石博文
shibw@tedu.cn

正则表达式 （regex）
动机 ： 1. 文本处理成为计算机的常见工作
2. 文本处理中，根据内容筛选查找匹配指定的内容又是常用功能之一
3. 方便快速的解决上述问题，正则表达式应运而生
定义 ： 使用字符和特殊符号组成的能够描述某一类字符串的表达式。
特点和具体使用
* 是一种高级的文本搜索匹配模式，提供了丰富的功能
* 目前常见程序语言（php,python,java,c#)都支持正则表达式。
* 高级程序语言的正则表达式语法几乎一致
* 在爬虫中大量使用正则表达式进行 html 文本的匹配

正则表达式规则及元字符
元字符 ： 正则表达式中定义的，具有特定意义的符号

import re
re 模块是 python 的标准库模块，是用来处理正则表达式的

re.findall(regex,string)
功能：使用 regex 去匹配 string 中的内容，如果匹配到则以一个列表的方式进行返回
findall(string)
功能 ： 获取字符串中能被正则表达式匹配的内容
参数 ： 目标字符串
返回值 ： 将匹配到的内容以列表形式返回
当正则表达式有分组的时候，则每一项会显示每个分组匹配到的内容

模块接口
compile(regex)
功能: 生成一个正则表达式对象
参数：传入一个正则表达式
返回值 ： 返回正则表达式响应的对象

* 正则表达式对象的一些属性函数 同 re 模块可调用的一些函数函数名功能相同，用法相近
* 这些函数多为常用的匹配显示函数
* 功能相同，使用的区别上只是 用 re 直接调用的时候，第一个参数需要传入正则表达式，使用 compile 对象调用的时候则不用

split()
功能 ： 将一个字符串，按照正则表达式进行分割
参数 ： 要分割的字符串

返回值 ： 分割后的内容放入一个列表返回

sub(repl,string,count = 0)
功能 ： 使用 replacestr 的内容 ， 替换 string 字符串中能被正则表达式匹配的部分
参数 ： repl ： 替换内容
string ： 要匹配的字符串
count ： 默认情况下替换所有匹配到的部分
如果赋值，表示最多替换 max 处
返回值 ： 返回替换后的字符串

match(string)
功能 ： 匹配一个字符串，得到匹配结果
参数 ： 要匹配的字符串
返回值 ： 匹配到返回一个 match object，没有匹配到返回 None
* 只有当正则表达式匹配的内容为字符串的开头的时候才能匹配到 ， 并且，当有多出的时候只能匹配一处

search()
同 match
区别：匹配第一次出现的

finditer()
同 findall
区别 ： 返回一个迭代对象，每个迭代对象都是一个 match object

显示匹配结果：
group([n])
功能 ： 显示匹配到的字符串
参数 ： 如果不加默认为 0 表示返回整体的匹配结果
如果加一个数字，表示返回对应的组的匹配结果 如果越界会报错
返回值 ： 返回响应的匹配结果

groups()
功能 ： 得到所有组匹配到的内容，以一个元组返回

********** 普通字符

元字符 ： abc
匹配规则 ： 匹配 字符串 的内容
e.g. ：
In [2]: re.findall('abc','abcdefabcdefgh')
Out[2]: ['abc', 'abc']

********** 匹配任意一个字符

元字符：　‘.’
匹配规则　使用“.”代表任意一个字符 不能代表‘\n’
　　f.o　---->　foo　fao　f@o

```
In [9]: re.findall('f.o','from china')
Out[9]: ['fro']
```

********** 匹配字符集合
元字符 ： [adfbd] [a-z] [A-Z] [0-9] [a-zA-Z0-9]
匹配规则 ： 匹配集合中任意一个字符
　　[abcde] ----> a　b　c
　　[a-zA-Z] ----> 所有字母　a　A　B
　　[abc0-9]
e.g.:

```
In [48]: re.findall('[0-9][abcd][A-Z]','1bDdg3cK')
Out[48]: ['1bD', '3cK']
```

********** 字符集合取非
元字符　[^ ...]
匹配规则 ： 匹配任意一个非集合中的字符

[^abcd]----> f l
[^_a-zA-Z0-9] --->　$ ^ %　* (&

```
In [55]: re.findall('[^aeiou]','hello world')
Out[55]: ['h', 'l', 'l', ' ', 'w', 'r', 'l', 'd']
```

```
In [56]: re.findall('[^0-9]','hello 12306')
Out[56]: ['h', 'e', 'l', 'l', 'o', ' ']
```

********** 任意 数字/非数字 字符
元字符 ： \d　\D
匹配规则 ： \d 匹配任意一个数字字符　[0-9]
　　\D 匹配任意一个非数字字符 [^0-9]
\d{3} ----> 123　435　546
\D　-----> a $　d *
e.g.:

```
In [58]: re.findall('\d{3}','hello 12306')
Out[58]: ['123']
```

```
In [59]: re.findall('\D{3}','hello 12306')
Out[59]: ['hel', 'lo ']
```

********** 匹配任意（非）数字字母下划线
元字符 ： \w　\W
匹配规则 ： \w　匹配任意数字字母下划线[_a-zA-Z0-9]
　　\W　与\w 相反 [^_a-zA-Z0-9]

e.g. :
In [60]: re.findall('[A-Z]\w*','Hello World')
Out[60]: ['Hello', 'World']

In [61]: re.findall('\w*-\d+','xiaoming-64')
Out[61]: ['xiaoming-64']

**********匹配 空白 / 非空白字符
元字符 ： \s \S
匹配规则 ： 空白字符 '' \n \r \0 \t
\s 匹配其中任意一个
\S 匹配任意一个非空白字符

In [69]: re.findall('hello\s+world','hello world')
Out[69]: ['hello world']

In [70]: re.findall('\S*','hello world')
Out[70]: ['hello', '', 'world', '']

********** 或连接多个正则表达式
元字符：|
匹配规则： abc|def 表示既能匹配 abc 也能匹配 def
e.g.:
In [5]: re.findall('abc|def','abcdkjgsdefasd')
Out[5]: ['abc', 'def']
* 竖线左右不要加空格，除非是要匹配空格
* 不能查找重叠

********** 匹配 0 次或多次正则表达式
元字符 ： ‘*’
匹配规则 ： 用 * 匹配它前面出现的正则表达式 0 次或多次 ab* ----》a ab abbb abbbbbbb
a.* ----> a ab ac adddsfdgdfg

e.g.
In [22]: re.findall('ab*','acdefghig')
Out[22]: ['a']

In [23]: re.findall('a.*','acdefghig')
Out[23]: ['acdefghig']

********** 匹配前面的正则表达式 1 次或多次

元字符 ： ‘+’
匹配规则 ：匹配前面出现的正则表达式至少出现 1 次

```
In [27]: re.findall('ab+','abbbbbbb')
Out[27]: ['abbbbbbb']

In [28]: re.findall('ab+','acadf')
Out[28]: []
```

********** 匹配前面出现的正则表达式 0 次或 1 次

元字符 ： ‘?’
匹配规则 ： 匹配前面的正则表达式，0 次，1 次

```
In [32]: re.findall('a.?','absdfsaa')
Out[32]: ['ab', 'aa']
```

********** 匹配前面的正则表达式 指定的次数

元字符 ： {N} N 表示一个正整数
匹配规则： 匹配前面的正则表达式出现 N 次
e.g.:

```
In [34]: re.findall('a.{3}','absdfsaa')
Out[34]: ['absd']
```

********** 匹配前面出现的正则表达式指定次数区间

元字符 ： {m,n}
匹配规则 ： 匹配前面的正则表达式 m---n 次
e.g.

```
In [36]: re.findall('ab{3,6}','abbbbbbbb')
Out[36]: ['abbbbbb']
```

********** 匹配字符串开头
元字符 ： ^
匹配规则 ： 匹配字符串的开头位置内容
^abc ----> 以 abc 开头的字符串
e.g.

```
In [12]: re.findall('^From','From China')
Out[12]: ['From']

In [13]: re.findall('^from','I am from China')
Out[13]: []
```

********** 匹配字符串结尾

元字符 ： $

匹配规则 ： 当一个正则表达式是一个字符串结尾时能匹配出来 \.py$ ----> test.py find.py

e.g.

```
In [20]: re.findall('py$','find.py')
Out[20]: ['py']
```

********** 匹配单词边界

元字符 ： \b \B

匹配规则 ：将连续字母认为是一个单词
而字母与非字母的交接认为是单词边界

```
In [85]: re.findall(r'\bto\b','Welcome to tornado')
Out[85]: ['to']
```

总结 ：

字符 ： 匹配实际字符

匹配单个字符： . [] \w \W \d \D \s \S

匹配正则表达式重复次数 * + ? { }

匹配开头结尾或边界 ^ $ \b \B

其他 | [^..]

r 的问题

raw（原始字符串格式）字符串特点 ： 不会进行转义，将字符串内所有的内容原样使用正则表达式转义

当正则表达式中要匹配 \ * . ? {} [] () " ' 这些字符时需要使用'\'进行转义。此时如果为了避免字符串 解析为 正则表达式 带来的麻烦，最好使用 raw 字符串

e.g.

```
In [102]: re.findall('\\\"hello\\\"','he said "hello"')
Out[102]: ['"hello"']
In [103]: re.findall(r'\"hello\"','he said "hello"')
Out[103]: ['"hello"']
```

贪婪 & 非贪婪

贪婪模式 ： 在默认情况下，正则表达式是贪婪模式，即使用 * + ? {m,n} 的时候，都尽可能多的向后匹配内容

非贪婪模式 ： 尽量少的匹配内容

贪婪 ----》 非贪婪 后面加 ?

```
*    ---》  *?
+            +?
?            ? ?
{m,n}      {m,n} ?
```

正则表达式的分组

* 正则表达式可以分组，分组的标志即为()，每个括号中的内容，是整体正则表达式的一个子组，表达了一个小的整体

* 子组可以影响 * + ？ {} 的重复行为，当重复是把子组当作整体进行对待

* 当有多个子组的时候，从外到内，从左到右，称为第一子组，第二子组 第三子组。。。。。。

练习 ：

1.匹配长度为 8-10 位的密码必须以字母开头，密码可以是数字字母下划线组成

^[a-zA-Z]\w{7,9}$

2.匹配身份证号

\d{17}(\d|x)

3. 浮点数

^-?\d+\.\d+$

#collections 是 Python 内建的一个集合模块，提供了许多有用的集合类。

import collections

#Counter 类是一个简单的计数器，目的是用来跟踪值出现的次数。它是一个无序的容器类型，以字典的键值对形式存储，其中元素作为 key，其计数作为 value。计数值可以是任意的整数（包括 0 和负数）

```
>>> from collections import Counter
>>> c = Counter("abcdefgab")
>>> c["a"]
2
>>> c["c"]
1
>>> c["h"]
0
```

update 可以使用一个 iterable 对象或者另一个 Counter 对象来更新键值。

不同于字典的 update 方法，这里更新 counter 时，相同的 key 的 value 值相加而不是覆盖

```
>>> c = Counter('which')
>>> c.update('witch') #  使用另一个 iterable 对象更新
>>> c['h']
3
>>> d = Counter('watch')
>>> c.update(d) #  使用另一个 Counter 对象更新
>>> c['h']
4
```

练习

```
import re
#collections 是 Python 内建的一个集合模块，提供了许多有用的集合类。
import collections
def count_patt(fname, patt):
```

```
    #Counter 是一个简单的计数器
    counter = collections.Counter()
    cpatt = re.compile(patt)

    with open(fname) as fobj:
        for line in fobj:
            m = cpatt.search(line)
            if m:
                counter.update([m.group()])

    return counter

if __name__ == '__main__':
    fname = 'access_log.txt'
    ip_patt = '^(\d+\.){3}\d+'
    a = count_patt(fname, ip_patt)
    print(a)



import re
import collections

class CounitPatt(object):
    def __init__(self, patt):
        self.cpatt = re.compile(patt)

    def count_patt(self, fname):
        counter = collections.Counter()

        with open(fname) as fobj:
            for line in fobj:
                m = self.cpatt.search(line)
                if m:
                    counter.update([m.group()])
        return counter

if __name__ == '__main__':
    fname = 'access_log.txt'
    ip_patt = '^(\d+\.){3}\d+'
    br_patt = 'Firefox|Chrome|MSIE'
    ip = CountPatt(ip_patt)
    print(ip.count_patt(fname))
    br = CountPatt(br_patt)
    print(br.count_patt(fname))
```