Python:re模块

- python中的re模块封装了正则表达式的全部功能，可以大幅简化在字符串中查询相应内容的时间。

常见符号和元字符：

- **普通字符：匹配自身**
  - abc匹配abc
- **‘.’： 匹配除了换行符’\n’外的所有字符（DOTALL中也能用于换行符）**
  - a.c匹配abc
- **\：转义字符**
  - a.c匹配a.c
- **‘*’：匹配前一个字符0或多次**
  - abc*匹配ab,abccc
- **+:匹配前一个字符1或多次**
  - abc+匹配abc，abccc
- **？：匹配前一个字符0或1次**
  - abc？匹配abc，ab
- **^：字符串开头，在多行模式下表示每行的开头**
- **$：字符串结尾，在多行模式下表示每行的结尾**
- **|：或，能够连接多个表达式，从左到右**
- **{m,n}：匹配前一个字符m到n次，若省略n则为m到无穷次**
  - ab{1,2}c匹配abc，abbc
- **[]：字符集，如abc可以表达成[abc]或[a-c]，[^abc]表示非abc的字符**
  - a[bcd]e匹配abe，ace，ade
- **()：分组**
  - (abc){2}a(123|456)c匹配abcabca456c

预定义字符集

- \d：数字：[0-9]
- \D：非数字:[^\d]
- \s：匹配任何空白字符:[<空格>\t\r\n\f\v]
- \S：非空白字符:[^\s]
- \w：匹配包括下划线在内的任何字字符:[A-Za-z0-9_]
- \W：匹配非字母字符，即匹配特殊字符
- \A：匹配字符串开头，同^
- \Z：匹配字符串结尾，同$
- \b：匹配单词边界，例如’er\b’ 可以匹配”never” 中的 ‘er’，但不能匹配 “verb” 中的 ‘er’。
- \B：[^\b]

分组的特殊用法

- 被()括起来的表达式将作为分组，从表达式左边开始没遇到一个分组的左括号“（”，编号+1.分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。
- (?P)：分组，除了原有的编号外再指定一个额外的别名，如：(?Pabc){2}
- (?P=name)：引用别名为的分组匹配到字符串，如：(?P\d)abc(?P=id)
- \：引用编号为的分组匹配到字符串，如：(\d)abc\1

re模块中的常用函数

compile()： 返回一个编译过的正则表达式模式

```
re.compile(pattern,flags=0)

pattern: 编译时用的表达式字符串。
```

flags 编译标志位，用于修改正则表达式的匹配方式，如：是否区分大小写，多行匹配等。

标志	含义
re.S(DOTALL)	使.匹配包括换行在内的所有字符
re.I（IGNORECASE）	使匹配对大小写不敏感
re.L（LOCALE）	做本地化识别（locale-aware)匹配，法语等
re.M(MULTILINE)	多行匹配，影响^和$
re.X(VERBOSE)	该标志通过给予更灵活的格式以便将正则表达式写得更易于理解
re.U	根据Unicode字符集解析字符，这个标志影响\w,\W,\b,\B

```
import re
tt = "Tina is a good girl, she is cool, clever, and so on..."
rr = re.compile(r'\w*oo\w*')
print(rr.findall(tt))   #查找所有包含'oo'的单词
执行结果如下：
['good', 'cool']
```

re.match(pattern, string[, flags])

这个方法将会从string（我们要匹配的字符串）的开头开始，尝试匹配pattern，一直向后匹配，如果遇到无法匹配的字符，立即返回None，如果匹配未结束已经到达string的末尾，也会返回None。两个结果均表示匹配失败，否则匹配pattern成功，同时匹配终止，不再对string向后匹配。

```
# -*- coding: utf-8 -*-

#导入re模块
import re

# 将正则表达式编译成Pattern对象，注意hello前面的r的意思是“原生字符串”
pattern = re.compile(r'hello')

# 使用re.match匹配文本，获得匹配结果，无法匹配时将返回None
result1 = re.match(pattern,'hello')
result2 = re.match(pattern,'helloo CQC!')
result3 = re.match(pattern,'helo CQC!')
result4 = re.match(pattern,'hello CQC!')

#如果1匹配成功
if result1:
  # 使用Match获得分组信息
  print result1.group()
else:
  print '1匹配失败！'

#如果2匹配成功
if result2:
  # 使用Match获得分组信息
  print result2.group()
else:
  print '2匹配失败！'

#如果3匹配成功
if result3:
  # 使用Match获得分组信息
  print result3.group()
else:
  print '3匹配失败！'
```

```
#如果4匹配成功
if result4:
  # 使用Match获得分组信息
  print result4.group()
else:
  print '4匹配失败！'
```

运行结果

```
hello
hello
3匹配失败!
hello
```

属性和方法

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、\g引用分组，但不能使用编号0。\id与\g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\g0。

案例：

```
# -*- coding: utf-8 -*-
#一个简单的match实例

import re
# 匹配如下内容：单词+空格+单词+任意字符
m = re.match(r'(\w+) (\w+)(?P.*)', '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():", m.group()
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'\g \g\g'):", m.expand(r'\2 \1\3')

### output ###
# m.string: hello world!
# m.re:
# 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!
```

re.search(pattern, string[, flags])

search方法与match方法极其类似，区别在于match()函数只检测re是不是在string的开始位置匹配，search()会扫描整个string查找匹配，match（）只有在0位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回None。同样，search方法的返回对象同样match()返回对象的方法和属性。

```
#导入re模块
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 = re.search(pattern,'hello world!')
if match:
  # 使用Match获得分组信息
  print match.group()
### 输出 ###
# world
```

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。

```
import re

pattern = re.compile(r'\d+')
print re.split(pattern,'one1two2three3four4')

### 输出 ###
# ['one', 'two', 'three', 'four', '']
```

re.findall(pattern, string[, flags])

搜索string，以列表形式返回全部能匹配的子串。

```
import re

pattern = re.compile(r'\d+')
print re.findall(pattern,'one1two2three3four4')

### 输出 ###
# ['1', '2', '3', '4']
```

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。

```
import re

pattern = re.compile(r'\d+')
for m in re.finditer(pattern,'one1two2three3four4'):
  print m.group(),

### 输出 ###
# 1 2 3 4
```

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、\g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。

```
import re

pattern = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'

print re.sub(pattern,r'\2 \1', s)

def func(m):
  return m.group(1).title() + ' ' + m.group(2).title()

print re.sub(pattern,func, s)

### output ###
# say i, world hello!
# I Say, Hello World!
```

re.subn(pattern, repl, string[, count])

返回 (sub(repl, string[, count]), 替换次数)。

```
import re

pattern = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'

print re.subn(pattern,r'\2 \1', s)

def func(m):
  return m.group(1).title() + ' ' + m.group(2).title()

print re.subn(pattern,func, s)

### output ###
# ('say i, world hello!', 2)
# ('I Say, Hello World!', 2)
```