

模式匹配的进一步问题

- 前面讨论的串匹配基于最简单的字符比较
 - 以常规的字符串作为模式
 - 比较的一方是**模式串**，另一方是一个字符串的**所有可能子串**
 - 匹配中考察的是模式串与目标串的所有可能子串之间的相等关系
- 基本串匹配有很广泛的应用，前面举过一些例子，如
 - 正文编辑器中最常用的操作是查找和替换
 - 网络搜索引擎，基本功能就是在网页中检查检索串的匹配
- 实际使用中，存在着许多不同的场景，如
 - 用一个模式串，在目标串里反复检索，找出一些或者所有出现
 - 在一个目标串里检查是否出现了一组模式串中的任何一个
 - 在一批目标串里检查一个或一组模式串是否出现，等等

模式匹配的进一步问题

- 实际中还经常需要（希望）考虑一些更一般的问题，例如
 - 一个目录下所有以 **.py** 结尾的文件名
 - 文件里所有形为 **href="..."** 的段（**HTML**网页里的网页链接）
 - **DNA**片段里以某碱基段开始以另一碱基段结束的片段
 - 计算机可执行文件中的某种片段模式（例如检查病毒），以一种形式的片段开始到另一片段结束，其中出现了某些片段
 - 等等
- 这种匹配中考虑的不是一个字符串，而是一集字符串
 - 可能有穷，也可能无穷
 - 罗列（枚举）的方式不适合这里的需要，因为可能很多或无穷多
 - 要处理这种匹配问题，就需要考虑字符串集合的描述问题，以及是否属于一个字符串集合的匹配问题

模式匹配的进一步问题

- 有关字符串集合的描述和匹配，需要考虑两个问题：
 1. 怎样描述被考虑的那个串集合？需要一种严格描述方式，能描述很多（所有？）有用的字符串集合。“系统化的”描述方式就是一种描述串检索模式的语言（简单串匹配的“模式语言”就是字符串本身）
 2. 如何（或，是否可能）高效实现所希望的检查（匹配）
- 模式描述语言的功能很强，就可能描述更多更复杂的模式（对应的，字符串集合），但匹配算法的复杂性也会提高。这方面有许多理论结果
 - 模式语言变得比较复杂以后，或许只能做出具有指数复杂性的匹配算法，这种情况使模式语言变得没有实用意义
 - 如果模式语言进一步复杂，模式匹配问题甚至可能变为不可计算问题。也就是说，根本不可能写出完成匹配的算法。这样的描述语言就完全没有实际价值了
 - 有意义的模式描述语言是描述能力和处理效率之间的合理平衡

模式匹配的进一步问题

- 如果大家对 **DOS** 操作系统或者 **Windows** 命令窗口（**cmd**）有些了解，可能会知道描述文件名的“通配符”
 - 在 **Windows** 系统里搜索文件，也会用到
 - **Windows/DOS** 的文件名描述中可以使用两个通配符 ***** 和 **?**
 - 写在文件名字符串里的 **?** 可以与任何实际字符匹配
 - ***** 可与任意一串字符匹配
 - 例：***.py** 与所有以 **py** 为扩展名的文件名匹配
- 在普通字符串的基础上增加通配符，形成了一种功能更强的模式语言
 - 一个模式描述一集字符串，例如 **a?b** 描述所有 **3** 个字符的串，其首字符为 **a**，尾字符为 **b**，中间字符任意
 - 能描述无穷字符串集合，例如 **a*** 描述了所有以 **a** 开头的字符串
- 但，只是加入了通配符的模式语言还不够灵活（描述能力不够强）

正则表达式

- 一种很有意义和实用价值的模式语言称为**正则表达式**（**Regular Expression, regex, regexp, RE, re**），由逻辑学家 **Kleene** 提出
 - 一个具体的正则表达式，描述字符集上的一个字符串集合
- 正则表达式语言的基本成分是字符集里的普通字符，另外还有几种特殊的组合结构（以及表示组合的括号）
 - 正则表达式里的普通字符只与该字符本身匹配
 - 顺序组合 $\alpha\beta$ ：若 α 匹配 s ， β 匹配 t ，那么 $\alpha\beta$ 匹配 st
 - 选择组合 $\alpha|\beta$ ：若 α 匹配 s ， β 匹配 t ， $\alpha|\beta$ 匹配 s 也匹配 t
 - 星号 α^* ：与 0 段或者任意多段与 α 匹配的序列的拼接串匹配
- 例：
 - **abc** 只与串 **abc** 匹配
 - **a(b*)(c*)** 与所有有一个 **a** 之后任意个 **b** 再后任意个 **c** 的串匹配
 - **a((b|c)*)** 与所有有一个 **a** 后任意个 **b** 和 **c** 组成的序列匹配

正则表达式

- 这里不需要通配符
 - 通配符 `?` 可以用 `|` 描述（由于字符集是有穷集）
 - 通配符 `*` 可以通过 `|` 和星号描述
- 正则表达式在实际的信息处理中非常有用
 - 人们以各种形式将其纳入编程语言或者语言的标准库
 - 存在很多不同设计，它们都是理论的正则表达式的子集或变形，基于对易用性和实现效率等方面的考虑，还可能有些扩充
 - 许多脚本语言提供正则表达式功能，一些常规语言正在或计划把正则表达式纳入标准库，**C/C++/Java** 等语言也有正则表达式包
 - 经过在 **Perl** 语言里的精炼，基本形成了一套比较标准的形式
- 可以看到许多有关正则表达式的书籍或文章，把正则表达式说成是程序员必备的重要武器，等等。网上的讨论很热闹，有若干书籍

正则表达式

■ 有关书籍



数据结构和算法 (Python 语言版)：字符串

裘宗燕, 2014-10-26-/7/

Python 正则表达式

- **Python** 的正则表达式功能由标准包 **re** 提供。正则表达式可以帮助我们实现一些复杂的字符串操作。正确使用这个包
 - 需要理解正则表达式的描述规则和效用
 - 理解使用正则表达式的各种方法
- 正则表达式采用 **Python** 字符串的形式描述（引号括起的字符序列）
 - 在用于一些特殊的操作时，一个具有正则表达式形式的字符串代表一种字符串模式，它能与特定的一集字符串匹配
 - 正则表达式的描述形式实际上构成一种特殊的“小语言”
 - 语法：**re** 规定的特殊描述规则
 - 语义：一个正则表达式所描述的那一集字符串
- **Python** 文档 **HOWTO** 里有一节 **Regular Expression HOWTO**。网上有些介绍 **Python** 正则表达式的网页，一些 **Python** 书籍里有讨论

数据结构和算法 (Python 语言版)：字符串

裘宗燕, 2014-10-26-/8/

原始字符串

- 在介绍 **Python** 正则表达式前，先介绍原始字符串（文字量）的概念
 - 原始字符串（**raw string**）是 **Python** 里一种写字符串文字量的形式，其值（和普通文字量一样）就是 **str** 类型的对象
 - 原始字符串的形式是在普通字符串文字量前加 **r** 或 **R** 前缀，如
`R"abcdefg" r"C:\courses\pathon\progs"`
 - 原始字符串里的 **** 不作为换意符，在相应 **str** 对象里原样保留
除了位于单/双引号前的反斜线符号
- 引入原始字符串机制，只是为了使一些字符串的写法简单
`r"C:\courses\pathon\progs"` 的等价写法是：
`"C:\\courses\\pathon\\progs"`
写模式匹配正文里的 **** 时情况更麻烦，匹配一个 **** 需要写 `\\`
有关详情见 **Python** 文档的 **HOWYO**。后面将提到两个常用情况

正则表达式

- **Python** 正则表达式包 **re** 规定了一组特殊字符，称为“元字符”。它们在匹配字符串时起着特殊的作用。这种字符一共 **14** 个
`. ^ $ * + ? \ | { } [] ()`
注意：这些字符在（常规）字符串里都是普通字符（“****”除外），只有在把字符串作为正则表达式使用时，它们有特殊意义
- 非特殊字符称为常规字符，是描述正则表达式的基础
正则表达式串里的常规字符在匹配中只与自己匹配
如果一个正则表达式串只包含常规字符，它就只能与自己匹配。也就是说，常规字符串是最基本的正则表达式
- 在介绍正则表达式元字符的使用之前，先介绍 **re** 包提供的几个操作
可以通过这些操作去使用正则表达式（还有其他方式，后面介绍）
在下面函数说明中，参数表里的 **pattern** 表示描述正则表达式的字符串（模式串），**string** 表示被处理的字符串，**repl** 表示替换串

正则表达式

- 生成正则表达式对象：`re.compile(pattern, flag = 0)`

从 `pattern` 生成对应的正则表达式对象。可用于下面几个操作。例：

```
r1 = re.compile("abc")
```

生成 "abc" 对应的正则表达式对象赋给变量 `r1`

- `re` 包的操作都有 `flag` 选项。`re` 专门提供了一组特殊标记，这里不考虑

- 实际上，下面几个操作都能自动从 `pattern` 串生成正则表达式对象。用 `compile` 生成对象并记入变量，可以避免在重复使用中重复生成。下面函数的 `pattern` 参数都可以用正则表达式串或对象作为实参

- 检索：`re.search(pattern, string, flag = 0)`

在 `string` 里检索与 `pattern` 匹配的子串。如找到就返回一个 `match` 类型的对象；没找到时返回 `None`

`match` 对象里记录成功匹配的相关信息，可以根据需要取出和使用。也可以简单地把它作为一个真值用于逻辑判断

正则表达式

- 匹配：`re.match(pattern, string, flag=0)`

检查 `string` 是否有与 `pattern` 匹配的前缀，匹配成功时返回相应的 `match` 对象，否则返回 `None`。例：

```
re.search( r1, "aaabcbcbabcb") 成功，但
```

```
re.match( r1, "aaabcbcbabcb") 返回 None
```

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

以 `pattern` 作为分割串将 `string` 分段，`maxsplit` 指明分割数，`0` 表示做完整个 `string`。函数返回分割得到的字符串的表。例

```
re.split(' ', "abc abb are not the same")
```

```
得到: ['abc', 'abb', 'are', 'not', 'the', 'same']
```

```
re.split(" ", "1 2 3 4") # 分割出了几个空串
```

```
得到: ['1', '2', '', '3', '', '', '4']
```

正则表达式

- 找到所有匹配串：`re.findall(pattern, string, flags=0)`

本函数返回一个表，其元素按顺序给出 `string` 里与 `pattern` 匹配的（从左到右非重叠的）各个子串

如果模式里只有常规字符，做这种匹配的价值不大，因为结果表里所有的字符串相同。但用一般的正则表达式模式，情况就会不同

- 还有操作后面介绍，下面逐步介绍正则表达式的情况。请注意两点：

- 一种表达式（或元符号）的构造形式（描述形式）

- 这种表达式能匹配怎样的字符串（集合）

- 正则表达式的最基本组合是顺序组合 $\alpha\beta$ ，若 α 匹配 `s`， β 匹配 `t`，那么 $\alpha\beta$ 匹配 `s+t`（`s`, `t` 为字符串，Python 写法 `s+t` 是两个字符串的拼接）

注意，在正则表达式里，同样可以（也常常需要）写普通 Python 字符串使用的换意字符，如 `\n` 表示换行，`\t` 表示制表符等

正则表达式里的空格也是常规字符，它只与自己匹配

字符组

- 字符组表达式 `[...]` 匹配括号中列出的任一个字符

`[abc]` 可以匹配字符 `a` 或 `b` 或 `c`

区间形式 `[0-9]` 是顺序列出的缩写，匹配所有十进制数字字符

`[0-9a-zA-Z]` 匹配所有字母（英文字母）和数字

- `[^...]` 中的 `^` 表示求补，这种模式匹配所有未在括号里列出的字符

`[^0-9]` 匹配所有非十进制数字的字符

`[^ \t\v\n\f\r]` 匹配所有非空白字符（非空格/制表符/换行符）

如果需要在字符组里包括 `^`，就不能放在第一个位置，或者写 `^`；如果需要在字符组包括 `-`，也必须写 `\-` 或 `\]`

- 圆点字符 `.` 匹配任意一个字符

`a..b` 匹配所有以 `a` 开头 `b` 结束的四字符串

`a[1-9][0-9]` 匹配 `a10`, `a11`, ..., `a99`

常用字符组

- 为了方便，**re** 用换意串形式定义了几个常用字符组，包括：
 - `\d`: 与十进制数字匹配，等价于 `[0-9]`
 - `\D`: 与非十进制数字的所有字符匹配，等价于 `[^0-9]`
 - `\s`: 与所有空白字符匹配，等价于 `[\t\v\n\f\r]`
 - `\S`: 与所有非空白字符匹配，等价于 `[^\t\v\n\f\r]`
 - `\w`: 与所有字母数字字符匹配，等价于 `[0-9a-zA-Z]`
 - `\W`: 与所有非字母数字字符匹配，等价于 `[^0-9a-zA-Z]`
- 还有些类似描述，提供这些只是为了使用方便
 - `p\w\w\w` 与 `p` 开头随后三个字母数字字符的串匹配

重复

- 常希望写重复匹配的模式 (部分)，任意次或若干次重复
- 基本重复运算符是 `*`， `α^*` 与 `α` 的 0 次或任意多次出现匹配
 - `re.split('[,]*', s)` 将串按空格和逗号 (任意个) 切分
 - `re.split('[,]*', '1 2, 3 4, , 5')`
 - 得到 `['1', '2', '3', '4', '5']`
 - `re.split('a*', 'abbaaabbdbbabbababbabb')`
 - 得到 `['', 'bb', 'bbdbb', 'bb', 'b', 'bb', 'bb']`
- 注意，`re.match('ab*', 'abbbbbb')` 时，模式可以与 `a` 匹配，与 `ab` 匹配，等等，它究竟匹配那个串？两种可能
 - 贪婪匹配：与有可能匹配的最长子串匹配
 - 在这里 `ab*` 匹配 `abbbbbb`，`*` 运算符做贪婪匹配
 - 非贪婪匹配：与有可能匹配的最短子串匹配

重复

- 与 * 略微不同的重复运算符 + 表示 1 次或多次重复
例：描述正整数的一种简单模式 ' $\backslash d^+$ '，等价于 ' $\backslash d \backslash d^*$ '
- 可选（片段）用 ? 运算符表示
? 表示 0 次或 1 次重复
例，描述整数（表示整数的字符串）的一种简单模式 ' $-? \backslash d^+$ '
- 确定次数的重复用 {n} 表示， $\alpha\{n\}$ 与 α 匹配的串的 n 次重复匹配
描述北京常规的固话号码：' $(010-)?[2-9][0-9]\{7\}$ '
注意：这种表达式描述的通常是实际字符串集合的超集，但可以用
- 注意：上面描述中出现了圆括号，描述 ? 的作用范围
 - *, ?, {3} 有作用范围问题（优先级），它们作用于最小表达式
 - '010-?' 表示其中的 '-' 可选，'(010-)?' 表示整个段可选

重复

- 重复范围用 {m,n} 表示， $\alpha\{m,n\}$ 与 α 匹配的串的 m 到 n 次重复匹配
 $a\{3,7\}$ 与 3 到 7 个 a 构成的串匹配
 $go\{2,10\}gle$ 与 google, gooole, ..., gooooooooooogle 匹配
- 重复范围中的 m 和 n 均可以省略， $\alpha\{,n\}$ 表示 $\alpha\{0,n\}$ ，而 $\alpha\{m, \}$ 表示 $\alpha\{m, \text{infinity}\}$ 。另外几种重复都可以用这个形式表示：
 $\alpha\{n\}$ 等价于 $\alpha\{n,n\}$ ， $\alpha?$ 等价于 $\alpha\{0,1\}$
 α^* 等价于 $\alpha\{0, \text{infinity}\}$ ， α^+ 等价于 $\alpha\{1, \text{infinity}\}$
- * + ? {m,n} 都采取贪婪匹配策略，与被匹配串中最长的合适子串匹配（因为它们可能出现更大的模式里，要照顾上下文的需要）
 - 与这些运算符对应的有一组非贪婪匹配运算符
 - *? +? ?? {m,n}?（各运算符后增加一个问号）的语义与上面几个运算符分别对应，但采用非贪婪匹配（最小匹配）的策略

选择

- 选择运算符 | 描述两种或多种情况之一的匹配。如果 α 或者 β 与一个串匹配，那么 $\alpha|\beta$ 就与之匹配

$a|b|c$ 可以匹配 a 或者 b 或者 c ， $[abc]$ 可以看作其简写。后者更简洁方便，有时还能简写如 $[a-z]$ ，但只能用于单字符选择

`'0+|[1-9]\d*'` 匹配 Python 程序的十进制整数（注意，Python 把负号看作运算符）。如果已知为独立字段，就可以用这个模式。但它会与 `0123` 的前段 `0` 匹配。进一步考虑还有上下文要求（如需排除 `abc123`, `a123b` 里的 `123`），这方面的问题后面考虑

| 的结合力最弱，比顺序组合还弱。上面描述不需要括号

- 实例：

匹配国内固定电话号码：`'0\d{2}-\d{8}|0\d{3}-\d{7,8}'`

注意，这个正则表达式描述的是实际集合的超集，如两位区号实际上只有 `010/020/021/022`，这段可写为 `0(10|20|21|22|23)-\d{8}`，另一段可以精化为 `0[3-9]\d{2}-\d{7,8}`

首尾匹配

- 行首匹配：以 \wedge 符号开头的模式，只能与一行的前缀子串匹配

`re.search('\w^for','books for children')` 得到 `None`

- 行尾匹配：以 $\$$ 符号结尾的模式，只与一行的后缀匹配

`re.search('fish$', 'cats like to eat fishes')` 得到 `None`

- 注意，“一行的”前缀/后缀包括整个被匹配串的前缀和后缀。如串里有换行符，还包括换行符前的子串（一行的后缀）和其后的子串（前缀）

- 串首匹配： $\wedge A$ 开头的模式只与被匹配串的前缀匹配

- 串尾匹配： $A \$$ 结束的模式只与被匹配串的后缀匹配

- 至此我们已经介绍了所有 14 个元字符

应特别提出换意字符 \backslash ，以它作为引导符定义了一批换意元字符，如 $\backslash d$, $\backslash D$ 等。它还用于在模式串里写非打印字符（如 $\backslash t$, $\backslash n$, ...）和 $\backslash \backslash$ 等，在 $[]$ 里写 $\backslash \wedge$, $\backslash -$, $\backslash]$

单词边界

两个换意元字符用于描述特殊子串的境界

- **\b** 描述单词边界，它表示单词边界匹配一个空串。单词是字母数字的连续序列，边界就是非字母数字字符或无字符（串开头/结束）

这里有个糟糕的问题：在 **Python** 字符串中 **\b** 表示退格符，而在 **re** 的正则表达式里 **\b** 表示单词边界。两种办法：

- 将 **** 双写，它表示把 **** 本身送给 **re.compile**，如 `'\\b123\\b'` 不匹配 `abc123a` 等里的 `123`，但匹配 `(123,123)` 里的 `123`

- 用 **Python** 原始字符串，其中的 **** 不换意。上面的模式可写为 `r'\b123\b'`

- 实例：匹配 **Python** 整数的模式可写为 `'\\b(0+|[1-9]\d*)\\b'`

用原始字符串可简单地写 `r'\b(0+|[1-9]\d*)\b'`。例如写

```
re_int = r'\b(0+|[1-9]\d*)\b'
```

单词边界

- 实例：一般的可能带正负号的整数，可以考虑用模式

```
'[+-]?\\b(0+|[1-9]\d*)\\b'
```

但它匹配 `x+5` 里的 `+5`，但不匹配 `3 + - 5` 里的 `- 5`。写这种表达式和使用时，都需要考虑被匹配对象的情况

- 例：求一个 **Python** 程序里出现的所有整数之和

```
def sumInt(fname):
    re_int = '\\b(0+|[1-9]\d*)\\b'
    inf = open(fname)
    if inf == None:
        return 0
    ilist = map(int, re.findall(re_int, inf.read())) # 可改为分行读入
    s = 0
    for n in ilist:
        s += n
    return s
```

边界

- `\B` 是 `\b` 的补，也是匹配空串，但要求相应位置是字母或数字

- 实例：

```
>>> re.findall('py.\B', 'python, py2, py342, py1py2py4')
['pyt', 'py3', 'py1', 'py2']
```

模式里的组（group）

- 正则表达式描述中的另一个重要概念是组（group）

- 圆括号括起的模式段 (...) 也是模式，它与被括号模式匹配的串匹配。但在此同时还确定了一个被匹配的“组”（字符段）

模式中 (...) 确定的组可以在后面使用，要求匹配同样字符串

(?...) 形式的片段有另外的特殊意义，表示另一类特殊模式形式。有关情况细节较多，这里不介绍了

- 被匹配的组可用 `\n` 形式在模式里“引用”，要求匹配同样字符段。这里的 `n` 表示一个整数序号，组从 1 开始编号

- 例：`r'(.{2}) \1'` 可匹配 `'ok ok'` 或 `'no no'`，不匹配 `'no oh'`

- 注意，组编号应该是 `\1`, `\2` 等，但在普通字符串里，`\1` 表示二进制编码为 1（经常可以看到被写成 `0x01`）的那个（特殊）字符，而现在需要模式串里出现 `\1`, `\2` 等

- 为此就需要写 `'(.{2}) \\1'`，或者用原始字符串形式简化写法

其他匹配操作

- `re.fullmatch(pattern, string, flags=0)`

如果整个 `string` 与 `pattern` 匹配则成功并返回相应的 `match` 对象，否则返回 `None`

- `re.finditer(pattern, string, flags=0)`

功能与 `findall` 类似，但返回的不是表而是一个迭代器，使用该迭代器可顺序取得表示各非重叠匹配的 `match` 对象

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

做替换，把 `string` 里顺序与 `pattern` 匹配的各非重叠子串用 `repl` 代换。`repl` 是串则直接代换；另一情况，`repl` 还可以是以 `match` 对象为参数的函数，这时用函数的返回值代换被匹配子串

例：把串 `text`（例如 `Python` 程序）里的 `\t` 都代换为 4 个空格

```
re.sub('\t', '    ', text)
```

- 另外还有几个操作，见 `re` 文档

匹配对象（`match` 对象）

- 许多匹配函数在匹配成功时返回一个 `match` 对象，对象里记录了所完成匹配的有关信息，可以取出使用。下面介绍这方面的情况

- 首先，这样的匹配结果可以用于逻辑判断，成功时得到的 `match` 对象总表示逻辑真，不成功得到的 `None` 表示假。例如

```
match1 = re.search( pt, text)
if match1:
    ... match1 ... text ... # 使用 match 对象的处理操作
```

- `match` 对象提供了一组方法，用于检查与匹配有关的信息。下面介绍一些基本用法，更多信息（包括可选参数）见 `re` 包文档

在下面介绍中，`mat` 表示通过匹配得到的一个 `match` 对象

- 取得被匹配的子串：`mat.group()`

在一次成功匹配中，所用的正则表达式匹配了目标串的一个子串，操作 `mat.group()` 给出这个子串

匹配对象（match 对象）

- 在目标串里的匹配位置：**mat.start()**

得到 **mat** 代表的成功匹配在目标串里的实际匹配位置，这是目标串的一个字符位置（下标）
- 目标串里被匹配子串的结束位置：**mat.end()**

这个位置采用常规表示方式。设 **text** 是目标串，有如下关系：
mat.group() == text[mat.start():mat.end()]
- 目标串里被匹配的区间：**mat.span()**

得到匹配的开始和结束位置形成的二元组
mat.span() == mat.start(), mat.end()
- **mat.re** 和 **mat.string** 分别取得得到这个 **match** 对象所做匹配的正则表达式对象和目标串
- 应用实例见后

正则表达式对象

- 前面说过，**re.compile(pattern)** 生成一个正则表达式对象
 - 这种对象可以反复用于匹配
 - 实际上，正则表达式对象支持一组方法，与直接使用 **re.nnn** 形式调用匹配函数相比，这组方法的功能更多，使用也更灵活
 - 在下面介绍中，**regex** 代表一个正则表达式对象
- 检索：**regex.search(string[, pos[, endpos]])**
 - 在给定的串 **string** 里检索，可以指定开始和结束位置。按 **Python** 惯例，两个位置确定一个左闭右开的区间
 - 默认从头到尾对 **string** 检索，只给了 **pos** 就从那里检索到最后
- 匹配：**regex.match(string[, pos[, endpos]])**
 - 检查给定的串 **string** 是否有与 **regex** 匹配的前缀
 - **pos** 指定开始匹配前缀的位置，**endpos** 给定被匹配段的终点

正则表达式对象

- 完全匹配: `regex.fullmatch(string[, pos[, endpos]])`
 - 检查 `string` 中所指定范围构成的那个子串是否与 `regex` 匹配, 默认范围是整个串
- 下面两个方法与 `re.` 同名操作功能类似, 但都可以指明匹配区间
 - `regex.findall(string[, pos[, endpos]])`
 - `regex.finditer(string[, pos[, endpos]])`
- 下面两个操作与 `re.` 同名操作功能相同
 - 切分: `regex.split(string, maxsplit=0)`
 - 替换: `regex.sub(repl, string, count=0)`
- `regex.pattern` 取得生成 `regex` 所用的模式字符串
- 下面介绍正则表达式的一些使用方式

正则表达式的使用

- 在一些情况中, 目标串里可能有一些 (或许多) 与所考虑的正则表达式匹配的子串, 需要逐个处理。如果处理中不修改目标串, 采用匹配迭代器的方式最方便。编程模式是:

```
re1 = re.compile( pattern ) # 这里写实际的模式串
for mi in re1.finditer( text ) : # text 是被匹配的目标串
    ... mi.group() ... # 取得被匹配的子串, 做所需操作
    ... text[mi.start()] ... text[mi.end()] ... # 基于匹配检查 text
```

注意: 采用这种方式循环处理, 操作中不要修改目标串, 否则可能影响后续匹配。具体情况依赖于操作和 `re` 包的实现, 无法预计
- 需要修改目标串的被匹配子串时, 应首先考虑能否用 `sub` 完成操作
 - 如果被替换的新串可以直接写出, 就用新串作为 `repl` 的实参
 - 如果要代换的新串与被匹配的串有关, 可以按统一规则从被匹配的串构造出来, 就应该定义一个函数生成新串

正则表达式的使用

- 例：假设需要把一个 Python 程序里的变量和函数名都加上 `my_` 前缀，可以考虑下面的做法

```
from keyword import iskeyword

ident = r"\b[a-zA-Z_]\w*\b" # 标识符由字母和 _ 开头

def add_prefix(name):
    return name if iskeyword(name) else "my_" + name

... ..

modified = re.sub( ident, add_prefix, prog_text)

... ..
```

- 处理复杂的匹配和修改情况，需要每次自己确定匹配成功的位置，完成所需操作，然后确定下次继续匹配的起始位置

这种循环应该用 **while** 描述：用一个位置变量 **pos** 记录维持匹配的起始位置位置，在循环的每次迭代中正确更新

展望和总结

- 模式匹配问题还有许多可能扩展：

- 近似匹配

- 串中数据是通过测量得到的，原本就不准确
- 并不需要准确的匹配，近似可以根据应用的需要定义
例如，定义两个串的接近程度，定义一种“距离”

- 其他模式匹配问题，例如，二维或者高维描述中的模式匹配（字符串是一维描述）

- 等等

- 字符串类型和操作

- 构造、拼接、子串替换等都是典型的字符串操作
- 串匹配是许多串操作的基础。存在很多串匹配算法，值得关注
- 正则表达式是完成字符串操作的有用工具