python的re模块--细说正则表达式

翻译自官方文档。

**前言**

什么是正则表达式,regular expression在英语中是有规则的表达式,也就是说,该表达式只是一条的规则,而正则表达式引擎能够根据这条规则,帮你在字符串中寻找所有符合规则的部分,比如,我有一条字符串"hello world 123"  
而规则,可以很具体,比如hello 那么引擎会帮你把hello从字符串中找出来,规则也可以比较抽象,比如\d这个表示数字,也就是引擎会帮你把字符串中的数字寻找出来,根据正则表达式的理论,我们可以把规则串联起来,变成一条更复杂的规则。想要从一条字符串中找到对你来说有意义的部分,你的任务可能不仅仅是从字符串中提取数字这么简单,因此需要设计非常复杂的规则,而本教程则会告诉你,如何去制定非常那些非常复杂的规则。事实上,制定好规则之后,引擎不仅能够根据规则进行查找,还可以进行分割,替换等复杂的操作,能让你随心所欲得处理字符串。本教程默认你懂得一些python的简单语句尤其是对str处理时的语句。

python中的re模块提供无比强大的正则表达式功能。如果需要更加逆天的正则表达式功能,第三方regex模块具有与标准库re模块兼容的API,但提供了额外的功能和更全面的Unicode支持。不过python自带的re模块已经足够牛逼了,只要你能够掌握下面的内容。

**学习正则表达式的意义**

为什么要学习正则表达式？不仅python,绝大多数语言的正则表达式的设计理念都是差不多的,你在这里学熟练了,那么对于其他语言的正则表达式也会融会贯通。学会正则表达式,不仅能提升编程能力,还能让你的工作和生活更加方便。比如,有人在处理文章的时候,想把文章的数字标号"第1章"变成汉字数字编号"第一章",如果章节非常多,手动修改是非常痛苦的,而word自带的替换功能已经无能为力了,那么可以考虑使用一个功能强大的编辑器比如sublime然后通过正则表达式进行替换。类似的文本处理在工作和生活中经常出现,掌握正则表达式绝对会让你能够更加轻松应对这些场景。此外,学会正则表达式能够做的东西非常多,也非常酷,比如本教程最后讲的分词器,就是编写编译器的关键步骤,没错,学完正则表达式,是开发一个属于你自己的语言的关键步骤,说不定你的语言能够和python一样广受欢迎。

**关于\**

正则表达式使用反斜线字符 \ 作为Escape符,让特殊字符失去含义。这与Python在字符串的使用相冲突,所以,如果我们需要匹配反斜杠,我们需要输入的正则表达式应该是\\也就是我们要输入两个反斜杠,这样一来,要告诉python的字符串,这里有两个反斜杠,那么,我们的python字符串里面就需要有四个反斜杠\\\\。

！！！如果你觉得很难理解,没关系,只要记得下面的解决方案就可以了。

解决方案是将Python的原始字符串表示法用于正则表达式模式;在以 'r' 为前缀的字符串文字中不以任何特殊方式处理反斜杠。所以 r"\n" 是包含 \ 和 'n' 的两个字符的字符串,而 "\n" 是包含换行符的单字符字符串。re读取了 \ 和 n 连续两个字符,他就知道,这种表达式是要匹配一个换行符。总而言之,使用的时候一定要使用r'str'这样的模式,你在查找资料的时候看到也经常是这种情况。

！！！以上内容看不懂没关系,只要记住,python的正则表达式都是 r"str" 这样的形式就可以了！

**正则表达式规则**

规则是相通的，除了python，其他语言的正则表达式规则基本上都是一样的，只是语言的实现不一样，因此，我们先看看什么是正则表达式规则。

正则表达式（RE）实际上就是一条规则,能让字符串的某些部分匹配上,或者说该部分符合这条规则。有时候规则比较具体,如找出字符串中所有的hello,那么就字符串中一定要是要有'hello'才能匹配,其他的'Hello',或者'HELLO',再或者有错别字的'gello'都是不符规则的。有时候呢,可以放宽一些规则,比如允许第一个字母是错别字,只要是小写字母就好了,那么就可以这样写规则[a-z]ello,其中 [a-z]表示a到z中的任意一个。我们可以看到,在规则[a-z]ello中既有ello这样的普通字符,又有[a-z]这样的特殊规则,组成了一条完整的规则,我们也能看出,只有普通字符那么规则是非常具体严格的,加上[a-z]这样的特殊符号之后规则变得更加灵活。所以,特殊符号才是正则表达式强大所在,我们可以使用不同的特殊符号进而构造更加复杂的规则。  
下面就看看python的正则表达式有哪些特殊符号。（其他的语言中的特殊符号也都大同小异）

值得说明一下的是,下面的规则有些非常复杂,很多人可能这辈子都用不上,所以,看到你觉得复杂或者看不懂的规则,你可以跳过去,没有任何影响,或许你在看完全部的内容之后再回过来看这些特殊的符号,特殊的规则,会有新的领悟,所以,学习正则表达式不可能一下就能学会的,一定要多看几遍,多练习,才能融会贯通,因为学习最主要的还是要运用。

特殊符号（又称特殊字符）如下:

.

在默认模式下,它匹配除换行符以外的任何字符。（如果已经指定了DOTALL标志,则它匹配包括换行符的任何字符。至于如何设定这个,我们之后再讲）

因此我们的设定规则 .ello 的时候,符合这条规则的字符串有'hello','8ello','kello'...非常多,不管开头是什么字符,只要后面是'ello'就符合规则。

^

匹配字符串的开头,并且在MULTILINE模式下,也会在每个换行符后立即匹配。

我们设定规则^hello 那么字符串 'hello world'里面的'hello'能够符合规则,而'world hello'里面的'hello'不符合规则,因为它不在字符串的开头。  
由于在Multiline模式下,也匹配每个换行符后,所以,在开启了multiline模式下,  
'world\nhello'中的'hello'也符合规则。

$

匹配字符串的结尾或紧挨在字符串末尾的换行符之前,并且在MULTILINE模式下也匹配换行符之前的匹配。

foo 匹配 'foo' 和 'foobar' ,而正则表达式 foo$ 只匹配 'foo'或者'hello foo'。 更有趣的是,在 "foo1\nfoo2\n" 中搜索 foo.$ 通常与 'foo2' 匹配,而在MULTILINE模式下得到的是'foo1'。在'foo\n'中搜索单个 $ 将会找到两个(空)匹配:一个位于换行符之前,另一个位于字符串末尾。

\*

对\*前面的RE匹配0次或多次,并尽可能多的重复。

我们设定规则 ab\* ,意思是对b可以重复匹配。 将匹配'a','ab'或'a'后跟任意数量的'b'。而且\*默认的是贪婪模式,也就是说对于字符串'abbb'规则认为整个字符串符合规则,而不是子字符串符合规则。将来我们要求返回符合规则的部分时,将会返回整个字符串abbb而不仅仅是ab或者a。

+

对于'+'前面的RE进行一个或多个重复。

ab+ 将与 'ab' 匹配,后面还可以跟任意个'b' ,但这条规则不会匹配 'a' 。值得注意的是,这条规则也是贪婪的。

?

对? 前面的RE进行0或1个重复。

ab? 将匹配 'a' 或 'ab'。这条规则也是贪婪的。

\*?, +?, ??

\*, + 和 ? 限定符都是贪婪的; 它们匹配尽可能多的文本。 有时候这种行为是不希望的; 如果RE <.\*>与 '<a>b<c>' 匹配,它将匹配整个字符串,而不仅仅是 '<a>' 。 添加? 在限定符之后以非贪婪或最小方式进行匹配; 尽可能少的字符将被匹配。 使用RE <.\*?>将只匹配 '<a>'。

值得一提的是,我们看到 a\*? 我们应当想到这是一条关于\*的规则,而不是关于?的规则,后者只是为前者服务的。

{m}

指定应该匹配{m}前一个RE的正好m个副本; 更少的匹配导致整个RE不匹配。 例如, a{6} 将完全匹配六个 'a' 字符,但不是五个。  
如果多于6次,那么多出的部分被忽略。

{m,n}

使得到的RE匹配前面RE的m到n次重复,试图尽可能多地匹配重复。 例如, a{3,5} 将匹配3到5个 'a' 字符。省略m则指定零作为下限,省略n指定无限上限。 例如, a{4,}b 会匹配 'aaaab' 或一千个 'a' 字符,后跟一个 'b',但不匹配 'aaab'。 逗号不能省略,否则修饰符会与之前描述的表单混淆。

这条规则也是贪婪的,它会尽可能到达重复n次

{m,n}?

使得到的RE从m到n重复前面的RE,尝试匹配尽可能少的重复。 这是以前限定符的非贪婪版本。 例如,在6个字符的字符串'aaaaaa'上, a{3,5}将匹配5个 'a'字符,而 a{3,5}? 只会匹配3个字符 'a'。

\

要么特殊字符失去意义,要么标志一个特殊序列; 之后会讨论特殊序列。

首先,你在python中写规则的时候一定要用raw字符串,也就是r'...',否则就会被''的特殊性给绕晕。其次要记住,这个符号叫Escape符号,使得特殊符号逃离特殊功能,比如,我们要匹配一个'.',那么我们写规则的时候就需要这样写 r'\.'

[]

用于指示一组(或者说一个集合)字符。在一组中:

* 字符可以单独列出,例如 [amk] 将匹配 'a','m'或'k'。
* 字符的范围可以通过给出两个字符并用'-'分隔来指示,例如 [a-z]将匹配任何小写ASCII字母, [0-5][0-9] 将匹配所有的两位数字00到59, [0-9A-Fa-f] 将匹配任何十六进制数字。如果 - 被转义(例如[a\-z]),或者被放置为第一个或最后一个字符(例如 [-a] 或 [a-] ),它将匹配一个'-'。
* 特殊字符在集合内部失去其特殊含义。例如, [(+\*)]将匹配任何文字字符 '(' , '+' , '\*' 或 ')' 。
* 字符类如 \w 或 \S 也可以在一个集合内接受,尽管它们匹配的字符取决于ASCII或LOCALE模式是否有效。(之后会有关于字符类更加详细的说明)
* 不在一个范围内的字符可以通过对该集合进行补充来匹配。如果[]中的第一个字符是 '^' ,则不匹配的所有字符将被匹配。例如,[^5]将匹配除'5'以外的任何字符,并且[^^]将匹配除 '^' 以外的任何字符。 '^' 如果它不是集合中的第一个字符,它没有特别的意义。
* 在一个集合内匹配']',在它之前加一个反斜杠,或者将它放在集合的开头。例如, [()[]{}]和[{}]都将与括号匹配。

|

A|B,其中A和B可以是任意RE,创建一个匹配A或B的正则表达式。任意数量的RE可以用'|'分隔。通过这种方式,这可以在组内使用(见下文)。当目标字符串被扫描时,由'|'分隔的RE 从左到右尝试。当一个模式完全匹配时,该分支被接受。 这意味着一旦A匹配,B将不会被进一步测试,即使它会产生更长的整体匹配。 换句话说, '|'操作从不贪婪。要匹配'|',请使用 \| ,或将其放在字符类中,如[|]中所示

\d

对于Unicode(str)模式:

* 匹配任何Unicode十进制数字。这包括[0-9]以及许多其他数字字符。 如果使用ASCII标志,则只匹配[0-9]。

对于8位(字节)模式:

* 匹配任何十进制数字; 这相当于[0-9]。

\D

匹配任何不是十进制数字的字符。 这与\d相反。 如果使用ASCII标志,则相当于[^ 0-9]。

\s

对于Unicode(str)模式:

* 匹配Unicode空白字符(包括[\t \n \r \f \v]以及许多其他字符,例如许多语言中的排版规则强制的非空白空格)。 如果使用ASCII标志,则只匹配[\t \n \r \f \v]。

对于8位(字节)模式:

* 在ASCII字符集中匹配被认为是空白的字符; 这相当于[\t \n \r \f \v]。

\S

匹配任何不是空白字符的字符。 这与\s相反。 如果使用ASCII标志,则这等价于[^ \t \n \r \f \v]。

\w

对于Unicode(str)模式:

* 匹配Unicode字符; 这包括大多数可以是任何语言的单词的一部分的字符,以及数字和下划线。 如果使用ASCII标志,则只匹配[a-zA-Z0-9\_]。

对于8位(字节)模式:

* 匹配ASCII字符集中被认为是字母数字的字符; 这相当于[a-zA-Z0-9\_]。 如果使用LOCALE标志,则匹配在当前语言环境和下划线中被认为是字母数字的字符。

\W

匹配任何不是单词字符的字符。 这与\ w相反。 如果使用ASCII标志,则变成等效于[^ a-zA-Z0-9\_]。 如果使用LOCALE标志,则匹配在当前语言环境和下划线中被认为是字母数字的字符。

\Z

只匹配字符串的末尾。只能放在正则表达式末尾

**进阶规则**

(...)

匹配括号内的任何正则表达式,并指示组的开始和结束; 在匹配完成后可以检索组的内容,并且可以在后面的字符串中使用\number特殊序列进行匹配,之后有详细描述。 要匹配'('或')',请使用 \( 或 \) ,或将它们放在字符类中: [(],[]] 。 举个例子我有个字符串 'asdfa020-12345678bsaefga' 可以看到我们的电话号码被一群字母围住了,怎么把电话号码拿出来,顺便把区号和真正的号码都分开呢？ 可以用这样的规则(\d{4})-(\d)\* 这样符合结果字符串就是 '020-12345678' 但是,返回的结果对象并不仅仅是一个字符串,他还有.group,其中 group(1) = '020' group(2)='12345678'

\number

匹配相同编号的组的内容。 例如 (.+) \1 这个式子等价于 (.+) (.+) 和'the the' 或 '55 55'匹配,但不匹配'thethe'(注意组之后的空格)。 该特殊序列只能用于匹配前99个组中的一个。 如果数字的第一个数字是0或数字是3个八进制数字长度,则不会将其解释为组匹配,而是将其解释为具有八进制数值的字符（这句话初学者可以略过）。

这个规则非常棒,能帮你省不少事情,比如一个字符串里面有两个邮件地址,你想把它们都找出来,你写好了一个检测邮件地址规则,剩下那个你不想写了,你可以引用一下就可以了。

温馨提示,下面的特殊符号可能比较难,需要耐心去体会,或者干脆略过。

(?P<name>...)

与常规圆括号类似,但可以通过符号组名称来访问与该组匹配的子字符串。 组名称必须是有效的Python标识符,并且每个组名称只能在正则表达式中定义一次。

相当于给我们之前提到的组(...)添加了一个名字而已,添加名字的好处是可以根据名字来引用这个组,而不是靠编号。  
相当于给我们之前提到的组(...)添加了一个名字而已,添加名字的好处是可以根据名字来引用这个组,而不是靠编号。举个之前的例子,有字符串 'asdfa020-12345678bsaefga' ,我想取出号码,可以这样写  
(?P<quhao>\d{4})-(?P<haoma>\d\*)  
这样返回的结果字符串就是020-12345678,但是返回的对象还有group属性,而且 group['quhao'] = 020 group['haoma'] = 12345678

命名组可以在三种情况下被引用。 比如我们想在句子中找到引号,这样我们就能找到文章中引用的内容,所以,我们可以这样写 (P<quote>['"]) 这样就能找到一个单引号或者双引号,但是一般来说,我们需要找到一对单引号或者双引号,这时我们不必在写一个单引号或双引号的正则表达式,只要引用一下之前的就好了,如(?P<quote>['"]).\*?(?P=quote) 后面一个实际上就是对于前面的引用,这时再正则表达式中引用的情况,也就是下表中第一种情况。实际上不仅规则中很有可能会引用前面写好的组,还有其他的两种情况需要引用,请见表

| **引用'quote'** | **引用的方法** |
| --- | --- |
| 在正则表达式中 | (?P=quote)  \1 |
| 在匹配的结果中 | m.group('quote')  m.end('queto') |
| 在re.sub中的repl里 | \g<quote>  \g<1>  \1 |

(?P=name)

对指定组的反斜线引用; 它匹配与早先的组命名相匹配的任何文本。见上一条的说明

(?...)

这是一个扩展符号。'?'后面的第一个字符决定了结构的含义和进一步语法。扩展通常不会创建一个新的组。(?P<name> ...)是这个规则的唯一例外,以下是当前支持的扩展。

(?aiLmsux)

(?号后面跟着'a','i','L','m','s','u','x'的一个或多个字母)。这些字母设置相应的标志:re.A(仅ASCII匹配),re.I(忽略大小写),re.L(依赖于语言环境),re.M(多行),re.S(点匹配全部) ,re.U(Unicode匹配)和re.X(详细),用于整个正则表达式。这些标志的作用在模块内容中有具体的描述。事实上,这些标志可以通过flag参数传给正则表达式,也可以像这样直接写进正则表达式里面。

(?:...)

常规圆括号的非捕获版本。 匹配括号内的任何正则表达式,但匹配的子字符串在执行匹配或稍后引用模式后无法检索。

(?aiLmsux-imsx:...)

(来自'a','i','L','m','s','u','x'的零个或多个字母,可选地后面跟着' - ',后面跟着一个或多个来自'','m','s','x')。字母设置或删除相应的标志:re.A(仅ASCII匹配),re.I(忽略大小写),re.L(依赖于语言环境),re.M(多行),re.S(点全部匹配),re.U(Unicode匹配)和re.X(冗长),用于表达部分。 注意,这里上面提到的(?aiLmsux)的区别是,这里有一个冒号,后面跟正则表达式,因此这里设置的标志仅适用于窄内联组,并且原始匹配模式在组外部恢复。在设置标志的时候(?aiLmsux-imsx:...)等价于(?aLu:...),也就是没必要刻意先在左边写进去后边再跟随'-'号删掉,但是,有时候'-'也是很有用的,比如之前设置了(?i)说明对所有的正则表达式都忽略大小写,但是你对某一个部分需要强调大小写,比如,你想要找superMAN对前面的super的大小写无所谓,但要求MAN一定是大写,可以这样写 (?i)super(?-i:MAN)这样 sUpeRMAN能匹配,而superMan则不能匹配

(?#...)

一条评论; 圆括号的内容被简单地忽略。

(?=...)

如果...匹配next,但不消耗任何字符串。 这被称为前瞻断言。 例如, Isaac(?=Asimov) 只有跟随着'Asimov'才会匹配'Isaac'。但返回的结果依然还是Isaac,所谓不消耗字符串,意思是,后面的Asimov依然可以被继续匹配,如  
Isaac(?=[A])AsimovAlab 匹配 'IsaacAsimovAlab'  
而 Isaac(?=[Asimov])G 则会无法匹配任何字符串,因为根据(?=...)要求,Issac后面必须是Asimov,而它后面又要跟着一个G,所以,互相矛盾。

(?!...)

如果...不匹配。 这是一个负面的前瞻断言。 例如,Isaac(?！Asimov)只有在没有跟随'Asimov'时才会匹配'Isaac'。

(?<=...)

匹配如果字符串中的当前位置在...之前匹配...,并以当前位置结束。 这被称后行断言。 (?<= abc)def 会在'abcdef' 中找到一个匹配项,因为lookbehind会回看3个字符并检查包含的模式是否匹配。 包含的模式只能匹配一些固定长度的字符串,这意味着 abc 或 a|b 是允许的,但a\*和a{3,4}不是。 请注意,像这种向前看的搜索字符串模式在匹配开头的时候会出现问题;因此 必须使用search()函数而不是match()函数:仔细想想为什么？

>>> import re

>>> m = re.search('(?<=abc)def', 'abcdef')

>>> m.group(0)

'def'

本示例在连字符后面查找单词:

>>> m = re.search(r'(<=-)\w+', 'spam-egg')

>>> m.group(0)

'egg'

(?<!...)

上一例子的不匹配情况

(?(id/name)yes-pattern|no-pattern)

如果存在给定id或名称的组,则尝试与yes-模式匹配,如果不存在,则使用无模式。 无图案是可选的,可以省略。 例如,正则表达式(<)?(\w+@\w+(?:\.\w +)+)(?(1)>|$)是一个电子邮件匹配模式,它将匹配'<[user@host.com](mailto:user@host.com) >'以及'[user@host.com](mailto:user@host.com)',但不匹配'<[user@host.com](mailto:user@host.com)'和'[user@host.com](mailto:user@host.com)>'。为什么,就是因为后面有一个(?(1)>|$) 我们先看看其中的(1) 表示什么,表示正则表达式最前面的那个括号,也就是邮件地址匹配正则表达式中的(<) 如果这个括号匹配到了内容,也就是检查看到了邮件地址以'<'字符开始,那么,我们对于该邮件地址末尾字符的检测就是>,如果(<)检测失败,也就是邮件地址不以'<'开头,那么我们对于该邮件地址末尾字符的检测就是'$'

\A

只匹配字符串的开头。只能放在正则表达式开头

\b

匹配空字符串,但仅限于单词的开头或结尾。一个单词被定义为一个单词字符序列。 请注意,在形式上,\b 被定义为 \w 和 \W 字符之间的界限,或 \w 和字符串的开始/结尾之间的界限。这意味着r'\bfoo\b' 匹配 'foo' , 'foo.' , '(foo)' , 'bar foo baz' ,但不匹配'foobar'或'foo3'。

默认情况下,Unicode字母数字是Unicode模式中使用的字母数字,但可以通过使用ASCII标志来更改。如果使用LOCALE标志,字边界由当前的区域设置确定。在字符范围内,\ b代表退格字符,以便与Python中的字符串文字兼容。

\B

匹配空字符串,但仅限于它不在单词的开头或结尾。 这意味着r'py\B'匹配'python','py3','py2',但不匹配'py','py。'或'py！'。 \B与\b相反,因此Unicode模式中的单词字符(\w)是Unicode字母数字加下划线。 如果使用LOCALE标志,单词的定义由当前的区域设置确定。

**正则表达式对象**

我们写好一条规则,如r'.ello',目前在python里面只是一个字符串,我们要让这个规则字符串变成正则表达式,需要将这个字符串编译一下,成为表达式对象,这个对象又称pattern,即样式。

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

通过上面的模块函数,根据我们写的规则字符串,生成一个正则表达式对象。

序列

# regex = r'你的规则'

pattern = re.compile(regex) #这里 regex只是规则字符串,pattern才是正则表达式对象

result = pattern.match(string)

等价于

result = re.match(regex, string)

使用re.compile()生成的正则表达式对象可以重用,更高效。因此,推荐使用第一种用法,第二种方法明显是在背后偷偷编译了个 pattern,只是为了方便,快捷地使用而已。

**正则表达式对象支持以下方法**

**pattern.search(string[, pos[, endpos]])**

扫描字符串查找第一个能匹配上pattern的部分,并返回相应的匹配对象。 如果字符串中没有位置与模式匹配,则返回None; 请注意,这与在字符串中的某处找到零长度匹配不同。如果你本身要匹配的就是一个空字符,如上述规则中的r'\b',那么返回一个""表示的是找到了对应的r'\b',也就是找到了空字符,而返回None找不到任何匹配的空字符。

可选的第二个参数pos在搜索要开始的字符串中给出一个索引,它默认为0,从指定的pos开始搜索,也就是说避开字符串开始的部分。pattern.search("hello world") 表示在 hello world 中查找  
pattern.search("hello world",2) 表示在 llo world 中查找

可选参数endpos表示搜索字符串的截止位置,也就是说,你想避开字符串结尾的部分。只有从pos到endpos - 1的字符才会被搜索到。如果endpos小于pos,则不会找到匹配,否则pattern.search(string,0,50)等同于pattern.search(string[:50],0)。

>>> pattern = re.compile("d")

>>> pattern.search("dog") # 能匹配

<re.Match object; span=(0, 1), match='d'>

>>> pattern.search("dog", 1) # 不能匹配

我们可以看到,我们首先,写好一个规则d,这是一个具体的规则,没有用到任何特殊符号,接着,我们把这条规则编译成pattern,这是一个正则表达式对象,然后使用pattern的search函数,在字符串"dog"中搜索。匹配的结果返回一个Match对象,之后会详细讲解这个Match对象。

**Pattern.match(string[, pos[, endpos]])**

如果字符串开头的零个或多个字符与此正则表达式匹配,则返回相应的匹配对象。 如果字符串与模式不匹配,则返回None; 请注意,这与零长度匹配不同。

可选的pos和endpos参数与search()方法具有相同的含义。

>>> pattern = re.compile(r"o")

>>> pattern.match("dog") # 不匹配,因为 "o" 不是 "dog"的最前面.

>>> pattern.match("dog", 1) # 能匹配 "o" 是 "dog" 的第二个字母.

<re.Match object; span=(1, 2), match='o'>;

这个函数和search差不多,但是,规定一定要从起始位置就得匹配上,否则就不算匹配成功。比如,我们在"dog"中搜索o但是,由于开始的位置不是o所以匹配失败。  
如果您想在字符串中的任何位置找到匹配项,请改用search()(另请参阅search()与match())。

**pattern.fullmatch(string[, pos[, endpos]])**

如果整个字符串匹配此正则表达式,则返回相应的匹配对象。 如果字符串与模式不匹配,则返回None; 请注意,这与零长度匹配不同。

可选的pos和endpos参数与search()方法具有相同的含义。

>>> pattern = re.compile("o[gh]")

>>> pattern.fullmatch("dog") # No match as "o" is not at the start of "dog".

>>> pattern.fullmatch("ogre") # No match as not the full string matches.

>>> pattern.fullmatch("doggie", 1, 3) # Matches within given limits.

<re.Match object; span=(1, 3), match='og'>

这个函数规定整个字符串要跟pattern匹配上,才算匹配成功。

**pattern.split(string, maxsplit=0)**

平时,如果我们想拆分一个字符串,python内置的split函数需要写入固定的分拆字符,比如下面的字符串"abc1efg1rgh" 我们可以以'1'字符来分拆这个字符串,但是万一,分割字符串的不只是'1'而是其他数字怎么办？如"abc1efg2hij3klm"python自带的split就束手无策了。这时候就需要用到正则表达式的拆分了,你会发现解决这个问题是分分钟的事。

>>> pattern = re.compile(r'[0-9]')

>>> pattern.split("abc1efg2hij3klm")

['abc', 'efg', 'hij', 'klm']

这个函数根据表达式规则分拆字符串,返回一个list。（温馨提示,后面这点内容可以跳过）如果在表达式中使用捕获括号,则表达式中所有组的文本也会作为结果列表的一部分返回。 如果maxsplit不为零,则最多发生maxsplit分割,并且字符串的其余部分作为列表的最后一个元素返回。

>>> pattern = re.compile(r'\W+')

>>> pattern.split( 'Words,words,words.')

['Words', 'words', 'words', '']

>>> pattern = re.compile(r'(\W+)')

>>> pattern.split( 'Words, words, words.')

['Words', ', ', 'words', ', ', 'words', '.', '']

>>> pattern = re.compile(r'\W+')

>>> pattern.split('Words, words, words.', 1)

['Words', 'words, words.']

>>> pattern = re.compile('[a-f]+',flags=re.IGNORECASE)

>>> pattern.split( '0a3B9' )

['0', '3', '9']

如果分隔符中有捕获组,并且它在字符串的起始处匹配,则结果将以空字符串开头。 字符串的结尾也是一样:

>>> pattern = re.compile(r'(\W+)')

>>> pattern.split( '...words, words...')

['', '...', 'words', ', ', 'words', '...', '']

这样,分隔符组件总是在结果列表中的相同索引处找到。

**pattern.findall(string[, pos[, endpos]])**

返回字符串中模式的所有非重叠匹配项,作为字符串列表。 字符串从左到右扫描,匹配按照找到的顺序返回。 如果模式中存在一个或多个组,返回组列表; 如果模式有多个组,这将是一个元组列表。 结果中包含空匹配项。

比如,我们想要查找一个字符串里面全部的数字

>>> pattern = re.compile(r'\d+')

>>> pattern.findall("a12b56c54d89")

['12', '56', '54', '89']

如果有分组,比如,我们需要把查找的数字的末尾作为分组提取出来,那么返回的就是数字的最后一位

>>> pattern = re.compile(r'\d\*([0-9])')

>>> pattern.findall("a124b567c54d892")

['4', '7', '4', '2']

r'\d\*([0-9])' 这条规则本来匹配的是下面的123,567,54,892但是这里由于有了后面的括号,所以只返回括号里面的

如果有多个分组,则返回这些分组的tuple,比如,要返回数字的个位数和十位数

>>> pattern = re.compile(r'\d\*([0-9])([0-9])')

>>> pattern.findall("a124b567c54d892")

[('2', '4'), ('6', '7'), ('5', '4'), ('9', '2')]

**pattern.finditer(string[, pos[, endpos]])**

返回一个迭代器,产生字符串中RE模式的所有非重叠匹配的匹配对象。 字符串从左到右扫描,匹配按照找到的顺序返回。  
和上面唯一的区别是返回的是一个iter而不是一个list,如果你对Python熟悉的话应该了解这两者的区别,如果不熟悉的话,建议使用上一种方法就好了,不要管这个。  
接受可选的pos和endpos参数,这些参数限制搜索区域,如search()。

**pattern.sub(repl, string, count=0)**

python自带replace函数的加强版。  
先通过正则表达式规则找到string中符合规则的部分,然后替换成repl  
如果未找到能匹配规则的部分,则字符串将保持不变。 repl可以是一个字符串或一个函数; 如果它是一个字符串,则处理其中的任何反斜杠转义。 也就是\n被转换为单个换行符,\r被转换为回车符,等等。 像 & 一样的未知转义单独保留。引用(例如\6)被替换为模式中由组6匹配的子字符串。 例如: 我们把分割字母的数字换成'000'

>>> pattern = re.compile(r'\d+')

>>> pattern.sub('000', '12abc34de56fg89')

'000abc000de000fg000'

repl可以引用分组,因此,假如我们要把分割字母的数字换成他们的个位数,可以这样.  
注意这里repl需要用raw字符串,否则re模块无法识别'\1',还是那个''的问题。所以,能用raw尽量用raw

>>> pattern = re.compile(r'\d\*(\d)')

>>> pattern.sub(r'\1','12abc34de56fg89') # 这里的r'\1'对应的就是上面 r'\d\*(\d)'中的(\d)

'2abc4de6fg9'

如果repl是一个函数,它会实现更加复杂的替换,比如我们需要把分割字母的数字翻倍。也就是,我们我们每个匹配到的数字,我们都要通过函数处理一下,以返回值作为repl 例如:

>>> def func(n):

... return str(int(n.group())\*2)

...

>>> pattern = re.compile(r'\d+')

>>> pattern = re.sub(func,'12abc34de56fg89')

'24abc68de112fg178'

注意到,传给func的是一个个匹配对象(后面会讲到),该对象包装了真正匹配到的数字,如'12','34','56','89',所以使用了一个 n.group()来提取数字。匹配对象的强大之处还在于我们不仅可以提取完整的匹配数字如'12',我们还可以提取分组,比如,我们在pattern中设定了个位数字作为组,那么就可以提取出来,如下面的例子,把找到的数字替换成该数字的个位数的两倍

>>> def func(n):

... return str(int(n.group(1))\*2)

# 对比上面n.group()其实是n.group(0)表示匹配的字符串,也就是整个r'\d\*(\d)' 能匹配到的字符串

# n.group(1) 对应的就是 r'\d\*(\d)' 中的 (\d)匹配到的东西

...

>>> pattern = re.compile(r'\d\*(\d)')

>>> pattern.sub(func,'12abc34de56fg89')

'4abc8de12fg18'

关于匹配对象,我们在后面还会有详细的讲解。

**pattern.subn(repl, string, count=0)**

执行与sub()相同的操作,但返回一个元组(new\_string,number\_of\_subs\_made),也就是不仅返回一个和上面函数一样的字符串,还多返回了一个数字,代表了总共替换的次数,像上面的例子

>>> pattern.subn(func,'12abc34de56fg89')

('4abc8de12fg18', 4)

**正则表达式对象的属性:**

**pattern.flags**

正则表达式匹配标志。在re.compile()函数中设定,比如要忽略大小写

>>> pattern = re.compile(r'd',flags=re.IGNORECASE)

>>> pattern.findall('DOG')

['D']

如果要设置多个flags,可以用 | 隔开

>>> pattern = re.compile(r'd',flags=re.IGNORECASE | re.MULTILINE)

当然,设置flags也可以通过内联的方式

```python

>>> pattern = re.compile(r'(?im)d')

>>> pattern = re.compile(r'd',flags=re.IGNORECASE | re.MULTILINE)

上面两者是等价的。

**pattern.groups**

查看模式中的捕获组数量。也就是你在规则中使用捕获括号的数量  
比如 p = re.compile(r'hello(\d)') 那么这时 p.groups 就是 1

**pattern.groupindex**

如果你在规则设定的时候使用了(?P<name>),那么这个变量可以返回该特殊符号使用的情况。  
将(?P<id>)定义的任何符号组名称映射到组编号的字典。 如果模式中没有使用符号组,则字典为空。

**pattern.pattern**

模式对象编译的模式字符串。就是那个写下的规则。

>>> p = re.compile(r'(?im)d')

>>> p.pattern

'(?im)d'

**re模块自带的函数**

我们上面学习了使用正则表达式的两个步骤,首先编译出正则表达式对象pattern,然后,调用pattern的search,findall,match等函数进行匹配等操作,实际上re直接提供了search,findall,match等快捷操作,允许我们直接操作,而不需先编译pattern,而是直接把规则写在操作函数之中。这样的操作比上面提到的方法减少了一行,有了更好地便捷性,但是也牺牲了复用性等功能。

**re.match(pattern, string, flags=0)**

下面的两个匹配操作是等价

>>> pattern = re.compile(r"o")

>>> pattern.match("dog") # No match as "o" is not at the start of "dog".

>>> re.match(r"o","dog") # No match as "o" is not at the start of "dog".

事实上,第二种直接操作在背后也是先用第一个参数编译成正则表达式对象然后在让这个对象调用match方法。其他的search,findall,split等操作也是一样的。

**re.search(pattern, string, flags=0)**

参考pattern.search

**re.fullmatch(pattern, string, flags=0)**

参考pattern.full

**re.split(pattern, string, maxsplit=0, flags=0)**

参考pattern.split

**re.findall(pattern, string, flags=0)**

参考pattern.findall

**re.finditer(pattern, string, flags=0)**

参考pattern.finditer

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

参考pattern.sub

**re.subn(pattern, repl, string, count=0, flags=0)**

执行与sub()相同的操作,但返回一个元组(new\_string,number\_of\_subs\_made)。

**re.escape(pattern)**

让pattern中的特殊字符失去意义。 如果您想匹配任何可能具有正则表达式元字符的文字字符串,这非常有用。 例如:

>>> print(re.escape('python.exe'))

python\.exe

>>> p = re.compile(re.escape(r'.')) # 等价于 p = re.compile(r'\.')

>>> p.search("abc")

# 不能匹配

>>> p.search(".")

<\_sre.SRE\_Match object at 0x05370D08>

再来一个例子

>>> operators = ['+', '-', '\*', '/', '\*\*']

>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))

/|\-|\+|\\*\\*|\\*

**re模块自带的参数**

**re.A == re.ASCII**

使\w,\W,\b,\B,\d,\D,\s和\S执行仅ASCII匹配而不是完全Unicode匹配。 这只对Unicode模式有意义,并且在字节模式中被忽略。 对应于内联标志(?a)。

请注意,为了向后兼容,re.U标志仍然存在(以及它的同义词re.UNICODE及其嵌入对象(?u)),但这些在Python 3中是多余的,因为默认情况下匹配是Unicode的Unicode(和Unicode 字节不允许匹配)。

**re.I == re.IGNORECASE**

执行不区分大小写的匹配; 像[A-Z]这样的表达式也将匹配小写字母。 除非使用re.ASCII标志来禁用非ASCII匹配,否则完全的Unicode匹配(例如Ü匹配ü)也是有效的。 除非使用re.LOCALE标志,否则当前语言环境不会更改此标志的效果。 对应于内联标志(?i)。

请注意,当Unicode模式[a-z]或[A-Z]与IGNORECASE标志组合使用时,它们将与52个ASCII字母和另外4个非ASCII字母匹配:'İ'(U + 0130,拉丁语大写字母I与 (U + 0131,拉丁小字母无点i),'s'(U + 017F,拉丁小写字母长)和'K'(U + 212A,开尔文符号)。 如果使用ASCII标志,只匹配字母'a'到'z'和'A'到'Z'。

**re.L == re.LOCALE**

根据当前语言环境,使\ w,\ W,\ b,\ B和不区分大小写的匹配。 该标志只能用于字节模式。 由于区域设置机制非常不可靠,因此不鼓励使用此标志,它一次只处理一种"文化",并且仅适用于8位语言环境。 对于Unicode(str)模式,默认情况下,Unicode匹配已在Python 3中启用,并且它能够处理不同的语言环境/语言。 对应于内联标志(?L)。

在版本3.6中更改:re.LOCALE只能与字节模式一起使用,并且与re.ASCII不兼容。

在版本3.7中更改:使用re.LOCALE标志编译的正则表达式对象在编译时不再依赖于语言环境。 匹配时只有语言环境会影响匹配结果。

**re.M == re.MULTILINE**

指定时,模式字符'^'匹配字符串的开头和每行的开头(紧跟在每个换行符之后); 并且模式字符'$'匹配字符串的末尾和每行末尾(紧接在每个换行符之前)。 默认情况下,'^'只匹配字符串的开头,'$'只匹配字符串的末尾,紧接在字符串末尾的换行符(如果有的话)之前。 对应于内联标志(?m)。

**re.S == re.DOTALL**

制作'.' 特殊字符完全匹配任何字符,包括换行符; 没有这个标志,'.' 将匹配除换行符之外的任何内容。 对应于内联标志(?s)。

**re.X == re.VERBOSE**

该标志允许您通过一种编辑模式来编写正则表达式,允许您在视觉上分离模式的逻辑部分并添加注释,该正则表达式看起来更好,并且更易读。模式中的空格被忽略,除非在字符类中,或者前面有一个未转义的反斜杠,或者在诸如 \*? , (?: 或 (?P<...>)。我们还可以通过#号进行注释。

这意味着匹配一个十进制数的下面两个正则表达式对象在功能上是相等的:

a = re.compile(r"""\d + # the integral part

\. # the decimal point

\d \* # some fractional digits""", re.X)

b = re.compile(r"\d+\.\d\*")

对应于内联标志(?x)。

**Match Objects 匹配的结果对象**

我们之前就提到过当我们用写好的规则去匹配一个字符串的时候,返回的不是匹配好的子字符串！而是返回一个匹配对象,里面除了包装好子字符串,还提供其它的功能,因为,我们在写规则的时候有通过捕获括号对规则进行分组,因此,我们也可以通过匹配对象把分组给提取出来。比如hello(\w) 规则能够匹配 'helloA',如果直接返回'helloA',那我们设置的捕获括号不就失去意义了吗,而匹配对象则正是能够完成这些功能的关键所在。

匹配对象始终具有布尔值True。 由于match()和search()在不匹配时返回None,因此可以通过简单的if语句测试是否成功匹配:

match = re.search(pattern, string)

if match:

process(match)

匹配对象支持以下方法和属性:

**match.expand(template)**

我们设定一个模板,来显示我们查找到的子字符串,其中,我们可以反斜杠替换的方式嵌入捕获括号所匹配到的子字符串。  
数字引用(\1, \2)或命名反斜线引用(\g<1>,\g <name>)被替换为相应组的内容。  
比如我们要展示某个字符串中的数字,并列出个位数。

import re

p = re.compile(r'\d+(\d)(\d)')

match = p.search("adsfaddf12345fgd")

print(match .expand(r'The number in the string is \g<0>.\nThe last number is \2'))

The number in the string is 12345.

The last number is 5

**match.group([group1, ...])**

返回匹配的一个或多个子组。 如果只有一个参数,结果是一个单独的字符串; 如果有多个参数,则结果是一个tuple。  
没有参数,group默认为零(整个匹配被返回)。 如果group参数为零,则相应的返回值是整个匹配的字符串; 如果参数在包含范围[1..99]中,则它是匹配相应括号组的字符串。 如果组编号为负数或大于模式中定义的组数,则会引发IndexError异常。 如果一个组包含在不匹配的模式的一部分中,则相应的结果为无。 如果一个组包含在多次匹配的模式的一部分中,则返回最后的匹配。

>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

>>> m.group(0) # 整个的匹配结果

'Isaac Newton'

>>> m.group(1) # 第一个括号匹配到的结果

'Isaac'

>>> m.group(2) # 第二个括号匹配到的结果

'Newton'

>>> m.group(1, 2) # 返回多个结果

('Isaac', 'Newton')

如果正则表达式使用 (?P<name>...)语法,则group参数也可以是通过组名称标识组的字符串。 如果字符串参数未在模式中用作组名称,则会引发IndexError异常。

>>> m = re.match(r"(?P<first\_name>\w+) (?P<last\_name>\w+)", "Malcolm Reynolds")

>>> m.group('first\_name')

'Malcolm'

>>> m.group('last\_name')

'Reynolds'

命名组也可以通过它们的索引来引用:

>>> m.group(1)

'Malcolm'

>>> m.group(2)

'Reynolds'

如果一个组匹配多次,只能访问最后一场匹配:

>>> m = re.match(r"(..)+", "a1b2c3") # 由于贪婪模式，这个表达式明显匹配的结果是全部的字符串

# 但是其中的括号匹配到的是什么呢？

>>> m.group(1) # 返回最后能匹配到的

'c3'

读者可以自行试试这个例子，看看 m.group() m.group(0) m.group(1) m.group(2) 分别是什么

**match.getitem(g)**

如果你熟悉python就知道这个属性就像dict一样,让我们能够直接用[]的形式取出数据,而不需要使用group函数,更加方便快捷。  
这与m.group(g)相同。 这允许从比赛中更容易地访问个人组:

>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

>>> m[0] # 等价于 m.group(0)

'Isaac Newton'

>>> m[1] #

'Isaac'

>>> m[2] #

'Newton'

**match.groups(default=None)**

返回一个包含匹配所有子组的元组,从1开始,直到模式中有多个组。 default参数用于补齐没能成功匹配的组; 它默认为None。

例如:

>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")

>>> m.groups()

('24', '1632')

如果我们将小数点后的位置及其后的所有内容都设为可选,则并非所有组都可以参与该匹配。 除非给出默认参数,否则这些组将默认为None。

>>> m = re.match(r"(\d+)\.?(\d+)?", "24")

>>> m.groups() # 第二组默认为 None.

('24', None)

>>> m.groups('0') # Now, the second group defaults to '0'.

('24', '0')

**match.groupdict(default=None)**

返回包含匹配的所有命名子组的字典的子集名称。 缺省参数用于未参与匹配的组; 它默认为None。 例如:

>>> m = re.match(r"(?P<first\_name>\w+) (?P<last\_name>\w+)", "Malcolm Reynolds")

>>> m.groupdict()

{'first\_name': 'Malcolm', 'last\_name': 'Reynolds'}

**match.start([group])**

**match.end([group])**

返回按组匹配的子串在原字符串中开始和结束的位置;

组默认为零,意味着整个匹配的子字符串,一个将从电子邮件地址中删除remove\_this的示例:

>>> email = "tony@tiremove\_thisger.net"

>>> m = re.search("remove\_this", email)

>>> email[:m.start()] + email[m.end():]

'tony@tiger.net'

>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")

>>> m.start(0) # 整个匹配到的是 "Isaac Newton" 其中"I"在原字符串的开头，因此是 0

0

>>> m.end(0) # 整个匹配到的是 "Isaac Newton" 其中"n"在原字符串的第12位，因此是 12

12

>>> m.start(1) # 第一组匹配到的是 "Isaac" 其中"I"在原字符串的第0位，因此是 0

0

>>> m.end(1) # 第一组匹配到的是 "Isaac" 其中"c"在原字符串的第5位，因此是 5

5

>>> m.start(2) # 第2组匹配到的是 "Newton" 其中"N"在原字符串的第6位，因此是 6

6

>>> m.end(2) # 第2组匹配到的是 "Newton" 其中"n"在原字符串的第12位，因此是 12

12

**match.span([group])**

对于匹配m,返回2元组(m.start(group),m.end(group))。 请注意,如果组没有参与匹配,则为(-1,-1)。 组默认为零。对于上面的例子 m.span() 也就是 m.span(0) 返回 (0,12) m.span(1) 返回 (0,5) m.group(2) 返回 (6,12)

**Match Object的一些参数**

**match.pos**

传递给正则表达式对象的search()或match()方法的pos的值。 这是RE引擎开始寻找匹配的字符串的索引。根据此值,我们能够知道在引擎搜索时设定的pos

**match.endpos**

传递给正则表达式对象的search()或match()方法的endpos的值。 这是RE引擎不会去的字符串的索引。

**match.lastindex**

最后一个匹配捕获组的整数索引,或者如果没有匹配组,则为None。 例如,如果将表达式(a)b,((a)(b))和((ab))应用于字符串"ab",则lastindex == 1,而表达式(a)(b)将 如果应用于相同的字符串,则lastindex == 2。

**match.lastgroup**

最后匹配的捕获组的名称,如果组没有名称,或者根本没有匹配组,则为None。

**match.re**

正则表达式对象的match()或search()方法生成此匹配实例。

**match.string**

传递给match()或search()的字符串。

**一些例子**

**寻找对子**

在这个例子中,我们将使用以下辅助函数来更加优雅地显示匹配对象:

def displaymatch(match):

if match is None:

return None

return '<Match: %r, groups=%r>' % (match.group(), match.groups())

假设您正在编写一个扑克程序,其中玩家的手牌为5个字符的字符串,每个字符代表一张牌,"a"代表王牌,"k"代表国王,"q"代表女王,"j"代表插孔, "t"为10,"2"至"9"代表具有该值的卡。

要查看给定的字符串是否是有效的,可以执行以下操作:

>>> valid = re.compile(r"^[a2-9tjqk]{5}$")

>>> displaymatch(valid.match("akt5q")) # Valid.

"<Match: 'akt5q', groups=()>"

>>> displaymatch(valid.match("akt5e")) # Invalid.

>>> displaymatch(valid.match("akt")) # Invalid.

>>> displaymatch(valid.match("727ak")) # Valid.

"<Match: '727ak', groups=()>"

最后一只手牌,"727ak",包含一对,或两个相同的价值卡。 为了与正则表达式匹配,可以使用反斜线引用:

>>> pair = re.compile(r".\*(.).\*\1")

>>> displaymatch(pair.match("717ak")) # Pair of 7s.

"<Match: '717', groups=('7',)>"

>>> displaymatch(pair.match("718ak")) # No pairs.

>>> displaymatch(pair.match("354aa")) # Pair of aces.

"<Match: '354aa', groups=('a',)>"

为了找出这对卡片组成的卡片,可以按照以下方式使用匹配对象的group()方法:

>>> pair.match("717ak").group(1)

'7'

# Error because re.match() returns None, which doesn't have a group() method:

>>> pair.match("718ak").group(1)

Traceback (most recent call last):

File "<pyshell#23>", line 1, in <module>

re.match(r".\*(.).\*\1", "718ak").group(1)

AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)

'a'

**search() vs. match()**

Python提供了基于正则表达式的两种不同的基本操作:re.match()仅在字符串的开始处检查匹配,而re.search()检查字符串中任意位置的匹配(这是Perl默认执行的操作)。

例如:

>>> re.match("c", "abcdef") # No match

>>> re.search("c", "abcdef") # Match

<re.Match object; span=(2, 3), match='c'>

以'^'开头的正则表达式可以与search()一起用于限制字符串开始处的匹配:

>>> re.match("c", "abcdef") # No match

>>> re.search("^c", "abcdef") # No match

>>> re.search("^a", "abcdef") # Match

<re.Match object; span=(0, 1), match='a'>

但是请注意,在MULTILINE模式下match()只匹配字符串的开头,而使用search()与以'^'开头的正则表达式匹配每行的开头。

>>> re.match('X', 'A\nB\nX', re.MULTILINE) # No match

>>> re.search('^X', 'A\nB\nX', re.MULTILINE) # Match

<re.Match object; span=(4, 5), match='X'>

**做一个电话本**

split()将字符串分割成由正则表达式分隔的列表。该方法对于将文本数据转换为可由Python轻松读取和修改的数据结构非常有用,如以下创建电话簿的示例所示。

首先,这是输入。通常它可能来自一个文件:

>>> text = """Ross McFluff: 834.345.1254 155 Elm Street

...

... Ronald Heathmore: 892.345.3428 436 Finley Avenue

... Frank Burger: 925.541.7625 662 South Dogwood Way

...

...

... Heather Albrecht: 548.326.4584 919 Park Place"""

条目由一个或多个换行符分隔。 现在我们将字符串转换为一个列表,每个非空行都有自己的条目:

>>> entries = re.split("\n+", text)

>>> entries

['Ross McFluff: 834.345.1254 155 Elm Street',

'Ronald Heathmore: 892.345.3428 436 Finley Avenue',

'Frank Burger: 925.541.7625 662 South Dogwood Way',

'Heather Albrecht: 548.326.4584 919 Park Place']

最后,将每个条目分成一个名字,姓氏,电话号码和地址。 因为地址中有空格,为了不把地址分隔开,我们使用split()的maxsplit参数,:

>>> [re.split(":? ", entry, 3) for entry in entries]

[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],

['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],

['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],

['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]

我们可以将最多的房屋号码与街道名称分开:

>>> [re.split(":? ", entry, 4) for entry in entries]

[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],

['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],

['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],

['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]

**Text Munging**

Mung或munge是计算机术语,用于对一段数据或文件进行一系列潜在的破坏性或不可撤销的更改。它有时用于说话人尚不清楚的模糊数据转换步骤。常见的搜索操作包括删除标点或html标签,数据解析,过滤和转换。 [wiki]

sub()用一个字符串或一个函数的结果替换每个匹配规则的子部分。 这个例子演示了如何使用sub()和函数来"munge"文本,或者随机化除了第一个和最后一个字符之外的每个单词中所有字符的顺序:

>>> def repl(m):

... inner\_word = list(m.group(2))

... random.shuffle(inner\_word)

... return m.group(1) + "".join(inner\_word) + m.group(3)

>>> text = "Professor Abdolmalek, please report your absences promptly."

>>> re.sub(r"(\w)(\w+)(\w)", repl, text)

'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'

>>> re.sub(r"(\w)(\w+)(\w)", repl, text)

'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'

**寻找所有的动词**

findall()匹配所有匹配的子部分,而不仅仅是search()所做的第一个。 例如,如果一个人是作家,并且想要在某些文本中找到所有副词,他或她可以按以下方式使用findall():

>>> text = "He was carefully disguised but captured quickly by police."

>>> re.findall(r"\w+ly", text)

['carefully', 'quickly']

**寻找所有的动词和对应位置**

如果想要获得关于匹配文本的所有匹配的更多信息,finditer()很有用,因为它提供了匹配对象而不是字符串。 继续前面的例子,如果一个作家想要在某些文本中找到所有副词及其位置,他或她会按以下方式使用finditer():

>>> text = "He was carefully disguised but captured quickly by police."

>>> for m in re.finditer(r"\w+ly", text):

... print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))

07-16: carefully

40-47: quickly

**原始字符串符号**

原始字符串符号(r"text")使正则表达式保持正常。 没有它,正则表达式中的每个反斜杠( '' )必须以另一个反斜杠作为前缀。 例如,以下两行代码在功能上是相同的:

>>> re.match(r"\W(.)\1\W", " ff ")

<re.Match object; span=(0, 4), match=' ff '>

>>> re.match("\\W(.)\\1\\W", " ff ")

<re.Match object; span=(0, 4), match=' ff '>

当想要匹配文字反斜杠时,它必须在正则表达式中转义。 用原始字符串表示法,这意味着r"\"。 没有原始字符串表示法,必须使用"\\",使以下代码行功能相同:

>>> re.match(r"\\", r"\\")

<re.Match object; span=(0, 1), match='\\'>

>>> re.match("\\\\", r"\\")

<re.Match object; span=(0, 1), match='\\'>

总而言之,最好使用raw字符串

**写一个分词器**

分词器或扫描器分析字符串以对字符组进行分类。这是编写编译器或解释器的第一步。所以,理论上来说,学完re模块就可以写一个属于自己的计算机语言,努力吧,说不定你的语言会成为下一个python,成为广受欢迎的语言。

下面就是告诉大家如何写一个分词器,让你的编译器识别代码中的关键字,识别符,值等等元素。  
编译器首先需要把代码中的一个个词拆开分类为一个个token  
一个token有'type' 属性,'value'属性,'line'属性,'column'属性  
一条代码 'if a > 1:' 就会被分为5个token  
第一个token, 'type' 是关键字 , 'value'是IF, 'line'是该语句所在的行数, 'column'是 if在该行从左往右数的位置  
第二个token, 'type'是 标识符(ID) 'value'是 a,  
第三个token, 'type'是 操作符(Operator) 'value'是 '>'  
第四个token, 'type'是 数量(Number) 'value'是 1  
第五个token, 'type'是 判断结束符(EndOfIf) 'value'是 :  
就这样把代码拆开,准确地识别各个token,是每一个编译器工作的第一步。  
token的结构可以由你自己定义,但一般来说都要都有type属性和value属性,你还可以自己添加其他的属性来方便编译器工作,这取决于你的编程水平。

import collections

import re

# 这是一个简单的定义一个class的方法,我们定义一个token类,我们把代码拆开然后根据正则表达式的识别将其实例化为一个个token

Token = collections.namedtuple('Token', ['type', 'value', 'line', 'column'])

def tokenize(code):

# 定义我们的编译器有哪些关键字,不同的语言关键字是不一样的,需要语言开发者自行定义

keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}

# 语言除了关键字,还有其他的类型

token\_specification = [

# 定义我们语言能使用的数值

('NUMBER', r'\d+(\.\d\*)?'), # Integer or decimal number

# 定义语言的赋值符号

('ASSIGN', r':='), # Assignment operator

#定义语句的结束标识,一般来说都是分号,我们也用分号来标识结尾吧

('END', r';'), # Statement terminator

#标识符,比如变量的名字,就是一个标识符,我们这里规则变量只能用字母连下划线都不能用

('ID', r'[A-Za-z]+'), # Identifiers

#定义运算符,我们的语言能进行+-\*/

('OP', r'[+\-\*/]'), # Arithmetic operators

#要能够识别新的一行代码

('NEWLINE', r'\n'), # Line endings

#要能够识别各种空白

('SKIP', r'[ \t]+'), # Skip over spaces and tabs

#其他任何的东西我们都是别错误匹配,相当于写了错误的语句,我们的语言要报错,比如你在python里面写prinf("hello")肯定是错误的,这是C里面的语句

('MISMATCH',r'.'), # Any other character

]

#我们把上面的正则表达式用或串起来

tok\_regex = '|'.join('(?P<%s>%s)' % pair for pair in token\_specification)

line\_num = 1

line\_start = 0

# 我们开始对代码进行识别

for mo in re.finditer(tok\_regex, code):

kind = mo.lastgroup

value = mo.group(kind)

if kind == 'NEWLINE':

line\_start = mo.end()

line\_num += 1

elif kind == 'SKIP':

pass

elif kind == 'MISMATCH':

raise RuntimeError(f'{value!r} unexpected on line {line\_num}')

else:

if kind == 'ID' and value in keywords:

kind = value

column = mo.start() - line\_start

yield Token(kind, value, line\_num, column)

statements = '''

IF quantity THEN

total := total + price \* quantity;

tax := price \* 0.05;

ENDIF;

'''

# 把我们实例化的token打印出来

for token in tokenize(statements):

print(token)

分词器产生以下输出:

Token(type='IF', value='IF', line=2, column=4)

Token(type='ID', value='quantity', line=2, column=7)

Token(type='THEN', value='THEN', line=2, column=16)

Token(type='ID', value='total', line=3, column=8)

Token(type='ASSIGN', value=':=', line=3, column=14)

Token(type='ID', value='total', line=3, column=17)

Token(type='OP', value='+', line=3, column=23)

Token(type='ID', value='price', line=3, column=25)

Token(type='OP', value='\*', line=3, column=31)

Token(type='ID', value='quantity', line=3, column=33)

Token(type='END', value=';', line=3, column=41)

Token(type='ID', value='tax', line=4, column=8)

Token(type='ASSIGN', value=':=', line=4, column=12)

Token(type='ID', value='price', line=4, column=15)

Token(type='OP', value='\*', line=4, column=21)

Token(type='NUMBER', value='0.05', line=4, column=23)

Token(type='END', value=';', line=4, column=27)

Token(type='ENDIF', value='ENDIF',line=5, column=4)

Token(type='END', value=';', line=5, column=9)