**【Regular Expression】正则表达式基础**

[](https://www.jianshu.com/u/32783e5e809b)

[liqingbiubiu](https://www.jianshu.com/u/32783e5e809b) 关注

2017.04.19 10:07\* 字数 4108 阅读 234评论 0喜欢 0

一 正则字符简单介绍  
**1.1 元字符介绍**  
**"^"** ：^会匹配行或者字符串的起始位置，有时还会匹配整个文档的起始位置。  
**"$"** ：$会匹配行或字符串的结尾  
**"\b"** :不会消耗任何字符只匹配一个位置，常用于匹配单词边界 如 我想从字符串中"This is Regex"匹配单独的单词 "is" 正则就要写成 "\bis\b"  
　　 \b 不会匹配is 两边的字符，但它会识别is 两边是否为单词的边界  
**"\d"**: 匹配数字，  
　　 例如要匹配一个固定格式的电话号码以0开头前4位后7位，如0737-5686123 正则:^0\d\d\d-\d\d\d\d\d\d\d$ 这里只是为了介绍"\d"字符，实际上有更好的写法会在 下面介绍。  
**"\w"**：匹配字母，数字，下划线.  
　　 例如我要匹配"a2345BCD\_\_TTz" 正则："\w+" 这里的"+"字符为一个量词指重复的次数，稍后会详细介绍。  
**"\s"**：匹配空格  
　　 例如字符 "a b c" 正则："\w\s\w\s\w" 一个字符后跟一个空格，如有字符间有多个空格直接把"\s" 写成 "\s+" 让空格重复  
**"."**：匹配除了换行符以外的任何字符  
　　 这个算是"\w"的加强版了"\w"不能匹配 空格 如果把字符串加上空格用"\w"就受限了，看下用 "."是如何匹配字符"a23 4 5 B C D\_\_TTz" 正则：".+"  
**"[abc]"**: 字符组 匹配包含括号内元素的字符  
这个比较简单了只匹配括号内存在的字符，还可以写成[a-z]匹配a至z的所以字母就等于可以用来控制只能输入英文了，

**1.2 几种反义**  
　　写法很简单改成大写就行了，意思与原来的相反，这里就不举例子了  
**"\W"** 匹配任意不是字母，数字，下划线 的字符  
**"\S"** 匹配任意不是空白符的字符  
**"\D"** 匹配任意非数字的字符  
**"\B"** 匹配不是单词开头或结束的位置  
**"[^abc]"** 匹配除了abc以外的任意字符

**1.3 量词**  
　　先解释关于量词所涉及到的重要的三个概念  
贪婪(贪心) 如"\*"字符 贪婪量词会首先匹配整个字符串，尝试匹配时，它会选定尽可能多的内容，如果失败则回退一个字符，然后再次尝试回退的过程就叫做回溯，它会每次回退一个字符，直到找到匹配的内容或者没有字符可以回退。相比下面两种贪婪量词对资源的消耗是最大的，  
懒惰(勉强) 如 "?" 懒惰量词使用另一种方式匹配，它从目标的起始位置开始尝试匹配，每次检查一个字符，并寻找它要匹配的内容，如此循环直到字符结尾处。  
占有 如"+" 占有量词会覆盖事个目标字符串，然后尝试寻找匹配内容 ，但它只尝试一次，不会回溯，就好比先抓一把石头，然后从石头中挑出黄金  
*""(贪婪)*\* 重复零次或更多  
　　　 例如"aaaaaaaa" 匹配字符串中所有的a 正则： "a\*" 会出到所有的字符"a"  
**"+"(懒惰)** 重复一次或更多次  
　　 例如"aaaaaaaa" 匹配字符串中所有的a 正则： "a+" 会取到字符中所有的a字符， "a+"与"a*"不同在于"+"至少是一次而"*" 可以是0次，  
　　 稍后会与"?"字符结合来体现这种区别  
**"?"(占有)** 重复零次或一次  
　　 例如"aaaaaaaa" 匹配字符串中的a 正则 ： "a?" 只会匹配一次，也就是结果只是单个字符a  
　 **"{n}"** 重复n次  
　　 例如从"aaaaaaaa" 匹配字符串的a 并重复3次 正则： "a{3}" 结果就是取到3个a字符 "aaa";  
　 **"{n,m}"** 重复n到m次  
　　 例如正则 "a{3,4}" 将a重复匹配3次或者4次 所以供匹配的字符可以是三个"aaa"也可以是四个"aaaa" 正则都可以匹配到  
**"{n,}"** 重复n次或更多次  
　　 与{n,m}不同之处就在于匹配的次数将没有上限，但至少要重复n次 如 正则"a{3,}" a至少要重复3次  
把量词了解了之后之前匹配电话号码的正则现在就可以改得简单点了^0\d\d\d-\d\d\d\d\d\d\d可以改为"0\d+−\d7" role="presentation" style="margin: 0px; padding: 0px; display: inline; font-style: normal; font-weight: normal; line-height: normal; font-size: 13px; text-indent: 0px; text-align: left; text-transform: none; letter-spacing: normal; word-spacing: normal; word-wrap: normal; white-space: nowrap; float: none; direction: ltr; max-width: none; max-height: none; min-width: 0px; min-height: 0px; border: 0px; position: relative;">可以改为"0\d+−\d7可以改为"0\d+−\d7"。  
这样写还不够完美如果因为前面的区号没有做限定，以至于可以输入很多们，而通常只能是3位或者4位，  
现在再改一下 "^0\d{2,3}-\d{7}"如此一来区号部分就可以匹配3位或者4位的了  
**1.4 懒惰限定符**  
　　*"?"*\* 重复任意次，但尽可能少重复  
　　 如 "acbacb" 正则 "a.\*?b" 只会取到第一个"acb" 原本可以全部取到但加了限定符后，只会匹配尽可能少的字符 ，而"acbacb"最少字符的结果就是"acb"  
　　**"+?"** 重复1次或更多次，但尽可能少重复  
　　 与上面一样，只是至少要重复1次  
　　**"??"** 重复0次或1次，但尽可能少重复  
　　 如 "aaacb" 正则 "a.??b" 只会取到最后的三个字符"acb"  
　　**"{n,m}?"** 重复n到m次，但尽可能少重复  
如"aaaaaaaa" 正则 "a{0,m}" 因为最少是0次所以取到结果为空  
　　**"{n,}?"** 重复n次以上，但尽可能少重复  
如 "aaaaaaa" 正则 "a{1,}" 最少是1次所以取到结果为 "a"

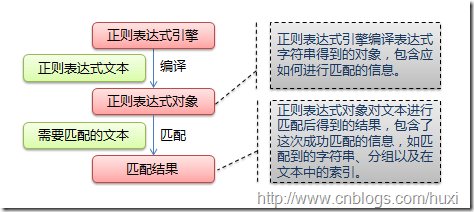
二 正则进阶  
**2.1 捕获分组**  
　　先了解在正则中捕获分组的概念，其实就是一个括号内的内容 如 "(\d)\d" 而"(\d)" 这就是一个捕获分组，可以对捕获分组进行 后向引用 (如果后而有相同的内容则可以直接引用前面定义的捕获组，以简化表达式) 如(\d)\d\1 这里的"\1"就是对"(\d)"的后向引用，那捕获分组有什么用呢看个例子就知道了  
如 "zery zery" 正则 \b(\w+)\b\s\1\b 所以这里的"\1"所捕获到的字符也是 与(\w+)一样的"zery"，为了让组名更有意义，组名是可以自定义名字的"\b(?<name>\w+)\b\s\k<name>\b" 用"?<name>"就可以自定义组名了而要后向引用组时要记得写成 "\k<name>";自定义组名后,捕获组中匹配到的值就会保存在定义的组名里  
下面列出捕获分组常有的用法  
**"(exp)"** 匹配exp,并捕获文本到自动命名的组里  
**"(?<name>exp)"** 匹配exp,并捕获文本到名称为name的组里  
**"(?:exp)"** 匹配exp,不捕获匹配的文本，也不给此分组分配组号  
以下为零宽断言  
**"(?=exp)"** 匹配exp前面的位置  
　　如 "How are you doing" 正则"(?<txt>.+(?=ing))" 这里取ing前所有的字符，并定义了一个捕获分组名字为 "txt" 而"txt"这个组里的值为"How are you do";  
**"(?<=exp)"** 匹配exp后面的位置  
　　如 "How are you doing" 正则"(?<txt>(?<=How).+)" 这里取"How"之后所有的字符，并定义了一个捕获分组名字为 "txt" 而"txt"这个组里的值为" are you doing";  
**"(?!exp)"** 匹配后面跟的不是exp的位置  
　　如 "123abc" 正则 "\d{3}(?!\d)"匹配3位数字后非数字的结果  
**"(?<!exp)"** 匹配前面不是exp的位置  
　　如 "abc123 " 正则 "(?<![0-9])123" 匹配"123"前面是非数字的结果也可写成"(?!<\d)123"

三 常用正则表达式  
**3.1 校验数字的表达式**  
1 数字：^[0-9]*$  
2 n位的数字：^\d{n}$  
3 至少n位的数字：^\d{n,}$  
4 m-n位的数字：^\d{m,n}$  
5 零和非零开头的数字：^(0|[1-9][0-9]*)$  
6 非零开头的最多带两位小数的数字：^([1-9][0-9]*)+(.[0-9]{1,2})?$  
7 带1-2位小数的正数或负数：^(-)?\d+(.\d{1,2})?$  
8 正数、负数、和小数：^(-|+)?\d+(.\d+)?$  
9 有两位小数的正实数：^[0-9]+(.[0-9]{2})?$  
10 有1~3位小数的正实数：^[0-9]+(.[0-9]{1,3})?$  
11 非零的正整数：^[1-9]\d*$ 或 ^([1-9][0-9]*){1,3}$ 或 ^+?[1-9][0-9]*$  
12 非零的负整数：^-[1-9][]0-9"*$ 或 ^-[1-9]\d*$  
13 非负整数：^\d+$ 或 ^[1-9]\d*|0$  
14 非正整数：^-[1-9]\d*|0$ 或 ^((-\d+)|(0+))$  
15 非负浮点数：^\d+(.\d+)?$ 或 ^[1-9]\d*.\d*|0.\d*[1-9]\d*|0?.0+|0$  
16 非正浮点数：^((-\d+(.\d+)?)|(0+(.0+)?))$ 或 ^(-([1-9]\d*.\d*|0.\d*[1-9]\d*))|0?.0+|0$  
17 正浮点数：^[1-9]\d*.\d*|0.\d*[1-9]\d*$ 或 ^(([0-9]+.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*.[0-9]+)|([0-9]*[1-9][0-9]*))$  
18 负浮点数：^-([1-9]\d*.\d*|0.\d*[1-9]\d*)$ 或 ^(-(([0-9]+.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*.[0-9]+)|([0-9]*[1-9][0-9]*)))$  
19 浮点数：^(-?\d+)(.\d+)?$ 或 ^-?([1-9]\d*.\d*|0.\d*[1-9]\d*|0?.0+|0)$  
**3.2 校验字符的表达式**  
1 汉字：^[\u4e00-\u9fa5]{0,}$  
2 英文和数字：^[A-Za-z0-9]+$ 或 ^[A-Za-z0-9]{4,40}$  
3 长度为3-20的所有字符：^.{3,20}$  
4 由26个英文字母组成的字符串：^[A-Za-z]+$  
5 由26个大写英文字母组成的字符串：^[A-Z]+$  
6 由26个小写英文字母组成的字符串：^[a-z]+$  
7 由数字和26个英文字母组成的字符串：^[A-Za-z0-9]+$  
8 由数字、26个英文字母或者下划线组成的字符串：^\w+$ 或 ^\w{3,20}$  
9 中文、英文、数字包括下划线：^[\u4E00-\u9FA5A-Za-z0-9\_]+$  
10 中文、英文、数字但不包括下划线等符号：^[\u4E00-\u9FA5A-Za-z0-9]+$ 或 ^[\u4E00-\u9FA5A-Za-z0-9]{2,20}$  
11 可以输入含有%&',;=?$"等字符：[%&',;=?$\x22]+  
12 禁止输入含有的字符：[^\x22]+  
**3.3 特殊需求表达式**  
1 Email地址：^\w+([-+.]\w+)*@\w+([-.]\w+)*.\w+([-.]\w+)*$  
2 域名：[a-zA-Z0-9][-a-zA-Z0-9]{0,62}(/.[a-zA-Z0-9][-a-zA-Z0-9]{0,62})+/.?  
3 InternetURL：[a-zA-z]+://[^\s]\* 或 ^http://([\w-]+.)+[\w-]+(/[\w-./?%&=]*)?$  
4 手机号码：^(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|18[0|1|2|3|5|6|7|8|9])\d{8}$  
5 电话号码("XXX-XXXXXXX"、"XXXX-XXXXXXXX"、"XXX-XXXXXXX"、"XXX-XXXXXXXX"、"XXXXXXX"和"XXXXXXXX)：^((\d{3,4}-)|\d{3.4}-)?\d{7,8}$  
6 国内电话号码(0511-4405222、021-87888822)：\d{3}-\d{8}|\d{4}-\d{7}  
7 身份证号(15位、18位数字)：^\d{15}|\d{18}$  
8 短身份证号码(数字、字母x结尾)：^([0-9]){7,18}(x|X)?$ 或 ^\d{8,18}|[0-9x]{8,18}|[0-9X]{8,18}?$  
9 帐号是否合法(字母开头，允许5-16字节，允许字母数字下划线)：^[a-zA-Z][a-zA-Z0-9\_]{4,15}$  
10 密码(以字母开头，长度在6~18之间，只能包含字母、数字和下划线)：^[a-zA-Z]\w{5,17}$  
11 强密码(必须包含大小写字母和数字的组合，不能使用特殊字符，长度在8-10之间)：^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$  
12 日期格式：^\d{4}-\d{1,2}-\d{1,2}  
13 一年的12个月(01～09和1～12)：^(0?[1-9]|1[0-2])$  
14 一个月的31天(01～09和1～31)：^((0?[1-9])|((1|2)[0-9])|30|31)$  
15 钱的输入格式：  
16 1.有四种钱的表示形式我们可以接受:"10000.00" 和 "10,000.00", 和没有 "分" 的 "10000" 和 "10,000"：^[1-9][0-9]*$  
17 2.这表示任意一个不以0开头的数字,但是,这也意味着一个字符"0"不通过,所以我们采用下面的形式：^(0|[1-9][0-9]*)$  
18 3.一个0或者一个不以0开头的数字.我们还可以允许开头有一个负号：^(0|-?[1-9][0-9]*)$  
19 4.这表示一个0或者一个可能为负的开头不为0的数字.让用户以0开头好了.把负号的也去掉,因为钱总不能是负的吧.下面我们要加的是说明可能的小数部分：^[0-9]+(.[0-9]+)?$  
20 5.必须说明的是,小数点后面至少应该有1位数,所以"10."是不通过的,但是 "10" 和 "10.2" 是通过的：^[0-9]+(.[0-9]{2})?$  
21 6.这样我们规定小数点后面必须有两位,如果你认为太苛刻了,可以这样：^[0-9]+(.[0-9]{1,2})?$  
22 7.这样就允许用户只写一位小数.下面我们该考虑数字中的逗号了,我们可以这样：^[0-9]{1,3}(,[0-9]{3})*(.[0-9]{1,2})?$  
23 8.1到3个数字,后面跟着任意个 逗号+3个数字,逗号成为可选,而不是必须：^([0-9]+|[0-9]{1,3}(,[0-9]{3})*)(.[0-9]{1,2})?$  
24 备注：这就是最终结果了,别忘了"+"可以用"*"替代如果你觉得空字符串也可以接受的话(奇怪,为什么?)最后,别忘了在用函数时去掉去掉那个反斜杠,一般的错误都在这里  
25 xml文件：^([a-zA-Z]+-?)+[a-zA-Z0-9]+\.[x|X][m|M][l|L]$  
26 中文字符的正则表达式：[\u4e00-\u9fa5]  
27 双字节字符：[^\x00-\xff] (包括汉字在内，可以用来计算字符串的长度(一个双字节字符长度计2，ASCII字符计1))  
28 空白行的正则表达式：\n\s*\r (可以用来删除空白行)  
29 HTML标记的正则表达式：<(\S*?)[^>]*>.*?</\1>|<.*? /> (网上流传的版本太糟糕，上面这个也仅仅能部分，对于复杂的嵌套标记依旧无能为力)  
30 首尾空白字符的正则表达式：\s\*|\s\*$或(\s*)|(\s*$) (可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式)  
31 腾讯QQ号：[1-9][0-9]{4,} (腾讯QQ号从10000开始)  
32 中国邮政编码：[1-9]\d{5}(?!\d) (中国邮政编码为6位数字) 33 IP地址：\d+.\d+.\d+.\d+ (提取IP地址时有用) 34 IP地址：((?:(?:25[0-5]|2[0-4]\d|[01]?\d?\d)\.){3}(?:25[0-5]|2[0-4]\d|[01]?\d?\d))

## 1. 正则表达式基础

### 1.1. 简单介绍

正则表达式并不是Python的一部分。正则表达式是用于处理字符串的强大工具，拥有自己独特的语法以及一个独立的处理引擎，效率上可能不如str自带的方法，但功能十分强大。得益于这一点，在提供了正则表达式的语言里，正则表达式的语法都是一样的，区别只在于不同的编程语言实现支持的语法数量不同；但不用担心，不被支持的语法通常是不常用的部分。如果已经在其他语言里使用过正则表达式，只需要简单看一看就可以上手了。

下图展示了使用正则表达式进行匹配的流程：   


正则表达式的大致匹配过程是：依次拿出表达式和文本中的字符比较，如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。如果表达式中有量词或边界，这个过程会稍微有一些不同，但也是很好理解的，看下图中的示例以及自己多使用几次就能明白。

下图列出了Python支持的正则表达式元字符和语法：     


### 1.2. 数量词的贪婪模式与非贪婪模式

正则表达式通常用于在文本中查找匹配的字符串。Python里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；非贪婪的则相反，总是尝试匹配尽可能少的字符。例如：正则表达式"ab\*"如果用于查找"abbbc"，将找到"abbb"。而如果使用非贪婪的数量词"ab\*?"，将找到"a"。

### 1.3. 反斜杠的困扰

与大多数编程语言相同，正则表达式里使用"\"作为转义字符，这就可能造成反斜杠困扰。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，这个例子中的正则表达式可以使用r"\\"表示。同样，匹配一个数字的"\\d"可以写成r"\d"。有了原生字符串，你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。

### 1.4. 匹配模式

正则表达式提供了一些可用的匹配模式，比如忽略大小写、多行匹配等，这部分内容将在Pattern类的工厂方法re.compile(pattern[, flags])中一起介绍。

## 2. re模块

### 2.1. 开始使用re

Python通过re模块提供对正则表达式的支持。使用re的一般步骤是先将正则表达式的字符串形式编译为Pattern实例，然后使用Pattern实例处理文本并获得匹配结果（一个Match实例），最后使用Match实例获得信息，进行其他的操作。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # encoding: UTF-8  import re    # 将正则表达式编译成Pattern对象  pattern = re.compile(r'hello')    # 使用Pattern匹配文本，获得匹配结果，无法匹配时将返回None  match = pattern.match('hello world!')    if match:      # 使用Match获得分组信息      print match.group()    ### 输出 ###  # hello |

**re.compile(strPattern[, flag]):**

这个方法是Pattern类的工厂方法，用于将字符串形式的正则表达式编译为Pattern对象。 第二个参数flag是匹配模式，取值可以使用按位或运算符'|'表示同时生效，比如re.I | re.M。另外，你也可以在regex字符串中指定模式，比如re.compile('pattern', re.I | re.M)与re.compile('(?im)pattern')是等价的。   
可选值有：

* re.**I**(re.IGNORECASE): 忽略大小写（括号内是完整写法，下同）
* **M**(MULTILINE): 多行模式，改变'^'和'$'的行为（参见上图）
* **S**(DOTALL): 点任意匹配模式，改变'.'的行为
* **L**(LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定
* **U**(UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性
* **X**(VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。以下两个正则表达式是等价的：

|  |  |
| --- | --- |
| 1  2  3  4 | a = re.compile(r"""\d +  # the integral part                     \.    # the decimal point                     \d \*  # some fractional digits""", re.X)  b = re.compile(r"\d+\.\d\*") |

re提供了众多模块方法用于完成正则表达式的功能。这些方法可以使用Pattern实例的相应方法替代，唯一的好处是少写一行re.compile()代码，但同时也无法复用编译后的Pattern对象。这些方法将在Pattern类的实例方法部分一起介绍。如上面这个例子可以简写为：

|  |  |
| --- | --- |
| 1  2 | m = re.match(r'hello', 'hello world!')  print m.group() |

re模块还提供了一个方法escape(string)，用于将string中的正则表达式元字符如\*/+/?等之前加上转义符再返回，在需要大量匹配元字符时有那么一点用。

### 2.2. Match

Match对象是一次匹配的结果，包含了很多关于此次匹配的信息，可以使用Match提供的可读属性或方法来获取这些信息。

属性：

1. **string**: 匹配时使用的文本。
2. **re**: 匹配时使用的Pattern对象。
3. **pos**: 文本中正则表达式开始搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
4. **endpos**: 文本中正则表达式结束搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
5. **lastindex**: 最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组，将为None。
6. **lastgroup**: 最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组，将为None。

方法：

1. **group([group1, …]):**  
   获得一个或多个分组截获的字符串；指定多个参数时将以元组形式返回。group1可以使用编号也可以使用别名；编号0代表整个匹配的子串；不填写参数时，返回group(0)；没有截获字符串的组返回None；截获了多次的组返回最后一次截获的子串。
2. **groups([default]):**   
   以元组形式返回全部分组截获的字符串。相当于调用group(1,2,…last)。default表示没有截获字符串的组以这个值替代，默认为None。
3. **groupdict([default]):**返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。default含义同上。
4. **start([group]):**   
   返回指定的组截获的子串在string中的起始索引（子串第一个字符的索引）。group默认值为0。
5. **end([group]):**返回指定的组截获的子串在string中的结束索引（子串最后一个字符的索引+1）。group默认值为0。
6. **span([group]):**返回(start(group), end(group))。
7. **expand(template):**   
   将匹配到的分组代入template中然后返回。template中可以使用\id或\g<id>、\g<name>引用分组，但不能使用编号0。\id与\g<id>是等价的；但\10将被认为是第10个分组，如果你想表达\1之后是字符'0'，只能使用\g<1>0。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | import re  m = re.match(r'(\w+) (\w+)(?P<sign>.\*)', 'hello world!')    print "m.string:", m.string  print "m.re:", m.re  print "m.pos:", m.pos  print "m.endpos:", m.endpos  print "m.lastindex:", m.lastindex  print "m.lastgroup:", m.lastgroup    print "m.group(1,2):", m.group(1, 2)  print "m.groups():", m.groups()  print "m.groupdict():", m.groupdict()  print "m.start(2):", m.start(2)  print "m.end(2):", m.end(2)  print "m.span(2):", m.span(2)  print r"m.expand(r'\2 \1\3'):", m.expand(r'\2 \1\3')    ### output ###  # m.string: hello world!  # m.re: <\_sre.SRE\_Pattern object at 0x016E1A38>  # m.pos: 0  # m.endpos: 12  # m.lastindex: 3  # m.lastgroup: sign  # m.group(1,2): ('hello', 'world')  # m.groups(): ('hello', 'world', '!')  # m.groupdict(): {'sign': '!'}  # m.start(2): 6  # m.end(2): 11  # m.span(2): (6, 11)  # m.expand(r'\2 \1\3'): world hello! |

### 2.3. Pattern

Pattern对象是一个编译好的正则表达式，通过Pattern提供的一系列方法可以对文本进行匹配查找。

Pattern不能直接实例化，必须使用re.compile()进行构造。

Pattern提供了几个可读属性用于获取表达式的相关信息：

1. pattern: 编译时用的表达式字符串。
2. flags: 编译时用的匹配模式。数字形式。
3. groups: 表达式中分组的数量。
4. groupindex: 以表达式中有别名的组的别名为键、以该组对应的编号为值的字典，没有别名的组不包含在内。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | import re  p = re.compile(r'(\w+) (\w+)(?P<sign>.\*)', re.DOTALL)    print "p.pattern:", p.pattern  print "p.flags:", p.flags  print "p.groups:", p.groups  print "p.groupindex:", p.groupindex    ### output ###  # p.pattern: (\w+) (\w+)(?P<sign>.\*)  # p.flags: 16  # p.groups: 3  # p.groupindex: {'sign': 3} |

实例方法[ | re模块方法]：

1. **match(string[, pos[, endpos]]) | re.match(pattern, string[, flags]):**这个方法将从string的pos下标处起尝试匹配pattern；如果pattern结束时仍可匹配，则返回一个Match对象；如果匹配过程中pattern无法匹配，或者匹配未结束就已到达endpos，则返回None。   
   pos和endpos的默认值分别为0和len(string)；re.match()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。   
   注意：这个方法并不是完全匹配。当pattern结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符'$'。   
   示例参见2.1小节。
2. **search(string[, pos[, endpos]]) | re.search(pattern, string[, flags]):**这个方法用于查找字符串中可以匹配成功的子串。从string的pos下标处起尝试匹配pattern，如果pattern结束时仍可匹配，则返回一个Match对象；若无法匹配，则将pos加1后重新尝试匹配；直到pos=endpos时仍无法匹配则返回None。   
   pos和endpos的默认值分别为0和len(string))；re.search()无法指定这两个参数，参数flags用于编译pattern时指定匹配模式。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # encoding: UTF-8  import re    # 将正则表达式编译成Pattern对象  pattern = re.compile(r'world')    # 使用search()查找匹配的子串，不存在能匹配的子串时将返回None  # 这个例子中使用match()无法成功匹配  match = pattern.search('hello world!')    if match:      # 使用Match获得分组信息      print match.group()    ### 输出 ###  # world |

1. **split(string[, maxsplit]) | re.split(pattern, string[, maxsplit]):**按照能够匹配的子串将string分割后返回列表。maxsplit用于指定最大分割次数，不指定将全部分割。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import re    p = re.compile(r'\d+')  print p.split('one1two2three3four4')    ### output ###  # ['one', 'two', 'three', 'four', ''] |

1. **findall(string[, pos[, endpos]]) | re.findall(pattern, string[, flags]):**搜索string，以列表形式返回全部能匹配的子串。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import re    p = re.compile(r'\d+')  print p.findall('one1two2three3four4')    ### output ###  # ['1', '2', '3', '4'] |

1. **finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):**搜索string，返回一个顺序访问每一个匹配结果（Match对象）的迭代器。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import re    p = re.compile(r'\d+')  for m in p.finditer('one1two2three3four4'):      print m.group(),    ### output ###  # 1 2 3 4 |

1. **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**使用repl替换string中每一个匹配的子串后返回替换后的字符串。   
   当repl是一个字符串时，可以使用\id或\g<id>、\g<name>引用分组，但不能使用编号0。   
   当repl是一个方法时，这个方法应当只接受一个参数（Match对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。   
   count用于指定最多替换次数，不指定时全部替换。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import re    p = re.compile(r'(\w+) (\w+)')  s = 'i say, hello world!'    print p.sub(r'\2 \1', s)    def func(m):      return m.group(1).title() + ' ' + m.group(2).title()    print p.sub(func, s)    ### output ###  # say i, world hello!  # I Say, Hello World! |

1. **subn(repl, string[, count]) |re.sub(pattern, repl, string[, count]):**返回 (sub(repl, string[, count]), 替换次数)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import re    p = re.compile(r'(\w+) (\w+)')  s = 'i say, hello world!'    print p.subn(r'\2 \1', s)    def func(m):      return m.group(1).title() + ' ' + m.group(2).title()    print p.subn(func, s)    ### output ###  # ('say i, world hello!', 2)  # ('I Say, Hello World!', 2) |

**以上就是Python对于正则表达式的支持。熟练掌握正则表达式是每一个程序员必须具备的技能，这年头没有不与字符串打交道的程序了。笔者也处于初级阶段，与君共勉，^\_^**

**另外，图中的特殊构造部分没有举出例子，用到这些的正则表达式是具有一定难度的。有兴趣可以思考一下，如何匹配不是以abc开头的单词，^\_^**

**全文结束**