# 正则表达式.md

## 一. 概念

以一个字符串来描述一类字符串;  
描述了模式的重复或者表述多个字符，按照某种模式匹配一系列有相似特征的字符串；  
**设计思想：**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，我们就认为它“匹配”了，否则，该字符串就是不合法的；

python术语中，两种完成模式匹配（pattern-matching）的方式：

1. 搜索（searching）：在字符串任意部分中搜索匹配的模式，通过 search() 函数来实现；
2. 匹配（matching） ：指判断一个字符串能否从起始处完全或部分的匹配某个模式，通过 match() 函数来实现;

注: 当严格讨论与字符串中模式相关的正则表达式时, 全部使用术语 "匹配"; 至于"搜索" 和 "匹配"两种方式按照python怎么完成模式匹配来进行区分;

在正则表达式中，如果直接给出字符，就是精确匹配。  
按照某种模式来进行匹配的表达式, 姑且叫模式匹配.

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| python | python |
| [a-zA-Z0-9]\* | abc123 |

## 二. 特殊符号和字符

列举常见的形成正则表达式的特殊符号和字符: 参见: [附表一: 常见正则表达式符号和特殊字符](#_附表一:_常见正则表达式符号和特殊字符)

## 三. 常见匹配样式

### 1.使用择一匹配符号匹配多个正则表达式模式 -- |

择一匹配符 - | : 用于分割不同的正则表达式, 匹配时从多个模式中选择其一

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| at|home | at / home |
| red|green | red / green |

### 2.匹配任意单个字符 -- 点号: .

点号 - . : 匹配除了换行符\n以外的任意字符(Python正则表达式中有一个编译标记[S或者DOTALL],该标记能够推翻这个限制,使点号能够匹配换行符)

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| f.o | 匹配f和o之间的任意字符,fqo / f9o .. |
| .. | 匹配任意两个字符 |

注: 如果要显示匹配一个句点/句号符号本身,必须使用反斜杠转义句点符号 -- `\.`; 其余拥有特殊含义的字符匹配同样需要转义;

### 3.从字符串起始或者结尾或者单词边界匹配

脱字符 - ^ : 匹配字符串的开始位置; 也可以使用特殊字符 \A 来匹配(一般仅用于那些没有脱字符的键盘);

美元符号 - $ : 匹配字符串的末尾位置; 也可以使用特殊字符 \Z 来表示

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| ^from | 匹配任意以from作为起始的字符串 |
| /bin/test$ | 匹配任意以/bin/test作为结尾的字符串 |
| ^/bin/test$ | 任何由单独的字符串/bin/test构成的的字符串 |

\b : 用来匹配一个单词的边界;即如果一个字符串中包含以模式开始的单词，则匹配成功；

\B : 用来匹配一个不包含以模式开始的单词但包含模式的字符串;

注: 对于要搜寻指定位置或方位的情况下，应该使用搜索模式；否则会导致匹配可能不正确, 即 search(pattern, string, flags=0);

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| the | 任何包含the的字符串 |
| \bthe | 任何包含以the开头单词的字符串 |
| \Bthe | 任何包含the 但并没有包含以the开头的字符串 |

### 4.创建字符集

句点虽然可以匹配任意符号, 但通常,我们仅需要匹配某些特定字符, 此时需要使用到字符集 -- [aeiou]   
表示能够匹配方括号中包含的任意字符;  
字符集单独使用仅能匹配单个字符, 如果要匹配多个字符还是应该使用择一匹配;

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| b[aeiou]t | bat、bet、bit、bot、but |
| [ae][io][up] | 第一个字符为'a'或'e',第二个字符为'i'或'o',第三个字符为'u'或'p' |
| \Bthe | 任何包含the 但并没有包含以the开头的字符串 |

### 5.限定范围和否定

字符集要匹配指定的字符范围,需要在方括号中两个符号之间用连字符 - -连接,用于指定一个字符范围,如 a-z, A-Z;  
如果脱字符 - ^ 紧跟在左括号后面,这个符号就表示不匹配给定的字符集中的任何一个字符;

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| z.[0-9] | z后面跟一个任意字符,然后跟一个0-9之间的数字 |
| [r-u][env-y][us] | 第一个字符在r-u之间,第二个字符为e,n,v,w,x,y中一个,第三个字符为u 或 s |
| [^aeiou] | 非元音字母 |
| ["-a] | 在一个ASCII系统中,所有位于`"`和`a`之间的字符 |

### 6.使用闭包操作符实现存在性和频数匹配

\* : 匹配其左边的正则表达式出现 0 次 或者 n 次的情况 -- Kleene闭包;  
+ : 匹配 1 次 或者 n 次出现的正则表达式 -- 正闭包操作符;  
? : 匹配 0 次 或者 1 次出现的正则表达式;  
{} : 精确或给出一定范围的匹配次数 -- {N} / {N, M}

注: 如果问号紧跟在任何使用的闭包操作符的匹配后面,它将直接要求正则表达式引擎匹配尽可能少的次数;  
当模式匹配使用分组操作符时,正则表达式引擎将试图"吸收"匹配该模式的尽可能多的字符 --- 贪婪匹配  
问号要求正则表达式引擎去偷懒,如果可能,就在当前的正则表达式中尽可能少的匹配字符,留下尽可能多的字符给后面的模式(如果存在)

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| [dn]ot? | 第一个字母为d/n, 紧跟一个o, 最后再跟最多一个t : do/no/dot/not |
| [0-9]{1,3} | 0-9之间的一个数字重复1-3次 |
| [KQRBNP][a-h][1-8]-[a-h][1-8] | 在"长代数"标记法中,表示国际象棋合法的期盼移动,即KQRBNP一个字母后加上a1-h8之间的棋盘坐标 |

### 7.表示字符集的特殊字符

\d : 匹配任何十进制数字 - [0-9]   
\w : 表示全部字母数字的字符集 - [A-Za-z0-9]  
\s : 表示空格字符  
对应大写字母表示取反;

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| \w+-\d+ | 一个由数字字母组成的字符串和一串由一个连字符分割的数字 |
| \d{3}-\d{3}-\d{4} | 美国电话号码格式,800-555-1212 |

### 8.使用圆括号指定分组

当不仅想要知道整个字符串是否匹配我们的标准，而且想要知道能否提取任何已经成功匹配的特定字符串或者子字符串时, 只要用一对圆括号- () 包裹任何正则表达式;  
当使用正则表达式时，一对圆括号可以实现以下任意一个（或者两个）功能：

• 对正则表达式进行分组；  
• 匹配子组。

运用场景:

* 当有两个不同的正则表达式而且想用它们来比较同一个字符串时;
* 对正则表达式进行分组可以在整个正则表达式中使用重复操作符（而不是一个单独的字符或者字符集）。

匹配模式的子字符串可以保存起来供后续使用。这些子组能够被同一次的匹配或者搜索重复调用，或者提取出来用于后续处理;

问: 为什么要使用分组?

答: 在很多时候除了进行匹配操作以外，我们还想要提取所匹配的模式。例如，如果决定匹配模式\w+-\d+，但是想要分别保存第一部分的字母和第二部分的数字，该如何实现？  
我们可能想要这样做的原因是，对于任何成功的匹配，我们可能想要看到这些匹配正则表达式模式的字符串究竟是什么。  
如果为两个子模式都加上圆括号，例如(\w+)-(\d+)，然后就能够分别访问每一个匹配子组。  
如果使用择一匹配通过编写代码来判断是否匹配，还需要执行另一个单独的程序（该程序也需要另行创建）来解析整个匹配用于提取两个部分, 而这本身即re模块支持的一个特性。  
因此更倾向于使用子组.

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| \d+(\.\d\*)? | 表示简单浮点数的字符串；也就是说，任何十进制数字，后面可以接一个小数点和零个或者多个十进制数字，例如“0.004”、“2”、“75.”等 |
| (Mr?s?\.)?[A-Z][a-z]\*[A-Za-z-]+ | 名字和姓氏，以及对名字的限制（如果有，首字母必须大写，后续字母小写），全名前可以有可选的“Mr.”、“Mrs.”、“Ms.”或者“M.”作为称谓，以及灵活可选的姓氏，可以有多个单词、横线以及大写字母 |

### 9.扩展表示法 - 了解

扩展表示法，它们是以问号开始（?…）  
通常用于在判断匹配之前提供标记，实现一个前视（或者后视）匹配，或者条件检查。尽管圆括号使用这些符号，但是只有（?P）表述一个分组匹配。所有其他的都没有创建一个分组;

|  |  |
| --- | --- |
| **正则表达式模式** | **匹配的字符串** |
| (?:\w+\.)\* | 以句点作为结尾的字符串，例如“google.”、“twitter.”、“facebook.”，但是这些匹配不会保存下来供后续的使用和数据检索 |
| (?=.com) | 如果一个字符串后面跟着“.com”才做匹配操作，并不使用任何目标字符串 |

## 正则表达式和Python语言 -- 模块re

re 模块在古老的 Python 1.5 版中引入，用于替换那些已过时的 regex 模块和 regsub 模块;  
这两个模块在 Python 2.5 版中移除，而且此后导入这两个模块中的任意一个都会触发 ImportError 异常。  
re 模块支持更强大而且更通用的 Perl 风格（Perl 5 风格）的正则表达式，该模块允许多个线程共享同一个已编译的正则表达式对象，也支持命名子组;  
re模块: 核心函数详细参见: [附表二:常见正则表达式函数和方法](#_附表二:_常见正则表达式函数和方法)

### 1.使用 compile()函数编译正则表达式

在模式匹配发生之前，正则表达式模式必须编译成正则表达式对象。由于正则表达式在执行过程中将进行多次比较操作，因此强烈建议使用预编译，这样匹配的性能会明显提升。  
re.compile(pattern，flags = 0)能够提供此功能。flags取值见附表二中常用的模块属性;  
正则表达式的编译并不是必须的;编译过就使用方法, 没有编译的就使用函数;而re模块中方法和函数的名字是相同的;

*# coding: utf-8*

import re

pattern = re.compile(r'^the', re.I)

print(0, pattern.search('Thebbbyrhguigthe')) *# (0, <\_sre.SRE\_Match object at 0x000000000592BE68>)*

### 2.匹配对象以及 group()和 groups()方法

匹配对象: 成功调用 match()或者 search()返回的对象。匹配对象有两个主要的方法：group()和 groups()。  
group() 要么返回整个匹配对象，要么根据要求返回特定子组;  
groups() 则仅返回一个包含唯一或者全部子组的元组。  
如果没有子组的要求，那么当group()仍然返回整个匹配时，groups()返回一个空元组。

*# coding: utf-8*

import re

pattern = re.compile(r'^the', re.I) *# 没有分组要求*

pattern1 = re.compile(r'(^the)', re.I) *# 有分组要求*

*# 测试 group() 和 groups() 方法*

gr = pattern.search('the name is leiax00')

gr1 = pattern1.search('the name is leiax00')

print(gr.group()) *# the*

print(gr.groups()) *# ()*

print(gr1.group()) *# the*

print(gr1.groups()) *# ('the',)*

### 3.使用 match()方法匹配字符串

match()函数试图从字符串的起始部分对模式进行匹配。如果匹配成功，就返回一个匹配对象；如果匹配失败，就返回 None, 匹配对象的 group()方法能够用于显示那个成功的匹配。

*# coding: utf-8*

import re

*# test match()*

match\_pattern = re.compile(r'f.o')

m\_gr = match\_pattern.match('fio')

if m\_gr is not None:

print(m\_gr.group())

try:

print(re.match(r'nam', 'name is leiax00').group())

except AttributeError:

print('fail to match...')

### 4.使用search()在一个字符串中查找模式（搜索与匹配的对比）

search()的工作方式与 match()完全一致，不同之处在于 search()会用它的字符串参数，在任意位置对给定正则表达式模式搜索第一次出现的匹配情况。  
如果搜索到成功的匹配，就会返回一个匹配对象；否则，返回 None。

match 方法要求被匹配的字符串必须以正则表达式对象要求的模式作为起始;  
search 方法只要被匹配的字符串中包含要求模式即可;  
等价的正则表达式对象方法使用可选的pos 和endpos参数来指定目标字符串的搜索范围。

*# test search() and compare match()*

print('======================test search() and compare match()============================')

search\_pattern = re.compile(r'nam')

print('search', re.search(r'nam', 'the name is leiax00')) *# ('search', <\_sre.SRE\_Match object at 0x0000000005892718>)*

print('match', re.match(r'nam', 'the name is leiax00')) *# ('match', None)*

*# pattern对象方法可指定搜索范围*

print('pattern search', search\_pattern.search('the name is leiax00', 0, 5)) *# ('pattern search', None)*

print('pattern search', search\_pattern.search('the name is leiax00', 0, 15)) *# ('pattern search', <\_sre.SRE\_Match object at 0x0000000005892718>)*

print('pattern match', search\_pattern.match('the name is leiax00', 0, 5)) *# ('pattern match', None)*

print('pattern match', search\_pattern.match('the name is leiax00', 0, 15)) *# ('pattern match', None)*

### 5.匹配多个字符串 -- 择一匹配符 |

*# 匹配多个字符串*

print('======================匹配多个字符串============================')

bt = 'bat|bet|bit' *# 正则表达式模式 : bat 、 bet 、 bit*

m = re.match(bt, 'bat') *# 'bat' 是一个匹配*

if m is not None:

print('match0', m.group())

m = re.match(bt, 'He bit me!') *# 对于 'blt' 没有匹配*

if m is not None:

print('match1', m.group())

m = re.search(bt, 'He bit me!') *# 通过搜索查找 'bit'*

if m is not None:

print('search', m.group())

### 6.匹配任何单个字符 -- 点号 .

点号.不能匹配一个换行符\n 或者非字符;

*# 匹配任何单个字符 -- 点号 `.`*

print('======================匹配任何单个字符 -- 点号 `.`============================')

anyend = '.end'

m = re.match(anyend, 'bend') *# 点号匹配 'b'*

if m is not None: print(0, m.group())

'bend'

m = re.match(anyend, 'end') *# 不匹配任何字符*

if m is not None: print(1, m.group())

m = re.match(anyend, '\nend') *# 除了 \n 之外的任何字符*

if m is not None: print(2, m.group())

m = re.search('.end', 'The end.') *# 在搜索中匹配 ' '*

if m is not None: print(3, m.group())

### 7.创建字符集 -- []

比较字符集和择一匹配付的区别

*# 创建字符集 -- `[]`*

print('======================创建字符集 -- `[]`============================')

m = re.match('[cr][23][dp][o2]', 'c3po') *# 匹配 'c3po'*

if m is not None: print(1, m.group()) *# 'c3po'*

m = re.match('[cr][23][dp][o2]', 'c2do') *# 匹配 'c2do'*

if m is not None: print(2, m.group()) *# 'c2do'*

m = re.match('r2d2|c3po', 'c2do') *# 不匹配 'c2do'*

if m is not None: print(3, m.group())

m = re.match('r2d2|c3po', 'r2d2') *# 匹配 'r2d2'*

if m is not None: print(4, m.group()) *# 'r2d2'*

### 8.重复、特殊字符以及分组

*# 重复、特殊字符以及分组 -- ？，+， \*， （）*

print('======================重复、特殊字符以及分组 -- ？，+， \*， （）============================')

patt0 = '\w+@(\w+\.)?\w+\.com' *# 任意字母和数字 @ 0个或1个子域名. 主域名.com*

patt1 = '\w+@(\w+\.)\*\w+\.com' *# 任意字母和数字 @ 任意个子域名. 主域名.com*

re.match(patt0, 'nobody@xxx.com').group()

re.match(patt1, 'nobody@www.xxx.yyy.zzz.com').group()

m = re.match('\w\w\w-\d\d\d', 'abc-123')

if m is not None: print(1, m.group()) *# (1, 'abc-123')*

m = re.match('\w\w\w-\d\d\d', 'abc-xyz')

if m is not None: print(2, m.group()) *#*

m = re.match('(\w\w\w)-(\d\d\d)', 'abc-123')

print(3, m.group()) *# (3, 'abc-123')*

print(4, m.groups()) *# (4, ('abc', '123'))*

print(5, m.group(1)) *# (5, 'abc')*

print(6, m.group(2)) *# (6, '123')*

### 9.匹配字符串的起始和结尾以及单词边界

该操作符更多用于表示搜索而不是匹配，因为 match()总是从字符串开始位置进行匹配

*# coding: utf-8*

import re

print(1, re.match(r'^the', 'thebbbyrhguigthe')) *# (1, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(2, re.search(r'^the', 'thebbbyrhguigthe')) *# (2, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(3, re.match(r'the$', 'bbbyrhguigthe')) *# (3, None)*

print(4, re.search(r'the$', 'bbbyrhguigthe')) *# (4, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(5, re.match(r'\bthe', 'aa thebbbyrhguig')) *# (5, None)*

print(6, re.search(r'\bthe', 'aa thebbbyrhguig')) *# (6, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(7, re.match(r'\bthe', 'thebbbyrhguig')) *# (7, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(8, re.search(r'\bthe', 'thebbbyrhguig')) *# (8, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

print(9, re.match(r'\Bthe', 'thebbbyrhguig')) *# (9, None)*

print(10, re.search(r'\Bthe', 'thebbbyrhguig')) *# (10, None)*

print(11, re.search(r'\Bthe', 'AA thebbbyrhguig')) *# (11, None)*

print(12, re.match(r'\Bthe', 'AAthebbbyrhguig')) *# (12, None)*

print(13, re.search(r'\Bthe', 'AAthebbbyrhguig')) *# (13, <\_sre.SRE\_Match object at 0x0000000004E7BE68>)*

*# print(re.search(r'^the$', 'the'))*

*# print(re.match(r'^the$', 'the'))*

*# print(re.match(r'f.o', 'foo'))*

### 10.使用 findall()和 finditer()查找每一次出现的位置

findall()查询字符串中某个正则表达式模式全部的非重复出现情况。这与 search()在执行字符串搜索时类似;  
不同之处：findall()总是返回一个列表。如果 findall()没有找到匹配的部分，就返回一个空列表，但如果匹配成功，列表将包含所有成功的匹配部分（从左向右按出现顺序排列）。  
finditer() 与 findall()函数类似但是更节省内存的变体; 和返回的匹配字符串相比，finditer()在匹配对象中迭代;   
与 match()和 search()类似，findall()和 finditer()方法的版本支持可选的 pos 和 endpos参数，这两个参数用于控制目标字符串的搜索边界;

*# 使用 findall()和 finditer()查找每一次出现的位置*

print('======================使用 findall()和 finditer()查找每一次出现的位置============================')

print(1, re.findall(r'r.\*r', 'rcrrcar')) *# (1, ['rcrrcar'])*

print(2, re.findall(r'c.r', 'carcur')) *# (2, ['car', 'cur'])*

s = 'This and that.'

print(21, re.findall(r'(th\w+) and (th\w+)', s, re.I)) *# (21, [('This', 'that')])*

print(22, re.finditer(r'(th\w+) and (th\w+)', s, re.I).next().groups()) *# (22, ('This', 'that'))*

print(23, re.finditer(r'(th\w+) and (th\w+)', s, re.I).next().group(1)) *# (23, 'This')*

print(24, re.finditer(r'(th\w+) and (th\w+)', s, re.I).next().group(2)) *# (24, 'that')*

print(25, [g.groups() for g in re.finditer(r'(th\w+) and (th\w+)',s, re.I)]) *# (25, [('This', 'that')])*

### 11.使用 sub()和 subn()搜索与替换

sub()和 subn() 实现搜索和替换功能; 两者几乎一样，都是将某字符串中所有匹配正则表达式的部分进行某种形式的替换。用来替换的部分通常是一个字符串，但它也可能是一个函数，该函数返回一个用来替换的字符串  
subn()还返回一个表示替换的总数，替换后的字符串和表示替换总数的数字一起作为一个拥有两个元素的元组返回。 使用匹配对象的 group（）方法除了能够取出匹配分组编号外，还可以使用\N，其中 N 是在替换字符串中使用的分组编号。

*# 使用 sub()和 subn()搜索与替换*

print('======================使用 sub()和 subn()搜索与替换============================')

*# attn: Mr. Smith*

*#*

*# Dear Mr. Smith,*

*#*

print(re.sub('X', 'Mr. Smith', 'attn: X\n\nDear X,\n'))

print(re.subn('X', 'Mr. Smith', 'attn: X\n\nDear X,\n')) *# ('attn: Mr. Smith\n\nDear Mr. Smith,\n', 2)*

print(re.sub('[ae]', 'X', 'abcdef')) *# XbcdXf*

print(re.subn('[ae]', 'X', 'abcdef')) *# ('XbcdXf', 2)*

*# 使用\N来获取分组的值, N为分组编号, 以下为日期转换*

print(re.sub(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})', r'\2/\1/\3', '2/20/91')) *# 20/2/91*

print(re.sub(r'(\d{1,2})/(\d{1,2})/(\d{2}|\d{4})', r'\2/\1/\3', '2/20/1991')) *# 20/2/1991*

### 12.在限定模式上使用 split()分隔字符串

split() 允许指定分割次数;

*# 在限定模式上使用 split()分隔字符串*

print('======================在限定模式上使用 split()分隔字符串============================')

print(re.split(':', 'str1:str2:str3')) *# ['str1', 'str2', 'str3']*

print(re.split(':', 'str1:str2:str3', 1)) *# ['str1', 'str2', 'str3']*

### 13.扩展符号

*# 扩展符号*

print('======================扩展符号============================')

*# re.I/IGNORECASE: 忽略大小写; 写作`?i`*

*# re.M/MULTILINE: ^ 和 $ 分别匹配目标字符串中行的起始和结尾,而不是严格匹配整个字符串的起始和结尾; 记作: `?m`*

print(1, re.findall(r'(?i)yes', 'yes? Yes. YES!!')) *# (1, ['yes', 'Yes', 'YES'])*

f = re.findall(r'(?im)(^th[\w ]+)', """

This line is the first,

another line,

that line, it's the best

""")

print(2, f) *# ['This line is the first', 'that line']*

*# re.S/DOTALL : 表明点号（.）能够用来表示\n 符号, 记作`?s`*

f = re.findall(r'(?s)th.+', '''

The first line

the second line

the third line

''')

print(3, f) *# (3, ['the second line\nthe third line\n'])*

*# re.X/VERBOSE 该标记允许用户通过抑制在正则表达式中使用空白符（除了在字符类中或者在反斜线转义中）来创建更易读的正则表达式,就是可以换行。 记作 `?x`*

g = re.search(r'''(?x)

\((\d{3})\) # 区号

[ ]

(\d{3})

-

(\d{4})''', '(800) 555-1212').groups()

print(4, g) *# (4, ('800', '555', '1212'))*

*# (?:…)符号将更流行；通过使用该符号，可以对部分正则表达式进行分组，但是并不会保*

*# 存该分组用于后续的检索或者应用。当不想保存今后永远不会使用的多余匹配时，这个符号就非常有用。*

*# (5, ['google.com', 'google.com', 'google.com'])*

print(5, re.findall(r'http://(?:\w+\.)\*(\w+\.com)', 'http://google.com http://www.google.com http://code.google.com'))

*# (6, [('', 'google.com'), ('www.', 'google.com'), ('code.', 'google.com')])*

print(6, re.findall(r'http://(\w+\.)\*(\w+\.com)', 'http://google.com http://www.google.com http://code.google.com'))

*# (?P<name>)通过使用一个名称标识符而不是使用从 1 开始增加到 N 的增量数字来保存匹配，*

*# 如果使用数字来保存匹配结果，我们就可以通过使用\1,\2 ...,\N \来检索。*

*# 而对于这种情况可以使用一个类似风格的\g<name>来检索它们。*

*# (7, {'areacode': '800', 'prefix': '555'})*

print(7, re.search(r'\((?P<areacode>\d{3})\) (?P<prefix>\d{3})-(?:\d{4})', '(800) 555-1212').groupdict())

*# 用于搜索替换 (8, '(800) 555-xxxx')*

print(8, re.sub(r'\((?P<areacode>\d{3})\) (?P<prefix>\d{3})-(?:\d{4})',

'(\g<areacode>) \g<prefix>-xxxx', '(800) 555-1212'))

*# (?P=name) 可以在一个相同的正则表达式中重用模式，而不必稍后再次在（相同）正则表达式中指定相同的模式。*

print(9, re.match(r'(?P<a>\d{3})[-](?P=a)', '999-999').group()) *# 999-999*

*# print(10, re.match(r'(?P<a>\d{3})[-](?P=a)', '999-888').group()) # AttributeError*

*# (?=...) 正向前视断言 -- L00 lei11 匹配 ' lei', 匹配上则取值 L00*

print(10, re.findall(r'\w+(?= lei)', '''L00 lei11

L01 lei12,

L02 lei13

''')) *# (10, ['L00', 'L01', 'L02'])*

*# (?!…) 负向前视断言*

print(11, re.findall(r'(?m)^\s\*(?!baidu)(\w+)', '''baidu.com

sina.com, google.com

tecent.cn

''')) *# (11, ['sina', 'tecent'])*

print(12, [e.group(1) for e in re.finditer(r'(?m)^\s\*(?!baidu)(\w+)', '''baidu.com

sina.com, google.com

tecent.cn

''')]) *# (12, ['sina', 'tecent'])*

*# ?(id/name)Y|N id/name存在则返回Y,否则返回N*

print(13, bool(re.search(r'(?:(x)|y)(?(1)y|x)', 'xy'))) *# (13, True)*

print(14, bool(re.search(r'(?:(x)|y)(?(1)y|x)', 'xx'))) *# (14, False)*

### 14.杂项

\w 和\W 字母数字字符集同时受 re.L/LOCALE 和 Unicode（re.U/UNICODE）标记所影响。  
定义正则表达式字符串时,要使用原始字符串(即: r'xxx'),可避免与ASCII码重复的字符必须进行转译的问题;

## 附表一: 常见正则表达式符号和特殊字符

|  |  |  |
| --- | --- | --- |
| **表示法** | **描述** | **表达式示例** |
| **符号** | | |
| literal | 匹配文本字符串的字面值literal | foo |
| re1|re2 | 匹配正则表达式re1或者re2 | foo|bar |
| . | 匹配任意字符( \n除外 ) | b.b |
| ^ | 匹配字符串起始部分 | ^Dear |
| $ | 匹配字符串终止部分 | /bin/\*sh$ |
| \* | 匹配 0 次或多次前面出现的正则表达式 | [a-zA-Z0-9]\* |
| + | 匹配 1 次或多次前面出现的正则表达式 | [a-z]+\.com |
| ? | 匹配 0 次或 1 次前面出现的正则表达式 | goo? |
| {N} | 匹配 N 次前面出现的正则表达式 | [0-9]{3} |
| {M,N} | 匹配 M ~ N 次前面出现的正则表达式 | [0-9]{1,3} |
| [...] | 匹配来自字符集的任意单一字符 | [aeiou] |
| [..x-y..] | 匹配 x~y 范围内的任意单一字符 | [0-9], [A-Za-z] |
| [^...] | 不匹配此字符集中出现的任何一个字符,包括某一范围的字符(如果在此字符集中出现) | [^aeiou], [^A-Za-z0-9] |
| (\*|+|?|{})? | 用于匹配上面频繁出现/重复出现符号的非贪婪版本 | .\*?[a-z] |
| (...) | 匹配封闭的正则表达式,然后另存为子组 | ([0-9]{3})?, f(oo|u)bar |
| **特殊字符** | | |
| \d | 匹配任何十进制数字, 与[0-9]一致 ( \D 与 \d 相反, 不匹配任何非数值型的数字) | data\d+.txt |
| \w | 匹配任何字母数字字符, 与[A-Za-z0-9]相同 ( \W 与之相反 ) | [A-Za-z\_]\w+ |
| \s | 匹配任何空格字符, 与[\n\t\r\v\f]相同 ( \S 与之相反 ) | of\sthe |
| \b | 匹配任何单词边界 (\B与之相反 ) | \bThe\b |
| \N | 匹配以保存的子组N ( 参见上面的 (...) ) | price:\16 |
| \c | 逐字匹配任何特殊字符c (即仅按照字面意义匹配,不匹配特殊含义) | \. , \\ , \\* |
| \A(\Z) | 匹配字符串的起始(结束) (另见上面的 ^ 和 $ ) | \ADear |
| **扩展表示法** | | |
| (?iLmsux) | 在正则表达式中嵌入一个或者多个特殊 "标记" 参数( 或者通过函数/方法 ) | (?x), (? im) |
| (?:...) | 表示一个匹配不用保存的分组 | (?:\w+\.)\* |
| (?P...) | 像一个仅由 name 标识而不是数字 ID 标识的正则分组匹配 | (?P) |
| (?P=name) | 在同一个字符串中匹配由(?P)分组的之前文本 | (?P=data) |
| (?#...) | 表示注释, 所有内容都被忽略 | (?#comment) |
| (?=...) | 匹配条件是如果...出现在之后的位置,而不适用输入字符串;称作正向前视断言 | (?=.com) |
| (?!...) | 匹配条件是如果...不出现在之后的位置.而不适用输入字符串;称作负向前视断言 | (?!.net) |
| (?<=...) | 匹配条件是如果...出现在之前的位置.而不适用输入字符串;称作正向后视断言 | (?<=800-) |
| (? | 匹配条件是如果...不出现在之前的位置.而不适用输入字符串;称作负向后视断言 | (? |
| (?(id/name)Y|N)) | 如果分组所提供的 id 或者 name(名称) 存在, 就返回正则表达式的条件匹配Y, 如果不存在,就返回N; |N 是可选项 | (?(1)y|x) |

## 附表二: 常见正则表达式函数和方法

|  |  |
| --- | --- |
| **函数/方法** | **描述** |
| **仅仅是 re 模块函数** | |
| compile(pattern，flags = 0) | 使用任何可选的标记来编译正则表达式的模式，然后返回一个正则表达式对象 |
| **re 模块函数和正则表达式对象的方法** | |
| match(pattern ， string ， flags=0) | 试使用带有可选的标记的正则表达式的模式来匹配字符串。如果匹配成功，就返回匹配对象；如果失败，就返回 None |
| search(pattern ， string ， flags=0) | 使用可选标记搜索字符串中第一次出现的正则表达式模式。如果匹配成功，则返回匹配对象；如果失败，则返回 None |
| findall(pattern ， string [, flags] ) | 查找字符串中所有（非重复）出现的正则表达式模式，并返回一个匹配列表 |
| finditer(pattern ， string [, flags] ) | 与 findall()函数相同，但返回的不是一个列表，而是一个迭代器。对于每一次匹配，迭代器都返回一个匹配对象 |
| split(pattern ， string ， max=0) | 根据正则表达式的模式分隔符，split 函数将字符串分割为列表，然后返回成功匹配的列表，分隔最多操作 max 次（默认分割所有匹配成功的位置） |
| **re 模块函数和正则表达式对象方法** | |
| sub(pattern ，repl ， string ， count=0) | 使用 repl 替换所有正则表达式的模式在字符串中出现的位置，除非定义 count，否则就将替换所有出现的位置（另见 subn()函数，该函数返回替换操作的数目） |
| purge() | 清除隐式编译的正则表达式模式 |
| **常用的匹配对象方法（查看文档以获取更多信息）** | |
| group(num=0) | 返回整个匹配对象，或者编号为 num 的特定子组 |
| groups(default=None) | 返回一个包含所有匹配子组的元组（如果没有成功匹配，则返回一个空元组） |
| groupdict(default=None) | 返回一个包含所有匹配的命名子组的字典，所有的子组名称作为字典的键（如果没有成功匹配，则返回一个空字典） |
| **常用的模块属性（用于大多数正则表达式函数的标记）** | |
| re.I、re.IGNORECASE | 不区分大小写的匹配 |
| re.L、re.LOCALE | 根据所使用的本地语言环境通过\w、\W、\b、\B、\s、\S 实现匹配 |
| re.M、re.MULTILINE | ^和$分别匹配目标字符串中行的起始和结尾，而不是严格匹配整个字符串本身的起始和结尾 |
| re.S、rer.DOTALL | “.”（点号）通常匹配除了\n（换行符）之外的所有单个字符；该标记表示“.”（点号）能够匹配全部字符 |
| re.X、re.VERBOSE | 通过反斜线转义，否则所有空格加上#（以及在该行中所有后续文字）都被忽略，除非在一个字符类中或者允许注释并且提高可读性 |