `'55 55'`, 但不会匹配 `'thethe'` (注意组合后面的空格)。这个特殊序列只能用于匹配前面99个组合。如果 *number* 的第一个数位是0， 或者 *number* 是三个八进制数，它将不会被看作是一个组合，而是八进制的数字值。在 `'['` 和 `']'` 字符集合内，任何数字转义都被看作是字符。

例如我这里有个内容，需要匹配出`<html>hh</html>`这种标签，我想熟悉的话已经写出来了，看代码

```
# 导包
>>> import re

# 进行匹配，首先是首位的标签，头部 <[a-zA-Z]*> 中间 \w*  尾部 <[a-zA-Z]*>，完成匹配
>>> ret = re.match("<[a-zA-Z]*>\w*</[a-zA-Z]*>", "<html>hh</html>")
>>> ret.group()
'<html>hh</html>'

# 进行匹配，首先是首位的标签，头部 <[a-zA-Z]*> 中间 \w*  尾部 <[a-zA-Z]*>,
# 但是匹配中我们发现，最后一个标签存在一定的问题，无法满足前后一致
>>> ret = re.match("<[a-zA-Z]*>\w*</[a-zA-Z]*>", "<html>hh</htmlbalabala>")
>>> ret.group()
'<html>hh</htmlbalabala>'

# 那此时我们明白，无论你的前面是何种类型的标签，后方结束标签理论上是需要一致的匹配结构
# 那这里就引出一个分组的概念，每个括号是一个组合，组合从1开始编号，那我们将操作范围限定我们的头部标签，
尾部进行选择编号
# <([a-zA-Z]*)>\w*</\1>  这里面的\1 既是对应标签的一种
>>> ret = re.match(r"<([a-zA-Z]*)>\w*</\1>", "<html>hh</html>")
>>> ret.group()
'<html>hh</html>'

# 可以看到经过组合编号进行同步的操作，完全避免了标签不同的情况
>>> ret = re.match(r"<([a-zA-Z]*)>\w*</\1>", "<html>hh</htmlbalabala>")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

' (?P) (?P=name) ' 用法

?P<要起的别名> (?P=起好的别名)

(?P)：（命名组合）类似正则组合，但是匹配到的子串组在外部是通过定义的 *name* 来获取的。组合名必须是有效的Python标识符，并且每个组合名只能用一个正则表达式定义，只能定义一次。一个符号组合同样是一个数字组合，就像这个组合没有被命名一样。

命名组合可以在三种上下文中引用。如果样式是 `(?P<quote>['"]).*?(?P=quote)` （也就是说，匹配单引号或者双引号括起来的字符串):

引用组合 "quote" 的上下文	引用方法
在正则式自身内	`(?P=quote)` (如示) `\1`
处理匹配对象 *m*	`m.group('quote')``m.end('quote')` (等)
传递到 `re.sub()` 里的 *repl* 参数中	`\g<quote>``\g<1>``\1`

(?P=name) 反向引用一个命名组合；它匹配前面那个叫 *name* 的命名组中匹配到的串同样的字串。

```
# 导包
>>> import re

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)

# 进行通过别名的方法解决问题
>>> ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "
<html><h1>www.mashibin.com</h1></html>")
>>> ret.group()
'<html><h1>www.mashibin.com</h1></html>'

# 这里起了别名，但是针对匹配的对应的字符串并没有对应的内容，无法完成匹配，所以错误
>>> ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "
<html><h1>www.mashibin.com</h2></html>")
>>> ret.group()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    ret.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

注意：`(?P<name>)`和`(?P=name)`中的字母p大写

10、RE模块高级使用

search

作用：扫描字符串，查找正则表达式模式产生匹配的第一个位置，并返回对应的匹配对象。如果字符串中没有位置与模式匹配，则返回None; 注意，这不同于在字符串的某个点上找到长度为零的匹配。

需求：匹配出文章阅读的次数

代码：

```
# 导包
>>> import re

# match方法
>>> ret = re.match('\d+','阅读次数为 9999 次')
>>> print(ret)
None

# search方法
>>> ret = re.search('\d+','阅读次数为 999 次')
>>> print(ret)
<re.Match object; span=(6, 9), match='999'>
```

findall

参数：re.findall(pattern, string, flags=0)

作用：pattern 在 string 里所有的非重复匹配，返回为一个迭代器 iterator保存了 匹配对象 。 *string* 从左到右扫描，匹配按顺序排列。空匹配也包含在结果里。也就是说以字符串列表的形式返回所有的非重叠的匹配结果。从左到右扫描字符串，并按照找到的顺序返回匹配项，如果存在一个或多个字符串,返回一个列表的组织;如果匹配表达式中有多个分组，这将是一个元组列表。结果中包含空匹配项。

注意：在 3.7 版更改: 非空匹配现在可以在前一个空匹配之后出现了

需求：统计出python、c、c++相应文章阅读的次数

```
# 导包
>>> import re

# 匹配多组，返回列表
>>> list1 = re.findall(r'\d+',"阅读次数C:129 Python:999 C++:99")
>>> print(list1)
['129', '999', '99']
```

sub

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

•	pattern:  是re.compile()方法生成Pattern类型，也就是索要匹配的模式

•	repl :  可以是一段字符串，或者是一个方法

•	string:  需要被匹配和替换的原字符串

•	count:  指的是最大的可以被替换的匹配到的字符串的个数，默认为0，就是所有匹配到的字符串

•	flags :  标志位
```

作用：返回repl字符串或者调用repl函数而得到的字符串。如果没有找到模式，则返回字符串不变

需求：将匹配到的阅读次数加1

```
# 导包
>>> import re

# 匹配对应的
>>> ret = re.sub(r"\d+", '998', "python = 997")
>>> print(ret)
python = 998
```

这里我们就可以使用上我们的函数进行做更改

```
# -*- coding: UTF-8 -*-
# 文件名  : zzbds_3_3_1.py

# 导包
import re

# 增加函数
def add(temp):
    str_num = temp.group()
    num = int(str_num) + 1
    return str(num)

# 匹配增加
ret = re.sub(r"\d+", add, "python = 997")
print(ret)


# 结果: python = 998

# 匹配增加
ret = re.sub(r"\d+", add, "python = 99")
print(ret)

# 结果: python = 100
```

通过添加函数，进行增加或者对数据进行处理

split

```
参数: re.split(pattern, string, maxsplit=0, flags=0)

      pattern: 相当于str.split()中的sep，分隔符的意思，不但可以是字符串，也可以为正则表达式: '[ab]'，表示的意思就是取a和b的任意一个值

      string: 要进行分割的字符串

      maxsplit: 分割的最大次数，这个参数和str.split()中有点不一样,默认值为0，表示分割次数无限制，能分几次分几次；取负数，表示不分割；若大于0，表示最多分割maxsplit次；

      flags: 该参数可以用来修改pattern表达式的功能，比如忽略大小写 re.IGNORECASE(简写: re.I)，即当flags = re.IGNORECASE ,pattern = [A-Z]不但能匹配到大写字母，也能匹配到小写字母。
```

参考代码：

```
# 导包
>>> import re

# 正则分割，这里是大写W，分割的是 非字母数字下划线
>>> 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.']

# 表示通过使用字母分割，这里加上re.IGNORECASE表示忽略大小写分割
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

需求：切割字符串“info:xiaoZhang 33 shandong”

```
# 导包
>>> import re

# 通过分支选择分组，以：号或者空字符匹配分割
>>> ret = re.split(r":| ","info:xiaoZhang 33 shandong")
>>> print(ret)
['info', 'xiaoZhang', '33', 'shandong']
```

练习

从下面题中选择部分，进行练习

```
1、验证账号是否合法(字母开头，允许5-16字节，允许字母数字下划线)。
2、验证密码是否合法(以字母开头，长度在6~18之间，只能包含字母、数字和下划线)
3、匹配是否全是汉字：
4、验证日期格式（2020-09-10）
5、验证身份证号码。
```

11、贪婪与非贪婪

什么是贪婪模式?

Python里数量词默认是贪婪的， 总是尝试匹配尽可能多的字符

什么是非贪婪?

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

```
# 贪婪模式，.+中的'.'会尽量多的匹配

>>> import re

# 我们需要的结果中相比，符合规范，在限定类型数据是，我们会将特定需要的一些字符做一些匹配，
# 例如下面的更好的是将电话与描述之间进行分割筛选，我们存在使用（）进行了两组分组，但实际匹配没变，知识理清规则
>>> ret = re.match(r'(.+)(\d+-\d+-\d+)','This is my tel:133-1234-1234')
>>> print(f'{ret.group(1)},   {ret.group(2)}')
This is my tel:13,   3-1234-1234

# 这里我们通过？进行将贪婪转换成非贪婪模式
>>> ret = re.match(r'(.+?)(\d+-\d+-\d+)','This is my tel:133-1234-1234')
>>> print(f'{ret.group(1)},   {ret.group(2)}')
This is my tel:,   133-1234-1234
```

正则表达式模式中使用到通配字，那它在从左到右的顺序求值时，会尽量“抓取”满足匹配最长字符串，在我们上面的例子里面，“.+”会从字符串的启始处抓取满足模式的最长字符，其中包括我们想得到的第一个整型字段的中的大部分，“\d+”只需一位字符就可以匹配，所以它匹配了数字“4”，而“.+”则匹配了从字符串起始到这个第一位数字4之前的所有字符。

解决方式：非贪婪操作符“？”，这个操作符可以用在"*","+","?"的后面，要求正则匹配的越少越好。