0.1 正则表达式介绍

常用的正则表达式

- ‘^$‘: 表示空白行
- ‘ooo*‘: 至少两个 o。
- ‘g.*g‘: 表示 g. . .g。
- ’t[ae]st’: 搜索 tast 或者 test
- ‘[^$g]oo$‘: 表示 oo 但是 oo 的前面不能为 g
- ‘[a-zA-Z0-9]‘: 表示 a 到 z，A-Z,0-9 之间的字符
- ‘^$[a-z]$‘: 小写字母开头
- ‘^$[^a-zA-Z]$‘: 表示首字符不是英文字符
- ‘\.$‘: 表示小数. 结尾
- ‘$g..d$‘: 表示 gd 之间有两个字符
- ‘[0-9][0-9]*‘: 查找任意数字
- ‘$o\{2\}$‘: 表示查找 o 两次
- ‘$go\{n,m\}g$‘: 表示查找 goog 或者 gooog
- ‘$g[abc]g$‘: 表示查找 gag，gbg 或者 gcg

扩展正则表达式

- ‘$go+d$‘:+ 表示前面的 o 出现了一次以及以上
- ‘go?d‘:? 表示前面的 o 出现一次或者零次，gd 或者 god
- ‘gd|good‘: 表示 gd 或者 good
- ‘g(la|oo)d‘: 表示 glad 或者 good

sed [-nefr] [动作]:

参数:

-n：使用 silent 模式。在一般的 sed 用法中，所有来自 STDIN 的数据一般都会被输出到屏幕上。但是如果加上-n 参数后，则只有警告 sed 处理的行（或者动作）才会被列出来

-e：直接在指令列模式下进行 sed 的动作编辑

-f：直接将 sed 动作写入一个文件内，-f filename 则可以执行 filename 内的 sed 动作

-r :sed 的动作支持的是扩展型正则表达式语法而不是预设的基本正规表达是语法

动作 ： [n1[,n2]]function,n1,n2 不一定存在，一般表达选择进行动作的行数。

function
- a：新增，a 的后面可以接字符串，二这些字符串会在新的一行出现（目前行的下一行）
- c：取代，c 后面可以接字符串，这些字符串可以取代 n1,n2 之间的行
- d：删除，因为是删除，所以 d 后面没有任何东西
- i：插入，i 后面接字符串，而这些字符串在新的行出现 (当前行的上一行)
- p：打印，将摸个选择的数据输出。通常 p 会和参数 sed -n 一起
- s：取代，可以直接进行取代的工作，通常这个 s 可以搭配正则表达式。例如 1,2s/old/new/g

例子

- cat -n test.tex|sed '1,2d': 将 test.tex 文件的第一行和第二行删除，并不改变 test.tex 文件的内容
- cat -n test.tex|sed '1a endfigure': 在第一行后（占据第二行）添加 endfigure
- cat -n test.tex|sed '1a beginfigure\ >endfigure': 在第一行末尾输入 \ 回车然后输入第二行内容
- cat -n test.tex|sed '1,3c documentclass': 将 1,3 行内容替换为 doucumentclass

- cat -n test.tex|sed -n ‘1,3p’ : 将 test.tex 文件的 1-3 行打印出来, 加上 n 参数为了显示最后输出，而不是打印一行输出一下。

awk 工具 awk ’ 条件类型 1 动作 1 条件类型 2 动作 2’filename:awk 乐意处理后续的档案，也可以读取来自前个制定的标准输出。但是 wak 主要处理每一行的字段的内容而预设的字段分隔符为空格键或者 tab 键。

例子

- last|awk ’print $1 ”\t” $3’ : 查看当前登录用户, 每一行都有变量名称, $1 表示第一列,$0 代表全部

变量名称	代表含义
NF	每行拥有的字段总数
NR	目前 awk 所处理的是第几行的数据
FS	目前的分割字符，预设是空格键

表 1: awk 参数

示例:

- last |awk ’print $1 ”\t lines:” $NR ”\t columns:” $NF’: 处理第一行，将一行的第一列取出打印然后输出制表符分割加上自己需要加上的打印信息 lines:, 输出变量行数，然后输出制表符输出列数

awk 的逻辑运算字。

例子

运算单元	代表含义
<	小于
>=	大于等于
<=	小于等于
==	等于
!=	不等于

- ```
  cat /etc/passwd |\
  > awk '{FS=":"} $3<10 {print $1 "\t" $3}'
  ```

  :passwd 中的内容是用: 分隔的，此条命令查看第三列小于 10 的数据，并且; 列出第三列 (第一行没有正确显示，这是因为我们读入第一行的时候，把鞋变量 $1,$2,... 预设还是以空格为分割的，所以虽然定义了 FS=":", 但却仅仅只能在第二行后开始生效)

- ```
  cat /etc/passwd|\
  > awk 'BEGIN {FS=":"} $3<10 {print $1 "\t␣"$3}'
  ```

- ?…这是一个扩展的符号, 第一个字符在'?' 后面决定了深层的语法。扩展通常没有创建一个新的 group,(?P<name>... 时该规则惟一的特例)

- (?aiLmsux) 来自集合'a','i','L','m','s','u','x' 的一个或者多个字母,group 匹配空字符串字符给整个正则表达式设置相关的 flags:re.A,re.I,re.L,re.M,re.S,re.X。如果你洗完桑包含 flags 作为正则表达式的一部分而不是传递一个 flag 参数到 re.compile() 函数这就是很有用的, Flasg 应该首先用在表达式字符串。

  字符集合, 对单个字符给出取值范围,[abc] 表示 a,b,c,[a-z] 表示 a 到 z 的单个字符

(?:...) 非捕获版本的正则括号, 匹配括号中无论什么正则表达式, 但是在执行一个匹配或者查询之后 group 中子字符串匹配不能被获得

\d 数字等价与 [0-9]

\D 非数字等价与 [0-9]

\number 匹配相同 number 的组。组以 1 开始, 例如 (.+) \1 匹配'the the'or'55 55', 但是'thethe'(中间需要有空格), 这种特殊的序列仅仅被用来匹配 1 到 99 组。如果第一个数字为 0 或者是 3 为八进制的, 他将被解释为一个 group match, 在字符类'[' and ']' 中, 所有的数被当作字符。

A 匹配字符串的开始

\b 匹配空字符串, 但是仅仅是单词前面或者后面的空字符串, 单词被定义为一个 unicode 字母数字序列或下划线特征, 因此单词为被空格或者为字母数字预示, 非强跳得字符串, 注意, \b 被定义为 a\w 和 a\W 之间, 或者在\w 和单词开始之间, 这意味着 r'\bfoo\b' 匹配'foo','foo.','(foo)','bar foo baz' 而不是'foobar' 或者'foo3'

\B 匹配空字符串, 但是仅仅当它不在单词的开头或者结尾时, 这意味着 r'py\B' 匹配'python','py3','py2', 而不是'py','py.' 或者是'py!'.\B 和\b 相反, 因此单词时 unicode 字母数字或者下划线, 尽管这能被 ASCII flag 改变

- \s 对于 unicode 字符串类型: 匹配 unicode 空格字符串 (包括 [\t\n\r\f\v], 因此一些其它字符, 例如不间断的空格), 如果 ascii flag 被用, 仅仅

[\t\n\r\f\v] 被匹配 (但是 flag 影响整个正则表达我时)，因此在这样的情况下用 [\t\n\r\f\v] 也许是更好的选择。

\s 匹配不是任何不是空格的 unicode 字符，和\s 相反，如果 ascii flag 被用这因为等于 [^ \t\n\r\f\v](但是 flag 影响整个正则表达式，因此在这种情况下 [^ \t\n\r\f\v])

\z 匹配字符串的尾部

(?+...)	一个注释，括号里面的内容被简单的忽视	
(?=...)	如果... 匹配下一步，不小号任何字符串。例如 isaac (?=asimov) 将匹配'isacc' 如果它被'asimov' 跟着的话。	
(?!...)	如果... 不匹配下一个，例如 isaac (?!asimov) 将匹配'isaac'，仅仅是它没有'asimov' 跟着。	
\w	单词字符，等价与 [A-Za-z0-9_]	

正则表达式的语法实例

P(Y|YT|YTH|YTHO)?N	’PN’,’PYN’,’PYTN’,’PYTHN’,’PYTHON’
PYTHON+	’PYTHON’,’PYTHONN’,’PYTHONNN’,...
PY[TH]ON	’PYTON’,’PYHON’
PY[TH]?ON	’PYON’,’PYaON’,’PYbON’,’PYcON’,...
PY{:3}N	’PN’,’PYN’,’PYYN’,’PYYYN’,...

常用的正则表达式:

^[A-Za-Z]+$	26 个字母组成的字符串
^[A-Za-z0-9]+$	由 26 个字母和数字组成的字符串
^-?\d+$	整数形式的字符串
^[0-9]*[1-9][0-9]* $	正整数形式的字符串
[1-9]\d5	中国境内邮政编码，6 位
[\u4e00-\u9fa5]	匹配中文字符
\d{3}-\d{8}|\d{4}-\d{7}	国内电话号码，010-68913536

匹配 IP 地址的正则表达式：\d+.\d+.\d+ 或者\{1,3}. 精确写法:
0-99:[1-9]?\d
100-199:1\d{2}
200-249:2[0-4]?\d
250-255:25[0-5]
IP 地址的正则表达式:(([1-9]?\d|1\d{2}|2[0-4]\d|25[0-5]).){3}([1-9]?\d|1\d{2}|2[0-4]\d|25[0-5]

0.2 RE 库的主要功能函数

re.search()	在一个字符串搜索匹配正则表达式的第一个位置。
re.match()	从一个字符的开始为值起匹配正则表达式，返回 match 对象。
re.fullmatch()	如果整个字符串匹配正则表达式然会相应的 match 对象，不匹配返回 None，注意这不同于 0 长度匹配
re.findall()	搜索字符串，以列表类型返回全部匹配的字串
re.split()	将一个字符串按照正则表达式匹配结果进行分割，返回列表类型
re.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 match 对象
re.sub()	在字符串中替换所有匹配正则表达式的子串，返回替换后的字符串。
re.subn()	执行替换操作凡是返回一个 (new_string,number_of_subs_made) 元组
re.escape(pattern)	转义素有的字符除了 ASCII 字母，数字和下划线，如果你想匹配一个也许有正则表达式在里面的任一字符串这是很有用的。
re.purge()	清除正则表达式缓存

re.search(pattern,string,flags=0): 在一个字符串中搜索匹配正则表达式的第一个位置返回 match 对象。

- pattern: 正则表达式的字符串或原声字符串表示。
- string: 待匹配字符串。
- flags: 正则表达式使用时的控制标记。

0.2.1 re 表达式中的 flags

re.A	使\w \W\b\B\d\D \s\S 值执行 ASCII 匹配而不是 Unicode 匹配，仅仅对于 Unicode 样式有意义对 Byte 样式忽略。
re.DEBUG	显示编译表达式的调试信息
re.I	忽略正则表达式的大小写，[A-Z] 能够匹配小写。
re.L	使得\w \W\b\B\d\D \s\S 依赖于当前现场，当现场机制不可信时不鼓励使用，在不管什么时候它处理一个 cultrue，你应该用 Unicode 匹配，这个 flag 仅仅可以被用在 bytes 样式中。
re.M	正则表达式中的操` 作能够将给定字符串的每一行当作匹配开始

re.S 正则表达式中的. 操作能够匹配所有的字符，默认匹配除换行外的所有字符 re.VERBOSE(re.X) 这个 flag 通过允许你分割逻辑部分和增加注释允许你写的正则表达式更好，空 pattern 中的空格被忽略特别是当一个字符类

或者当有为转义的反斜线时，当一行包含不饿时字符类得 # 和非转义斜线时，所有的左边以 # 开头的字符将被忽略

```
a = re.compile(r"""\d␣+␣␣#␣the␣integral␣part
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\.␣␣␣␣#␣the␣decimal␣point
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\d␣*␣␣#␣some␣fractional␣digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

re.error(msg,pattern=None,pos=None)

- msg: 非正式格式的错误消息
- pattern: 正则表达式
- pos: 在 pattern 编译失败的索引 (也许是 None)
- lineno: 对应位置的行 (也许是 None)
- colno: 对应位置的列 (也许是 None)

```
import re
match = re.match(r'1\d{5}','BIT␣100081')
if match:
    match.group(0)
```

re.match(pattern,string,flags=0): 从一个字符串的开始位置起匹配正则表达式，返回 match 对象。

```
import re
match = re.match(r'1\d{5}','100081␣BIT')
if match:
    print(match.group(0))
```

re.findall(pattern,string,flags=0): 搜索字符串，以列表类型返回能匹配的子串。

```
import re
ls = re.findall(r'1\d{5}','BIT␣100081␣TSU100084')
```

re.split(pattern,string,maxsplit = 0,flags=0): 将字符串按照正则表达式匹配结果进行分割，返回列表类型。

maxsplit: 最大分割数，剩余部分作为最后一个元素输出。

```
import re
re.split(r'1\d{5}','BIT100081␣TSU100084')
re.split(r'1\d{5}','BIT100081␣TSU100084',maxsplit=1)
```

re.finditer(pattern,string,flags=0): 搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素时 matchdurian。

```
import re
for m in re.finditer(r'1\d{5}','BIT100081␣TSU100084'):
    if m:
        print(m.group(0))
```

re.sub(pattern,repl,string,count=0,flags=0) 在一个字符串中替换所有匹配正则表达式的子串返回替代厚的字符串。

- repl: 替换匹配字符串的字符串
- string: 待匹配字符串
- count: 匹配的最大替换次数

```
import re
re.sub(r'1\d{5}','110','BIT100081␣TSU100084')
```

Re 库的另一种等价用法:

```
rst = re.search(r'1\d{5}','BIT␣100081')
```

等价于

```
pat = re.compile(r'1\d{5}')
pat.search('BIT␣100081')
```

regex.search	在字符串中搜索匹配正则表达式的第一个位置，返回 match 对象
regex.match()	在字符串的开始为值起配置正则表达式，返回 match 对象
regex.findall()	所有字符串，以列表类型返回全部能匹配的子串
regex.split()	将字符串按照正则表达式匹配结果进行分割，返回列表类型。
regex.finditer()	搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素是 match 对象
reg.sub()	在一个字符串中替换所有匹配正则表达式的子串，返回替换手的字符串

Match 对象：一次匹配的结果，包含匹配的很多信息。

```
match = re.search(r'1'\d{5},'BIT 100081')
if match:
    print(match.group(0))
type(match)
```

match 对象的属性和方法

.string	待匹配的文本
.re	匹配时使用的 patter 对象 (正则表达式)
.pos	正则表达式搜索文本的开始位置
.endpos	正则表达式搜索文本的结束位置
.group(0)	获得匹配后的字符串
.start()	匹配字符串在原始字符串的开始位置
.end()	匹配字符串的结尾位置
.span()	返回 (.start(),.end())
.expand()	用 sub() 方法返回一个通过在 temple 字符串替代\的像\n 被转换成合适的字符串, 数值反向索引 (\1,\2) 和 (\g<1>,\g<name>) 被相应组里面的内容取代\字符串
.___getitem___(g)	允许你轻松的访问一个 match 组

.groupdict(default=None)	返回一个包含所有子组的匹配对象，key 是子组的名字，被用在 groups 的默认参数 默认参数不参加匹配，默认值时 None。
.lastindex	最新匹配的组的整数索引，或者如果没有组被匹配就为 None。例如表达式 (a)b,((a)(b)) 和 ((ab)) 将有 lastindex == 1 如果应用的字符串'ab', 然而表达式 (a)(b) 将有 lastindex == 2, 如果与应用在同一个字符串。
.lastgroup	最新匹配名字，如果 group 没有一个名字或者没有 group 就匹配为 None。
.re	正则表达式的 match() 或者 search() 方法生成的 match 实例

Re 库默认采用贪婪匹配，即输出匹配最长的字子串

```
match = re.search(r'PY.*N', 'PYANBNCNDN')
match.group(0)
```

通常搜索的时候 PYAN 就能匹配出结果但是根据贪婪匹配，匹配待匹配字符串中最长的字符串。输出最短子串 PYAN。

```
match = re.search(r'PY.*?N', 'PYANBNCNDN')
```

最小匹配操作符

操作符	说明
*?	前一个字符 0 次或者无限次扩展，最小匹配
+?	前一个字符 1 次或者浮现次扩展，最小匹配
??	前一个字符 0 次或者 1 次扩展，最小匹配
{m,n}?	扩展前一个字符串 m 到 n 次 (含 n)，最小匹配

```
import re
m = re.match(r'(\w+␣\w+)', 'Isaac␣Newton, physicist')
m.group(0)
m.group(1)
m.group(2)
m.group(1,2)
```

输出:

'Isaac Newton'
'Isaac'
'Newton'
('Isaac','Newton')

```
m =re.match(r'(\d+).(\d+)', '3.1415')
m.groups()
```

输出:

('3','1415')

```
m = re.match(r'(?P<first_name>\w+)␣(?P<last_name>\w+)', 'Malcolm␣Reynolds')
```

```
m.groupdict()
```

输出:'first_name':'Malcolm','last_name':'Reynolds'

操作符	说明	例子
.	任何单个字符	
[^]	非字符集字符，对单个字符给出排除范围	[^*abc*] 表示非 *a* 或者 *b* 或者 *c* 的单个字符
*	前一个字符 *0* 次或者无限次扩展	*abc** 表示 *ab,abc,abcc* 等
+	前一个字符 *1* 次或无限次扩展	*abc+* 表示 *abc,abcc,abccc* 等
?	前一个字符 0 次或者一次扩展	abc? 表示 ac,abc
|	左右表达式任一个	abc|def 表示 abc 或者 def
$\{m\}$	扩展前一个字符 m 次	$ab\{2\}c$ 表示 *abc,abbc*
$\{m,n\}$	扩展前一个字符 m 到 n 次，包含 n	$ab\{1,2\}c$ 表示 *abc,abbc*
^	匹配字符串开头	{^*abc*} 表示 *abc* 且在一个字符串开头
$	匹配字符串结尾	*abc$* 表示 *abc* 且在一个字符串的结尾
()	分组标记，内部只能使用 | 操作符	*(abc)* 表示 *abc*，*(abc|def)* 表示 *abc* 或者 *def*

(?imsx-imsx:...)	在字符字母集合'i','m','s','x' 中，'-' 跟着的来自同样字母集合的一个或者更多字母)，对于部分表达式字母集合或者移去相关的 flags:re.i,re.m,re.s,re.x。
<?p=name>	: 对于 group 的一个反向引用，它匹配之前 name 命名的 group 无论什么文本。