**Pyparsing快速参考**

**作者：**

John W. Shipman / tcc-doc@nmt.edu

**原文链接：**http://infohost.nmt.edu/tcc/help/pubs/pyparsing/pyparsing.pdf

**翻译（按贡献程度排名）：**

谷歌翻译 translate.google.cn

必应翻译 <https://cn.bing.com/Translator>

老王 https://www.zhihu.com/people/lao-wang-96-3

目录

[译者前言 4](#_Toc4577)

[1、pyparsing：从文本（text）中提取信息的工具 5](#_Toc21882)

[2、构建您的应用 7](#_Toc7854)

[3、一个小而完整的例子 8](#_Toc11143)

[4、如何构造返回的ParseResults 11](#_Toc7757)

[5、类 17](#_Toc16889)

[5.1、ParserElement：基本的解析器构建单元 17](#_Toc19247)

[5.2、And：序列 28](#_Toc23115)

[5.3、CaselessKeyword：不区分大小写的关键字匹配 28](#_Toc22124)

[5.4、CaselessLiteral：不区分大小写的字符串匹配 29](#_Toc10396)

[5.5、CharsNotIn：匹配不在给定集合中的字符 29](#_Toc21890)

[5.6、Combine：组件融合 30](#_Toc16100)

[5.7、Dict：解析表格(文字待整理) 31](#_Toc25393)

[5.8、Each：组件可以按任意顺序出现 32](#_Toc736)

[5.9、Empty：匹配空内容 33](#_Toc13373)

[5.10、FollowedBy：添加前缀约束 33](#_Toc3163)

[5.11、Forward：解析器占位符 34](#_Toc9998)

[5.12、GoToColumn：前进到行中的指定位置【TODO：完全采用谷歌翻的文字，可能需要改进可读性】 38](#_Toc30396)

[5.13、Group：将重复的项目分组到列表中 39](#_Toc19853)

[5.14、Keyword：匹配与指定上下文不相邻的文字字符串 40](#_Toc18431)

[5.15、LineEnd：匹配行尾 40](#_Toc10925)

[5.16、LineStart：匹配行的开始 41](#_Toc24709)

[5.17、Literal：匹配特定字符串 42](#_Toc27013)

[5.18、MatchFirst：按给定顺序尝试多个匹配 42](#_Toc16994)

[5.19、NoMatch：一个永不匹配的解析器 43](#_Toc16865)

[5.20、NotAny：General lookahead condition 43](#_Toc13327)

[5.21、OneOrMore：重复一次或多次模式 44](#_Toc7861)

[5.22、Optional：匹配可选模式 45](#_Toc8613)

[5.23、Or：解析一组备选方案中的一个 45](#_Toc31964)

[5.24、ParseException 46](#_Toc30997)

[5.25、ParseFatalException：带我离开这里！ 46](#_Toc27903)

[5.26、ParseResults：从匹配返回的结果 47](#_Toc24821)

[5.27、QuotedString：匹配分隔的字符串 50](#_Toc14754)

[5.28、Regex：匹配正则表达式 51](#_Toc26330)

[5.29、SkipTo：提前搜索模式 52](#_Toc31398)

[5.30、StringEnd：匹配文本的结尾 53](#_Toc26966)

[5.31、StringStart：匹配文本的开头 53](#_Toc27070)

[5.32、Suppress：忽略结果中的匹配文本 54](#_Toc25361)

[5.33、Upcase：大写结果 54](#_Toc10152)

[5.34、White：匹配空白 55](#_Toc10143)

[5.35、Word：匹配指定集合中的字符 56](#_Toc24627)

[5.36、WordEnd：仅在单词的结尾处匹配 57](#_Toc26618)

[5.37、WordStart：仅在单词的开头匹配 58](#_Toc27470)

[5.38、ZeroOrMore：匹配任意数量的重复，包括None 59](#_Toc17350)

[6、函数 59](#_Toc15285)

[6.1、col()：将位置转换为列号 59](#_Toc5583)

[6.2、countedArray：解析N后跟N个事物 60](#_Toc12289)

[6.3、delimitedList()：为分隔列表创建解析器 61](#_Toc23902)

[6.4、dictOf()：从键/值对构建字典 62](#_Toc16665)

[6.5、downcaseTokens()：转换成小写的parse action 63](#_Toc29534)

[6.6、getTokensEndLoc()：找到tokens的结尾 64](#_Toc8543)

[6.7、line()：一个位置出现在哪一行？ 64](#_Toc29022)

[6.8、lineno()：将位置转换为行号 65](#_Toc14859)

[6.9、matchOnlyAtCol()：从指定的列开始匹配 65](#_Toc8695)

[6.11、matchPreviousLiteral()： 66](#_Toc8334)

[6.12、nestedExpr()：嵌套列表的解析器 67](#_Toc19748)

[6.13、oneOf()：匹配多个文字（中的一个），优先检查最长的 68](#_Toc11263)

[6.14、srange()：指定字符范围 69](#_Toc2184)

[6.15、removeQuotes()：去掉开头结尾的引号 70](#_Toc21200)

[6.17、traceParseAction()：可以输出调试信息的parse action装饰器 70](#_Toc12986)

[6.18、upcaseTokens()：转换成大写的解析动作 71](#_Toc31594)

[7、变量 71](#_Toc14855)

[7.1、alphanums：字母数字字符 71](#_Toc32344)

[7.2、alphas：字母 72](#_Toc4695)

[7.3、alphas8bit：补充Unicode字母 72](#_Toc20273)

[7.4、cStyleComment：匹配C语言注释 72](#_Toc6928)

[7.5、commaSeparatedList：解析以逗号分隔的列表 72](#_Toc10589)

[7.6、cppStyleComment：解析C++注释 73](#_Toc32041)

[7.7、dblQuotedString：用“......”括起来的字符串 73](#_Toc18529)

[7.8、dblSlashComment：解析以“//”开头的注释 74](#_Toc2768)

[7.9、empty：匹配空内容 74](#_Toc12140)

[7.10、hexnums：全部十六进制数字 74](#_Toc13649)

[7.11、javaStyleComment：Java语法的注释 74](#_Toc25122)

[7.12、lineEnd：LineEnd的一个实例 75](#_Toc4897)

[7.13、lineStart：LineStart的一个实例 75](#_Toc16642)

[7.14、nums：十进制数字 76](#_Toc32560)

[7.15、printables：所有可打印的非空白字符 76](#_Toc3558)

[7.16、punc8bit：一些Unicode标点符号 77](#_Toc25590)

[7.17、pythonStyleComment：Python语言风格的注释 77](#_Toc346)

[7.18、quotedString：解析默认的字符串 78](#_Toc6113)

[7.19、restOfLine：匹配当前行的剩余部分 78](#_Toc24550)

[7.20、sglQuotedString：用'......'括起来的字符串 79](#_Toc7674)

[7.21、stringEnd：匹配字符串的结尾 79](#_Toc13956)

[7.22、unicodeString：匹配Python样式的Unicode字符串 80](#_Toc29958)

# 译者前言

在尝试解析SQL语句的过程中，查过pyparsing的资料，感觉中文资源还不够丰富，打算翻译一篇介绍pyparsing的文章，争取每天翻译一点：

http://infohost.nmt.edu/tcc/help/pubs/pyparsing/pyparsing.pdf​

Tips:

1、python还有个叫pyparse的库，似乎是心理学领域使用的解析音频文件的工具，不要搞混了。

2、本文有七部分组成，前四章为功能介绍，后三章为参考手册，第五章为类（Class）的介绍，第六章为函数的介绍，第七章为变量的介绍。

3、关于setParseAction函数。如果用pyparsing的目标，不只是提取字符串出来，而是，比如说，要把某个子parser提取出来的字符串转换成数据库的table名、字段名，可能就会用到setParseAction，在解析的过程中，实现从字符串到table name的转换。

这是stackoverflow上的一个例子：

<https://stackoverflow.com/questions/2941029/pyparsing-is-this-correct-use-of-setparseaction​stackoverflow.com>

# 1、pyparsing：从文本（text）中提取信息的工具

pyparsing模块的用途，是为python程序员提供一个工具，去提取结构化文本数据中的信息。

从功能上说，这个模块要比python自带的正则表达式更强大（re），又不像全功能的编译器那样通用。

为了在结构化文本（structured text）中查找信息，就要有定义结构的方法。pyparsing模块用巴科斯范式，或者叫BNF，来定义语法结构。熟悉基于BNF的各种语法符号，会有助于使用pyparsing模块。

Pyparsing 模块匹配输入文本中的模式，使用的是递归下降解析器: 我们编写类似BNF 的语法定义,pyparsing提供匹配这些语法定义的自动机。

待分析文本的语法结构，如果能被准确定义，那pyparsing就会工作在最佳状态。一个常见的pyparsing应用是分析日志文件。日志文件通常具有固定的结构，包括日期、IP地址等字段。本文不涉及自然语言处理中，可能用到pyparsing的场景。

有用的在线参考包括:

pyparsing的主页：[pyparsing/pyparsing](https://github.com/pyparsing/pyparsing" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

完整的在线参考手册，包括类、函数、变量的说明：[pyparsing/pyparsing](https://github.com/pyparsing/pyparsing/wiki" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

...

pyparsing作者2004年写的指南，虽然稍微过时，但仍然有用：[Using the pyparsing module](http://www.ptmcg.com/geo/python/howtousepyparsing.html" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

使用pyparsing的一个小示例（大约10个语法结构），请参考icalparse：[icalparse: A pyparsing parser for .calendar files](http://infohost.nmt.edu/tcc/help/lang/python/examples/icalparse/web/index.html" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

使用pyparsing的一个适度的示例，《abaras:A shorthand notation for bird records》，这个例子有大约30个语法结构。实际的实现在一份单独的文档中描述，《abaraw: internal maintenance specification》，基本上，这是个用pyparsing实现的内核，附加了一些转换成XML的应用逻辑，以便后继流程继续处理。链接1：[abaraw: A shorthand notation for bird records](http://infohost.nmt.edu/~shipman/aba/raw/doc/web/index.html" \t "https://zhuanlan.zhihu.com/p/61077860/_blank) 链接2：[abaraw internal maintenance specification](http://www.nmt.edu/~shipman/aba/raw/doc/ims/" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

# 2、构建您的应用

简单概括下用pyparsing模块写程序的一般步骤：

1\编写BNF，描述待分析文本的结构。

2\如果需要，安装pyparsing模块。大部分最近的python安装会自带该模块，如果你的环境上没有pyparsing，请从pyparsing主页下载（译注：推荐用pip install pyparsing）。

3\在你的python代码中，import pyparsing模块。我们推荐采用这样的形式：

import pyparsing as pp

本文中的例子将用pp指代pyparsing。

4\您的脚本会组装一个和BNF定义匹配的parser，parser是抽象基类pp.ParserElement的实例，pp.ParserElement用于描述一般模式。

为输入文件格式构建解析器是一个自下而上的过程。 首先为最小的部分编写解析器，然后将它们组装成越来越大的部分，然后为整个文件提供解析器（这句是从google翻译直接拷贝过来的^\_^）。

5\构建包含要处理的输入文本的Python字符串（类型为str或unicode）。

6\如果parser是p，输入文本是s，则下面的代码将尝试匹配它们：

p.parseString(s)

如果s与p描述的语法匹配，上述代码将返回一个对象，这个对象是类pp.ParseResults的一个实例，代表了匹配到的部分。

如果s与您的parser不匹配，会触发类pp.ParseException的异常。

此异常会标明，输入字符串的哪里没能被匹配到。

.parseString（）方法依次处理输入文本，用您定义的parser匹配文本块。有时，最底层的parser称为token，更高层的被称为模式（pattern）。

您可以将解析操作附加到任何组件parser。 例如，整数的parser可能具有附加的解析操作，该操作将字符串转换为Python int。

7\返回的ParseResults实例中,包含应用程序的信息。 该实例的确切结构取决于您构建解析器的方式。

# 3、一个小而完整的例子

下面这个能“跑”的小例子，只是为了给读者一个大致的印象，知道pyparsing怎么用。

Python标识符的名称，由一个或多个字符构成，其中第一个字符是字母或者下划线（"\_"），后面可以接字母、数字、下划线。在扩展BNF中，我们可以这样写：

first ::= letter | "\_"

letter ::= "a" | "b" | ... "z" | "A" | "B" | ... | "Z"

digit ::= "0" | "1" | ... | "9"

rest ::= first | digit

identifier ::= first rest\*

最后一条规则（production，常见的译法是“产生式”），可以理解为：“标识符由一个first和零个或多个rest组成”。

下面是实现该语法的脚本，并用一些字符串对脚本进行测试。

#!/usr/bin/env python

#================================================================

# trivex: Trivial example

#----------------------------------------------------------------

# - - - - - I m p o r t s

import sys

下一行导入pyparsing模块并将其重命名为pp。

import pyparsing as pp

# - - - - - M a n i f e s t c o n s t a n t s

在下一行中，pp.alphas变量是一个包含所有小写和大写字母的字符串。 pp.Word（）类生成一个解析器，该解析器匹配由其第一个参数定义的字符串; 参数exact = 1告诉解析器（parser）接受该字符串中的一个字符。 first是一个解析器（即ParserElement实例），它只匹配一个字母或一个下划线。

first = pp.Word(pp.alphas+"\_", exact=1)

pp.alphanums变量是一个包含所有字母和所有数字的字符串。 因此，rest模式匹配一个或多个字母，数字或下划线。

rest = pp.Word(pp.alphanums+"\_")

Python“+”运算符被pp.ParserElement重载，以表示序列：即，标识符解析器匹配first解析器匹配的内容，后跟可选的rest解析器匹配的内容。

identifier = first+pp.Optional(rest)

testList = [ # List of test strings

# Valid identifiers

"a", "foo", "\_", "Z04", "\_bride\_of\_mothra",

# Not valid

"", "1", "$\*", "a\_#" ]

# - - - - - m a i n

def main():

"""

"""

for text in testList:

test(text)

# - - - t e s t

def test(s):

'''See if s matches identifier.

'''

print "---Test for '{0}'".format(s)

当您在pp.ParserElement类的实例上调用.parseString（）方法时，它会返回匹配元素的列表，或触发pp.ParseException异常。

try:

result = identifier.parseString(s)

print " Matches: {0}".format(result)

except pp.ParseException as x:

print " No match: {0}".format(str(x))

# - - - - - E p i l o g u e

if \_\_name\_\_ == "\_\_main\_\_":

main()

下面是脚本的输出：

---Test for 'a'

Matches: ['a']

6 pyparsing quick reference New Mexico Tech Computer Center

---Test for 'foo'

Matches: ['f', 'oo']

---Test for '\_'

Matches: ['\_']

---Test for 'Z04'

Matches: ['Z', '04']

---Test for '\_bride\_of\_mothra'

Matches: ['\_', 'bride\_of\_mothra']

---Test for ''

No match: Expected W:(abcd...) (at char 0), (line:1, col:1)

---Test for '1'

No match: Expected W:(abcd...) (at char 0), (line:1, col:1)

---Test for '$\*'

No match: Expected W:(abcd...) (at char 0), (line:1, col:1)

---Test for 'a\_#'

Matches: ['a', '\_']

返回值是pp.ParseResults类的一个实例; 打印时，它显示为匹配字符串的列表。 你会注意到，对于单个字符串，结果列表只有一个元素，而对于多字母字符串，列表有两个元素：第一个字符（first规则匹配的部分）和rest规则匹配到的其余字符。

如果想让结果列表只包含一个元素，需要改动一行代码：

identifier = pp.Combine(first+pp.Optional(rest))

pp.Combine【1】类让pyparsing把参数列表的匹配片段，放到一个单独的结果中。以下两行是修改脚本后的示例：

---Test for '\_bride\_of\_mothra'

Matches: ['\_bride\_of\_mothra']

# 4、如何构造返回的ParseResults

当您的输入与您构建的解析器匹配时，.parseString（）方法返回类ParseResults的实例。

对于复杂的结构，输入中的不同部分，在ParseResult实例中会有不同bits与之对应。 ParseResults实例内部的确切结构，取决于您如何构建顶层解析器。

生成的ParseResult实例，可以通过两种方式访问：

* 作为list。匹配到n个内部组件的解析器（parser），它返回的结果r，可以被当作有n个字段串的list访问。可以用r[n]访问第n个元素；或者，可以用list()函数，将结果转换成真正的list。

>>> import pyparsing as pp

>>> number = pp.Word(pp.nums)

>>> result = number.parseString('17')

>>> print result

['17']

>>> type(result)

<class 'pyparsing.ParseResults'>

>>> result[0]

'17'

>>> list(result)

['17']

>>> numberList = pp.OneOrMore(number)

>>> print numberList.parseString('17 33 88')

['17', '33', '88']

* 作为字典。您可以通过调用其.setResultsName（s）方法将结果名称r附加到解析器（请参见第5.1节“ParserElement：基本解析器构建块”（第11页））。 完成后，您可以从ParseResults实例r中提取匹配的字符串作为“r [s]”。

>>> number = pp.Word(pp.nums).setResultsName('nVache')

>>> result = number.parseString('17')

>>> print result

['17']

>>> result['nVache']

'17

以上是构造解析器的ParseResults实例的一些通用原则。

4.1、用pp.Group()分而治之

和任何复杂点的程序一样，分而治之的原则对构造解析器也是适用的，对任意复杂度的解析器，分而治之（或者称之为逐步细化），能让构造解析器的过程变得更容易。

实践中，顶层的ParseResults不应包含过多元素，比如说，不能超过五个或者七个。如果这一层的元素过多，请查看总体输入，并将其划分成两个或更多的子解析器。然后再重新构建顶层解析器，让它只包含这些片段。如果有必要，将子解析器再划分为更小的解析器，直到每一个解析器，或者由内置的基本函数构成，或者由其他解析器构成。

第5.13节，“Group：将重复的项目组合成一个列表”（第23页）是创建这些抽象级别的基本工具。

* 通常，匹配到多个内容的解析器，它的ParseResults实例会表现得像一个，由被匹配到的字符串组成的列表。例如，如果解析器匹配一个单词列表，它的ParseResults，可以像列表一样被打印（print）出来。使用type()函数，我们能看到列表中每个元素的实际类型，是python的字符串。

>>> word = pp.Word(pp.alphas)

>>> phrase = pp.OneOrMore(word)

>>> result = phrase.parseString('farcical aquatic ceremony')

>>> print result

['farcical', 'aquatic', 'ceremony']

>>> type(result)

<class 'pyparsing.ParseResults'>

>>> type(result[0])

<type 'str'>

* 但是，对解析器的定义包含[pp.Group](http://xn--pp-kn1dt4jz0b.group/" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)（）时，它的pp.ParseResults中的每个元素，表现得像是列表（译注：没看懂这句话的字面意思，不过，看实例代码能明白Group的效果，被Group匹配到的部分，是ParseResults中的一个单独的元素。）。

例如，假设您的程序正在拆解一系列单词，并且您希望以一种方式处理第一个单词，而将另一个单词处理为另一种方式。 这是我们的第一次尝试。

>>> ungrouped = word + phrase

>>> result = ungrouped.parseString('imaginary farcical aquatic ceremony')

>>> print result

['imaginary', 'farcical', 'aquatic', 'ceremony']

这个结果与我们的概念并不完全一致，即解析器是两个事物的序列：单个单词，后跟一系列单词。

[在解析器的定义中增加pp.Group](http://xn--pp-hm4c753coll81s5s6bfea4x.group/" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)（），它将返回与我们的概念匹配的两个事物的序列。

>>> grouped = word + pp.Group(phrase)

>>> result = grouped.parseString('imaginary farcical aquatic ceremony')

>>> print result

['imaginary', ['farcical', 'aquatic', 'ceremony']]

>>> print result[1]

['farcical', 'aquatic', 'ceremony']

>>> type(result[1])

<class 'pyparsing.ParseResults'>

>>> result[1][0]

'farcical'

>>> type(result[1][0])

<type 'str'>

1\包含分组的解析器，由两个部分构成，一个word和一个pp.Group。因此返回的结果就像是一个双元素的列表。

2\第一个元素是实际的字符串，'imaginary'。

3\第二部分是另一个pp.ParseResults实例，其作用类似于字符串列表。

因此对于较大的语法，顶级解析器在匹配时返回的pp.ParseResults实例，通常是混合的多层结构，既包含这种普通字符串，也包含其他pp.ParseResults实例。 下一节将为您提供有关管理这些beast结构的一些建议（译注：beast在这里是俚语么？直接翻译成野兽实在太突兀了）。

4.2、使用结果名称进行结构化

对于仅在特定级别出现一次的解析器，请考虑使用.setResultsName（）将结果名称与该解析器相关联。这样，就可以把ParseResults看成是字典，用名称作为关键字从字典中获取匹配到的文本。

此选项的设计规则：

* 通过名称访问比通过位置访问更robust。您正在处理的结构，可能会随时间而改变。以列表的方式访问结果，底层的结构变了，一个元素对应的位置可能会随之改变。

但是，如果您给结果命名为'swamp Name'，则访问代码result['swamp Name']可能仍然会继续工作，即使后来result中又增添了其他名称。

* 在一个ParseResults中，按位置访问或者按关键字访问（即按结果名称），不可得兼。 如果解析器的某些子元素被命名了，而另一些子元素没有被命名，则所有这些子元素的匹配文本，将在结果中混合在一起。

下面是一个示例，显示当您在同一级别混合位置和命名访问时会发生什么：在骑牛比赛中，总分是骑手的得分和公牛的得分的组合。

>>> rider = pp.Word(pp.alphas).setResultsName('Rider')

>>> bull = pp.Word(pp.alphas).setResultsName('Bull')

>>> score = pp.Word(pp.nums+'.')

>>> line = rider + score + bull + score

>>> result = line.parseString('Mauney 46.5 Asteroid 46')

>>> print result

['Mauney', '46.5', 'Asteroid', '46']

在上面显示的四元素列表中，您可以按名称访问第一个和第三个元素，但只能按位置访问第二个和第四个元素。

一种更合理的方法，是创建由名字和分数组合而成的解析器，然后，再用两个这样的解析器，构造一个更大的解析器。

>>> name = pp.Word(pp.alphas).setResultsName('name')

>>> score = pp.Word(pp.nums+'.').setResultsName('score')

>>> nameScore = pp.Group(name + score)

>>> line = nameScore.setResultsName('Rider') +

nameScore.setResultsName('Bull')

>>> result = line.parseString('Mauney 46.5 Asteroid 46')

>>> result['Rider']['name']

'Mauney'

>>> result['Bull']['score']

'46

* 不要对重复元素使用结果名称。 如果这样做，在ParseResults中用名称只能访问到最后一个元素。

>>> catName = pp.Word(pp.alphas).setResultsName('catName')

>>> catList = pp.OneOrMore(catName)

>>> result = catList.parseString('Sandy Mocha Bits')

>>> result['catName']

'Bits'

>>> list(result)

['Sandy', 'Mocha', 'Bits']

[更好的方法是将整个名称包装在pp.Group](http://xn--pp-gy2c64qymbl6ikxem7f0uzxtas4bq5dc65at45awzk2z1d.group/" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)（）中，然后将结果名称应用于该Group。

>> owner = pp.Word(pp.alphas).setResultsName('owner')

>>> catList = pp.Group(pp.OneOrMore(catName)).setResultsName('cats')

>>> line = owner + catList

>>> result = line.parseString('Carol Sandy Mocha Bits')

>>> result['owner']

'Carol'

>>> print result['cats']

['Sandy', 'Mocha', 'Bits']

# 5、类

以下是pyparsing模块中定义的类。

## 5.1、ParserElement：基本的解析器构建单元

定义：解析器是pp.ParserElement的某个子类的实例。

在用小的片段构建大型语法的过程中，为每个片段定义一个解析器，然后将这些片段滚雪球一般逐层拼接，直到能匹配整个输入的解析器被构建出来。

pyparsing的内置类，如pp.And，pp.Or和pp.OneOrMore，可以用于将解析器组装成更大的配置。 这些类的构造函数都返回一个解析器，并且其中许多接受一个或多个解析器作为参数。

例如，如果解析器p所描述的语法元素是可选的，则pp.optional(p)返回一个新的解析器 -- 即pp.ParserElement子类的另一个实例 -- 如果在输入中的对应位置出现了模式p，p就会被匹配到，如果没出现，则不执行任何操作。

以下是解析器实例p上可用的方法，p是pp.ParserElement的子类。

**p.addParseAction(f1, f2, ...)**

返回附加了一个或多个parse action的p的副本。 有关parse action的讨论，请参阅下面的p.setParseAction（）方法。

**p.copy()**

返回p的副本。

**p.ignore(q)**

此方法修改p，以便忽略与模式q匹配的任何数量的文本。 这是让解析器忽略注释的有用方法。

>>> number = pp.Word(pp.nums)

>>> name = pp.Word(pp.alphas).ignore(number)

>>> print name.parseString('23 84 98305478 McTeagle')

['McTeagle']

**p.leaveWhitespace()**

让p在匹配输入文本之前不要跳过空格，这个方法返回p。在包含多个片段的解析器上，使用leaveWhitespace函数,会抑制解析器跳过每个片段前的空格。 这是一个例子：

>>> word = pp.Word(pp.alphas)

>>> num = pp.Word(pp.nums)

>>> wn = (word + num).leaveWhitespace()

>>> nwn = num + wn

>>> print nwn.parseString('23xy47')

['23', 'xy', '47']

>>> print nwn.parseString('23 xy47')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/lib/python2.7/site-packages/pyparsing.py", line 1032,

in parseString

raise exc

pyparsing.ParseException: Expected W:(abcd...) (at char 2), (line:1,

col:3)

>>> print nwn.parseString('23xy 47')

pyparsing.ParseException: Expected W:(0123...) (at char 4), (line:1,

col:5)

注意

为了节省空间，在后续示例中，我们将省略除最后一行之外的所有“Traceback”行。

您会注意到，即使num解析器没有跳过空白，仍然不允许对字符串'47'使用空格，因为wn解析器禁用了自动空格跳过

**p.parseFile(f, parseAll=False)**

尝试将文件的内容与解析器p匹配。 参数f可以是文件名或类文件对象。

如果文件的全部内容与p不匹配，则除非您传递参数parseAll = True，否则不会将其视为错误

**p.parseString(s, parseAll=False)**

尝试将字符串s与解析器p匹配。 如果匹配，则返回第5.26节“ParseResults：从匹配返回的结果”的实例（第28页）。 如果没有匹配，则会引发pp.ParseException。

默认情况下，如果s的整体与p不匹配，则不会将其视为错误。 如果要确保所有s匹配p，请传递关键字参数parseAll = True。

**p.scanString(s)**

搜索字符串s以查找与p匹配的区域。 此方法是一个迭代器，它生成一系列元组（r，start，end），其中r是表示匹配部分的pp.ParseResults实例，start和end是匹配的文本在s中的起始位置和结束位置。

>>> name = pp.Word(pp.alphas)

>>> text = "\*\*\*\* Farcical aquatic ceremony"

>>> for result, start, end in name.scanString(text):

... print "Found {0} at [{1}:{2}]".format(result, start, end)

...

Found ['Farcical'] at [5:13]

Found ['aquatic'] at [14:21]

Found ['ceremony'] at [23:31]

**p.setBreak()**

使用解析器时，设置断点，调出Python调试器pdb。

p.setFailAction(f)

此方法修改p，以便在无法解析时调用函数f。 该方法返回p。

以下是失败操作的调用顺序：

f(s, loc, expr, err)

S

输入的字符串

Loc

解析失败的输入中的位置，作为从0开始计数的偏移量。

Expr

失败的解析器的名称。

Err

解析器触发的异常实例。

这是一个例子。

>>> def oops(s, loc, expr, err):

... print ("s={0!r} loc={1!r} expr={2!r}\nerr={3!r}".format(

... s, loc, expr, err))

...

>>> fail = pp.NoMatch().setName('fail-parser').setFailAction(oops)

>>> r = fail.parseString("None shall pass!")

s='None shall pass!' loc=0 expr=fail-parser

err=Expected fail-parser (at char 0), (line:1, col:1)

pyparsing.ParseException: Expected fail-parser (at char 0), (line:1,

col:1)

**p.setName(name)**

出于调试的目的，给解析器附加一个名字。参数是一个字符串。这个方法返回p。

>>> print pp.Word(pp.nums)

W:(0123...)

>>> count = pp.Word(pp.nums).setName('count-parser')

>>> print count

count-parser

>>> count.parseString('FAIL')

pyparsing.ParseException: Expected count-parser (at char 0), (line:1,

col:1)

在上面的示例中，如果将解析器转换为字符串，则会得到它的一般描述：字符串“W：（0123 ...）”告诉您它是一个Word解析器并显示前面的几个字符。一旦为其附加了名称，解析器的字符串形式就是该名称。 请注意，当解析失败时，错误消息会通过命名失败的解析器来标识它所期望的内容。

**p.setParseAction(f1, f2, ...)**

此方法返回附带一个或多个解析操作的p的副本。 当解析器匹配输入时，它会按指定的顺序调用每个函数fi。 解析操作的调用序列可以是以下四种原型中的任何一种：

f()

f(toks)

f(loc, toks)

f(s, loc, toks)

这些是action函数可能收到的参数，具体取决于它接受的参数数量：

s

被解析的字符串。如果字符串中含有制表符（Tab），请参考在线参考手册，以获取有关制表符扩展及其对列位置影响的说明。

loc

匹配子字符串的位置作为偏移量（索引，从0开始计数）。

toks

包含匹配结果的pp.ParseResults实例。

解析操作可以修改传入的toks参数，再将修改后的list返回，达到修改解析结果的效果。如果返回None，则解析结果维持原样。一下是包含两个解析操作的示例。

>>> name = pp.Word(pp.alphas)

>>> def a1():

... print "In a1"

...

>>> def a2(s, loc, toks):

... print "In a2: s={0!r} loc={1!r} toks={2!r}".format(

... s, loc, toks)

... return ['CENSORED']

...

>>> newName = name.setParseAction(a1, a2)

>>> r = newName.parseString('Gambolputty')

In a1

In a2: s='Gambolputty' loc=0 toks=(['Gambolputty'], {})

>>> print r

['CENSORED']

译注：

[Pyparsing setParseAction function is getting no arguments](https://stackoverflow.com/questions/10177276/pyparsing-setparseaction-function-is-getting-no-arguments" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

--这个问题里面，pyparsing的作者Paul建议parseAction函数应该自己处理函数逻辑中的异常，因为如果抛出到pyparsing的解析过程中，提示信息会让人困惑，不知异常是从哪来的。

然后，举了个应该抛出异常的例子：

However, there are sometimes when you *want* a parse action to raise an exception, such as when you perform some semantic validation, in which case your code should raise a pyparsing ParseException.

另，Paul在2004年的文章中也提到了parseAction：

Parse actions can be used to convert values from strings to other data types (ints, floats, booleans, etc.). But be careful not to include converted data within aCombineobject.

**p.setResultsName(name)**

解析器的.parseString()函数，会把匹配到的文本保存到ParseResult实例中，setResultName方法可以给匹配到的文本命名。命名之后，ParseResult会像Python的dict一样，可以用名字从ParseResult中获取数据。

**p.setWhitespaceChars(s)**

对于解析器p，将其空白定义更改为字符串s中的字符

**p.suppress()**

此方法返回p修改过的副本，以便它不会将匹配的文本添加到ParseResult。 这对于省略标点符号很有用。 另请参见第5.32节“抑制：忽略结果中的匹配文本”（第32页）。

>>> name = pp.Word(pp.alphas)

>>> lb = pp.Literal('[')

>>> rb = pp.Literal(']')

>>> pat1 = lb + name + rb

>>> print pat1.parseString('[hosepipe]')

['[', 'hosepipe', ']']

>>> pat2 = lb.suppress() + name + rb.suppress()

>>> print pat2.parseString('[hosepipe]')

['hosepipe']

译注：

stackoverflow上有个实际的例子：

[pyparsing example](https://stackoverflow.com/questions/8507694/pyparsing-example" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)

It is my first attempt to use pyparsing and I'd like to ask how to filter this sample line:

survey = '''GPS,PN1,LA52.125133215643,LN21.031048525561,EL116.898812'''

to get output like: 1,52.125133215643,21.031048525561,116.898812

In general I have problem with understanding pyparsing logic so any help with this example will be appreciated. Thanks

这是jcollado的回答，里面用到了Supress（大写开头的Supress是pyparsing的函数，小写开头的supress是某个parse的函数）：

You could start with something like this:

from pyparsing import \*

survey = '''GPS,PN1,LA52.125133215643,LN21.031048525561,EL116.898812'''

number = Word(nums+'.').setParseAction(lambda t: float(t[0]))

separator = Suppress(',')

latitude = Suppress('LA') + number

longitude = Suppress('LN') + number

elevation = Suppress('EL') + number

line = (Suppress('GPS,PN1,')

+ latitude

+ separator

+ longitude

+ separator

+ elevation)

print line.parseString(survey)

The output of the script is:

[52.125133215643, 21.031048525561, 116.898812]

此外，这些普通的Python运算符在ParserElement中被重载：

p1+p2

相当于“pp.And（p1，p2）”。

p \* n

对于解析器p和整数n，结果是匹配n次重复p的解析器。 您可以按任意顺序给出操作数：例如，“3 \* p”与“p \* 3”相同。

p1 | p2

相当于“pp.MatchFirst（p1，p2）”。

p1 ^ p2

相当于“pp.Or（p1，p2）”。

p1 & p2

相当于“pp.Each（p1，p2）”。

~ p

相当于“pp.NotAny（p）”。

pp.ParserElement类还支持一种静态方法：

pp.ParserElement.setDefaultWhitespaceChars(s)

此静态方法将空白的定义更改为字符串s中的字符。 调用此方法会对任何pp.ParserElement子类的所有后续实例化产生此影响。

>>> blanks = ' \t-=\*#^'

>>> pp.ParserElement.setDefaultWhitespaceChars(blanks)

>>> text = ' \t-=\*#^silly ##\*=---\t walks--'

>>> nameList = pp.OneOrMore(pp.Word(pp.alphas))

>>> print nameList.parseString(text)

['silly', 'walks']

## 5.2、And：序列

pp.And([expr, ...])

它的参数，是一系列的ParseExpress实例。生成的解析器将按照参数出现的次序，依次匹配这些表达式。也可以使用“+”号来实现同样的效果。一些例子：

>>> letterDigit = pp.And([pp.Word(pp.alphas, exact=1),

... pp.Word(pp.nums, exact=1)])

>>> print letterDigit.parseString('x5')

['x', '5']

>>> digitsLetters = pp.Word(pp.nums) + pp.Word(pp.alphas)

>>> print digitsLetters.parseString('23skiddoo')

['23', 'skiddoo']

>>>

## 5.3、CaselessKeyword：不区分大小写的关键字匹配

pp.CaselessKeyword(matchString, identChars=I)

这是第5.14节的变体，“关键字：匹配与指定上下文不相邻的文字字符串”（第23页），不区分大小写。

## 5.4、CaselessLiteral：不区分大小写的字符串匹配

pp.CaselessLiteral(matchString)

参数是要匹配的文字字符串。 生成的解析器匹配该字符串，但在匹配的过程中，参数中的大小写会被忽略。

匹配的值将始终与matchString参数具有相同的大小写，而不是匹配文本的大小写。

>>> ni=pp.CaselessLiteral('Ni')

>>> print ni.parseString('Ni')

['Ni']

>>> print ni.parseString('NI')

['Ni']

>>> print ni.parseString('nI')

['Ni']

>>> print ni.parseString('ni')

['Ni']

## 5.5、CharsNotIn：匹配不在给定集合中的字符

pp.CharsNotIn(notChars, min=1, max=0, exact=0)

notChars参数可以包含一个或多个字符，解析器会匹配未在notChars中出现的字符。min参数指定被匹配字符串的最小长度，max参数指定被匹配字符串的最大长度。 如果要精确匹配N个字符串，请使用exact参数。

>>> nonDigits = pp.CharsNotIn(pp.nums)

>>> print nonDigits.parseString('zoot86')

['zoot']

>>> nonDigits = pp.CharsNotIn(pp.nums, min=5)

>>> print(nonDigits.parseString('zoot86'))

pyparsing.ParseException: Expected !W:(0123...) (at char 4), (line:1, col:5)

（本例为译者所加，可以看到，如果被匹配的字符串长度不足min所指定的数值，会抛出异常）

>>> fourNonDigits = pp.CharsNotIn(pp.nums, exact=4)

>>> print fourNonDigits.parseString('a$\_/#')

['a$\_/']

## 5.6、Combine：组件融合

pp.Combine(parser, joinString='', adjacent=True)

对包含几个片段的解析器，Combine类可以改变解析器的行为，让返回的ParseResults实例中只包含一个字符串，该字符串由上述几个片段拼接而成。Combine的返回结果，是另一个ParseElement实例，它与parser参数具有相同的语法，但是匹配结果拼接起来。

Parser

解析器，ParserElement实例。

joinString

匹配文本在结果中连接时的连接字符串。

Adjacent

在默认情况下，adjacent = True，与解析器的组件匹配的文本必须是相邻的。 如果传递adjacent = False，则结果将匹配包含组件的文本解析器，即使它们被其他文本分开。

>>> hiwayPieces = pp.Word(pp.alphas) + pp.Word(pp.nums)

>>> print hiwayPieces.parseString('I25')

['I', '25']

>>> hiway = pp.Combine(hiwayPieces)

>>> print hiway.parseString('I25')

['I25']

>>> print hiway.parseString('US380')

['US380']

在上面的例子中，hiwayPieces匹配一个或多个字母（pp.Word（pp.alphas）），后跟一个或多个数字（pp.Word（pp.nums））。 因为它有两个组件，hiwayPieces解析器的匹配结果是包含两个字符串的列表。hiway解析器返回一个包含一个字符串的列表，这是匹配的片段的串联。

>>> myway = pp.Combine(hiwayPieces, joinString='\*', adjacent=False)

>>> print myway.parseString('I25')

['I\*25']

>>> print myway.parseString('Interstate 25')

['Interstate\*25']

>>> print hiway.parseString('Interstate 25')

pyparsing.ParseException: Expected W:(0123...) (at char 10), (line:1,

col:11)

## 5.7、Dict：解析表格(文字待整理)

pp.Dict(pattern)

Dict类是一种高度专业化的模式，用于从按行和列排列的文本中提取数据，其中第一列包含其余列的标签。 pattern参数必须是描述两级结构的解析器，例如组中的组。 可以使用其他类似组的模式，例如delimitedList（）函数。

构造函数返回一个解析器，其中.parseString（）方法将像大多数解析器一样返回一个ParseResults实例; 但是，在这种情况下，ParseResults实例可以像一个字典一样keys是行标签，每个相关值是该行中其他项的列表。

一个例子：

#!/usr/bin/env python

#================================================================

# dicter: Example of pyparsing.Dict pattern

#----------------------------------------------------------------

import pyparsing as pp

data = "cat Sandy Mocha Java|bird finch verdin siskin"

rowPat = pp.OneOrMore(pp.Word(pp.alphas))

bigPat = pp.Dict(pp.delimitedList(pp.Group(rowPat), "|"))

result = bigPat.parseString(data)

for rowKey in result.keys():

print "result['{0}']={1}".format(rowKey, result[rowKey])

上述代码的输出:

result['bird']=['finch', 'verdin', 'siskin']

result['cat']=['Sandy', 'Mocha', 'Java']

## 5.8、Each：组件可以按任意顺序出现

pp.Each([p0, p1, ...])

此类返回与给定的一组片段匹配的ParserElement，但这些片段可以按任何顺序出现。 您也可以使用等价形式来构造 -- 用“＆”运算符来连接组件 -- 即“Each（[p0，p1，p2，...]）== p0＆p1＆p2＆...“。 下面是一个示例：一个需要字母和一串数字的模式，但它们可以按任意顺序出现。

>>> num=pp.Word(pp.nums)

>>> name=pp.Word(pp.alphas)

>>> nameNum = num & name

>>> print nameNum.parseString('Henry8')

['Henry', '8']

>>> print nameNum.parseString('16Christine')

['16', 'Christine']

## 5.9、Empty：匹配空内容

pp.Empty()

构造函数返回一个始终匹配的解析器，并且不消费任何输入。 它可以在需要时候，当成解析器占位符来使用，它不会匹配任何内容。

>>> e=pp.Empty()

>>> print e.parseString('')

[]

>>> print e.parseString('shrubber')

[]

>>> print e.parseString('shrubber', parseAll=True)

pyparsing.ParseException: Expected end of text (at char 0), (line:1, col:1)

## 5.10、FollowedBy：添加前缀约束

pp.FollowedBy(parser)

这个类用来指定前向约束条件，也就是说输入中必须包含某些内容，但是又不能被匹配到，就可以使用FollowBy。一个例子：

>>> name = pp.Word(pp.alphas)

>>> oneOrTwo = pp.Word('12', exact=1)

>>> number = pp.Word(pp.nums)

>>> pat = name + pp.FollowedBy(oneOrTwo) + number

>>> print pat.parseString('Robin144')

['Robin', '144']

>>> print pat.parseString('Robin88')

pyparsing.ParseException: Expected W:(12) (at char 5), (line:1, col:6)

name模式匹配一个或多个字母; oneOrTwo模式匹配'1'或'2'; number模式匹配一个或多个数字。 表达“pp.FollowedBy(oneOrTwo)”需要name后面的内容匹配oneOrTwo模式，但输入没有越过这个位置。

因此，number模式匹配一个或多个数字，包括名称后面的“1”或“2”。 在'Robin88'示例中，匹配失败，因为'Robin'之后的字符既不是'1'也不是'2'。

## 5.11、Forward：解析器占位符

pp.Forward

这个解释起来有点复杂。

某些场景中，BNF可能无法描述什么是正确的内容，什么不是。例如，考虑这样一种模式“相同数字的两个副本，一个接一个”。“33”和“99”都符合这个模式。

编写匹配一位数字的模式不难：“pp.Word(pp.nums，exact = 1)”，挨着的两个数字：“pp.Word(pp.nums，exact = 1)+ pp.Word(pp.nums，exact = 1)”。

但是，虽然该模式能匹配'33'，它也能匹配'37'。 那么你的脚本如何指定两个部分匹配相同的数字呢？

在pyparsing中，我们使用Forward实例来达成这样的目标。Forward实例基本上是一个空的占位符，在执行解析期的过程中会添加一个模式。

让我们看一个演示使用Forward模式的完整脚本。 这个例子，将把我们带回计算机的“远古时代”，解析FORTRAN语言早期版本的一个特性：Hollerith字符串常量。

Hollerith常量是表示字符串的一种方式。 它由一个计数，后跟字母“H”，后跟计数指定的字符数组成。 以下是两个示例，其中包含Python等价物：

1HX 'X'

10H0123456789 '0123456789'

我们编写的模式，可以匹配大写或小写的'H'。

这是完整的脚本。 我们从通常的准备步骤开始：import，一些测试字符串，main，以及运行每个测试的函数。

#!/usr/bin/env python

#================================================================

# hollerith: Demonstrate Forward class

#----------------------------------------------------------------

import sys

import pyparsing as pp

# - - - - - M a n i f e s t c o n s t a n t s

TEST\_STRINGS = [ '1HX', '2h$#', '10H0123456789', '999Hoops']

# - - - - - m a i n

def main():

holler = hollerith()

for text in TEST\_STRINGS:

test(holler, text)

# - - - t e s t

def test(pat, text):

'''Test to see if text matches parser (pat).

'''

print "--- Test for '{0}'".format(text)

try:

result = pat.parseString(text)

print " Matches: '{0}'".format(result[0])

except pp.ParseException as x:

print " No match: '{0}'".format(str(x))

接下来我们定义函数hollerith（），它返回Hollerith字符串的解析器。

# - - - h o l l e r i t h

def hollerith():

'''Returns a parser for a FORTRAN Hollerith character constant.

'''

首先，我们定义一个与字符数匹配的解析器intExpr。 它有一个解析操作，可以将数字从字符形式转换为Python int。 lambda表达式定义了一个无名函数，它接受一个标记列表并将第一个标记转换为int。

#--

# Define a recognizer for the character count.

#--

intExpr = pp.Word(pp.nums).setParseAction(lambda t: int(t[0]))

接下来，我们创建一个空的Forward解析器作为匹配“H”和后续字符的逻辑占位符。

#--

# Allocate a placeholder for the rest of the parsing logic.

#--

stringExpr = pp.Forward()

接下来，我们定义一个闭包，它将作为第二个解析动作添加到intExpr中。 请注意，我们在函数中定义了一个函数。 在定义函数后，countsParseAction函数将保留对外部名称（stringExpr，在外部函数的作用域中定义）的访问。

#--

# Define a closure that transfers the character count from

# the intExpr to the stringExpr.

#--

def countedParseAction(toks):

'''Closure to define the content of stringExpr.

'''

参数是intExpr识别的token列表; 由于intExpr的解析操作，token列表已经把计数器转换为int，并保存到列表的第一个元素。

n = toks[0]

contents解析器会匹配恰好n个字符。我们使用第5.5节讲述的功能，“CharsNotIn：匹配不在给定集合中的字符”，来实现匹配n个字符的功能，将待排除的字符集指定为空字符，则能够匹配任何字符。顺便说一句，这不适用于n == 0的情况，但'0H'不是有效的Hollerith文字。 在这种情况下，更健壮的实现会触发pp.ParseException。

#--

# Create a parser for any (n) characters.

#--

contents = pp.CharsNotIn('', exact=n)

下一行将最终模式插入到占位符解析器中。一个大小写无关的H，后面跟着contents模式。 '<<'运算符在Forward类中被重载，可以执行这样的操作：对于任何Forward F和任何解析器p，表达式“F << p”修改F以使其匹配模式p。

#--

# Store a recognizer for 'H' + contents into stringExpr.

#--

stringExpr << (pp.Suppress(pp.CaselessLiteral('H')) + contents)

解析操作可以选择修改已识别的token，但我们不需要这样做，因此我们返回None表示tokens保持不变。

return None

countingParseAction闭包到此结束。现在我们回到hollerith（）的范围。下一行添加闭包作为intExpr解析器的第二个解析操作。

#--

# Add the above closure as a parse action for intExpr.

#--

intExpr.addParseAction(countedParseAction)

现在我们准备返回完成的hollerith解析器：intExpr识别count并且stringExpr识别'H'和字符串内容。 当我们返回它时，它仍然只是一个空的Forward，但它会在它被要求解析之前填充。

#--

# Return the completed pattern.

#--

return (pp.Suppress(intExpr) + stringExpr)

# - - - - - E p i l o g u e

if \_\_name\_\_ == "\_\_main\_\_":

main()

这是脚本的输出。 请注意，最后一次测试失败，因为'999H'后面没有999个字符。

--- Test for '1HX'

Matches: 'X'

--- Test for '2h$#'

Matches: '$#'

--- Test for '10H0123456789'

Matches: '0123456789'

--- Test for '999Hoops'

No match: 'Expected !W:() (at char 8), (line:1, col:9)'

## 5.12、GoToColumn：前进到行中的指定位置【TODO：完全采用谷歌翻的文字，可能需要改进可读性】

pp.GoToColumn(colNo)

此类返回一个解析器，使得输入位置前进到列号colNo，其中列号从1开始计数。此解析器匹配的值是当前位置和位置colNo之间的字符串。 如果当前位置超过列colNo，则会出错。

>>> pat = pp.Word(pp.alphas, max=4)+pp.GoToColumn(5)+pp.Word(pp.nums)

>>> print pat.parseString('ab@@123')

['ab', '@@', '123']

>>> print pat.parseString('wxyz987')

['wxyz', '', '987']

>>> print pat.parseString('ab 123')

['ab', '', '123']

在这个例子中，pat是一个包含三个部分的解析器。 第一部分匹配一到四个字母。 第二部分跳到第5列。第三部分匹配一个或多个数字。 在第一个测试中，GoToColumn解析器返回'@@'，因为它是字母和列5之间的文本。在第二个测试中，该解析器返回空字符串，因为'wxyz'和'987'之间没有字符。 在第三个示例中，与GoToColumn匹配的部分为空，因为在token之间忽略了空格。

## 5.13、Group：将重复的项目分组到列表中

pp.Group(parser)

这个类将匹配到的结果以列表（list）的形式返回。当作参数传入的parser，是解析器，可以匹配类似零个或多个token，或者由分隔符隔开的token列表。

>>> lb = pp.Literal('{')

>>> rb = pp.Literal('}')

>>> wordList = pp.OneOrMore(pp.Word(pp.alphas))

>>> pat1 = lb + wordList + rb

>>> print pat1.parseString('{ant bee crow}')

['{', 'ant', 'bee', 'crow', '}']

>>> pat2 = lb + pp.Group(wordList) + rb

>>> print pat2.parseString('{ant bee crow}')

['{', ['ant', 'bee', 'crow'], '}']

在上面的例子中，pat1和pat2都匹配大括号括起来的一系列单词。执行pat1，匹配的结果是五个元素，三个单词和两个括号。执行pat2，匹配的结果是三个元素，OneOrMore解析器匹配到的一系列单词，右括号。

## 5.14、Keyword：匹配与指定上下文不相邻的文字字符串

pp.Keyword(matchString, identChars=I, caseless=False)

matchString参数是一个文字字符串。 生成的解析器将匹配输入中的确切文本。 但是，与Literal类不同，下一个输入字符不能是I中的字符之一.identChars参数的默认值是包含字母和数字以及下划线（“\_”）和美元符号（“$”）的字符串【译注：谷歌翻译的句子，一字未改，可读性已经很好了】。

如果提供keyword参数caseless = True，则匹配将不区分大小写。

示例：

>>> key=pp.Keyword('Sir')

>>> print key.parseString('Sir Robin')

['Sir']

>>> print key.parseString('Sirrah')

pyparsing.ParseException: Expected "Sir" (at char 0), (line:1, col:1)

## 5.15、LineEnd：匹配行尾

pp.LineEnd()

如果当前位置位于行的末尾或字符串的结尾，则此类的实例匹配。 如果它在一行的末尾匹配，则在结果中返回换行符（'\ n'）。

>>> anb = pp.Word(pp.alphas) + pp.LineEnd() + pp.Word(pp.alphas)

>>> print anb.parseString('a\nb', parseAll=True)

['a', '\n', 'b']

>>> an = name + pp.LineEnd()

>>> print an.parseString('Dibley\n')

['Dibley', '\n']

在下一个示例中，请注意字符串的末尾与pp.LineEnd（）匹配，但在这种情况下，结果中不会添加任何值。

>>> print an.parseString('Basingstoke')

['Basingstoke']

## 5.16、LineStart：匹配行的开始

pp.LineStart()

如果当前位置位于行的开头，则能够被该类的实例匹配; 也就是说，如果当前位置是文本的开头或者在换行符之前。 它不会改变当前位置或为结果贡献任何内容。

这里有些例子。 第一个模式匹配行开头的名称。

>>> name = pp.Word(pp.alphas)

>>> sw = pp.LineStart() + name

>>> print sw.parseString('Dinsdale')

['Dinsdale']

这里的ansb模式匹配一个名称，后跟换行符后跟另一个名称。 请注意，虽然有四个组件，但结果中只有三个字符串; pp.LineStart（）不提供结果字符串。

>>> ansb = name + pp.LineEnd() + pp.LineStart() + name

>>> print ansb.parseString('Spiny\nNorman')

['Spiny', '\n', 'Norman']

以下是pp.LineStart（）无法匹配的示例。

>>> asb = name + pp.LineStart() + name

>>> asb.parseString('Bath Wells')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/lib/python2.7/site-packages/pyparsing.py", line 1032, in

parseString

raise exc

pyparsing.ParseException: Expected start of line (at char 4), (line:1,

col:5)

## 5.17、Literal：匹配特定字符串

pp.Literal(text)

匹配文本参数的确切字符。 这里有些例子。

>>> name = pp.Word(pp.alphas)

>>> pat = pp.Literal('x') + name

>>> print pat.parseString('xyz')

['x', 'yz']

>>> print pat.parseString('abc')

pyparsing.ParseException: Expected "x" (at char 0), (line:1, col:1)

## 5.18、MatchFirst：按给定顺序尝试多个匹配

pp.MatchFirst(parserList)

如果要尝试匹配两个或更多不同的解析器，请使用此类的实例，但是要指定测试的顺序。 parserList参数是解析器列表。

一个典型的使用方法是将文本与一组字符串相匹配，其中一些字符串是其他字符串的子字符串。 例如，假设您的输入文本有两个不同的命令名称CATCH和CAT：

你应该先测试一下CATCH。 如果您首先测试CAT，它将匹配CATCH的前三个字符，这可能不是您想要的。

>>> name = pp.Word(pp.alphas)

>>> keySet = pp.MatchFirst([pp.Literal('CATCH'), pp.Literal('CAT')])

>>> keyName = keySet + name

>>> print keyName.parseString('CATCH bullatfour', parseAll=True)

['CATCH', 'bullatfour']

>>> print keyName.parseString('CAT gotchertung')

['CAT', 'gotchertung']

您还可以通过将组件解析器与“|”运算符组合来获得此类的效果。 上面例子中keySet的定义也可以这样做：

>>> keySet = pp.Literal('CATCH') | pp.Literal('CAT')

## 5.19、NoMatch：一个永不匹配的解析器

pp.NoMatch()

此解析器将始终引发pp.ParseException。

>>> fail = pp.Literal('Go') + pp.NoMatch()

>>> fail.parseString('Go')

pyparsing.ParseException: Unmatchable token (at char 2), (line:1, col:3)

## 5.20、NotAny：General lookahead condition

pp.NotAny(parser)

NotAny的作用类似于第5.10节“FollowedBy：添加前缀约束”（第19页），NotAny会查看当前位置（position）的文本，是否不匹配parser参数，但是不会把position向前移动。当且仅当，parser模式在当前位置无法匹配时，NotAny才成功。无论parser是否能匹配到，当前位置都不会向前移动。

在下面的示例中，模式匹配一系列字母，后跟一系列数字，前提是第一个不是“0”。

>>> name = pp.Word(pp.alphas)

>>> num = pp.Word(pp.nums)

>>> nameNum = name + pp.NotAny(pp.Literal('0')) + num

>>> print nameNum.parseString('Woody37')

['Woody', '37']

>>> print nameNum.parseString('Tinny006')

pyparsing.ParseException: Found unwanted token, "0" (at char 5), (line:1,

col:6)

## 5.21、OneOrMore：重复一次或多次模式

pp.OneOrMore(parser)

这个类的实例，可以在输入中匹配到一个或多个，由解析的语法描述的模式。

>>> name = pp.Word(pp.alphas)

>>> nameList = pp.OneOrMore(name)

>>> print nameList.parseString('You are in great peril', parseAll=True)

['You', 'are', 'in', 'great', 'peril']

>>> hiway = pp.Combine(pp.Word(pp.alphas) + pp.Word(pp.nums))

>>> hiwayList = pp.OneOrMore(hiway)

>>> print hiwayList.parseString("I25 US380 NM18", parseAll=True)

['I25', 'US380', 'NM18']

## 5.22、Optional：匹配可选模式

pp.Optional(parser, default=D)

当语法元素是可选的时，请使用此模式。 parser参数是可选模式的解析器。 默认情况下，如果模式不存在，则不会向ParseResult添加任何内容; 如果您想提供要在这种情况下添加的内容，请将其作为默认关键字选项提供。

>>> letter = pp.Word(pp.alphas, exact=1)

>>> number = pp.Word(pp.nums)

>>> chapterNo = number + pp.Optional(letter)

>>> print chapterNo.parseString('23')

['23']

>>> print chapterNo.parseString('23c')

['23', 'c']

>>> chapterX = number + pp.Optional(letter, default='\*')

>>> print chapterX.parseString('23')

['23', '\*']

## 5.23、Or：解析一组备选方案中的一个

pp.Or(parserList)

Or的实例匹配给定的一组解析器; 参数是包含这些解析器的列表。 如果有多个解析器都能匹配，则匹配最长的文本字符串。

>>> name = pp.Word(pp.alphas)

>>> number = pp.Word(pp.nums)

>>> nameNoOrBang = pp.Or([name, number, pp.Literal('!')])

>>> print nameNoOrBang.parseString('Brian')

['Brian']

>>> print nameNoOrBang.parseString('73')

['73']

>>> print nameNoOrBang.parseString('!')

['!']

>>> several = pp.OneOrMore(nameNoOrBang)

>>> print several.parseString('18 years of total silence!')

['18', 'years', 'of', 'total', 'silence', '!']

您还可以使用“^”运算符构造一组备选方案。 该行等同于上面示例的第三行：

>>> nameNoOrBang = name ^ number ^ pp.Literal('!')

## 5.24、ParseException

这是解析失败时抛出的异常。 这些属性在实例上可用：

lineno

解析失败的行号，从1开始计算。

col

解析失败的列号，从1开始计数。

line

解析失败的行的文本。

>>> fail = pp.NoMatch()

>>> try:

... print fail.parseString('Is that an ocarina?')

... except pp.ParseException as x:

... print "Line {e.lineno}, column {e.col}:\n'{e.line}'".format(e=x)

...

Line 1, column 1:

'Is that an ocarina?'

## 5.25、ParseFatalException：带我离开这里！

如果您的某个解析器发现它无法继续，则可以引发此异常以立即终止解析。

## 5.26、ParseResults：从匹配返回的结果

所有解析器都返回ParseResults的实例。ParseResults可以表现得像一个字符串列表，其中的每个字符串都是对应的解析器匹配的结果。也可以表现得像字典一样，不过需要先在解析器上调用.setResultsName()函数，设置匹配结果在ParseResults中的名字。

对于实例R，操作包括：

R[index]

要检索其中一个匹配的token，可以将实例视为列表。 您还可以在ParseResult上执行常规列表操作，例如替换或删除其中一个值。

>>> nameList = pp.OneOrMore(pp.Word(pp.alphas))

>>> r = nameList.parseString('tungsten carbide drills')

>>> len(r)

3

>>> r[0]

'tungsten'

>>> r[2]

'drills'

>>> r[1] = 'fluoride'

>>> print r

['tungsten', 'fluoride', 'drills']

>>> del r[1]

>>> print r

['tungsten', 'drills']

如果解析器的任一组件被指定了名称，则可以将ParseResults实例用作字典：键是名称，对应的值是与该组件匹配的字符串

>>> firstName = pp.Word(pp.alphas).setResultsName('first')

>>> lastName = pp.Word(pp.alphas).setResultsName('last')

>>> fullName = firstName + lastName

>>> r = fullName.parseString('Doug Piranha')

>>> r['last']

'Piranha'

>>> r['first']

'Doug

以下是ParseResults实例上可用的方法。

R.asDict()

此方法将R的命名项作为普通Python dict返回。 继续上面的例子：

>>> r.asDict()

{'last': 'Piranha', 'first': 'Doug'}

R.asList()

此方法将R作为普通Python列表返回。

>>> r.asList()

['Doug', 'Piranha']

R.copy()

返回R的副本。

.get(key, defaultValue=None)

与标准Python dict类型的.get（）方法类似：如果ParseResult没有名为key的组件，则返回defaultValue。

>>> r.get('first', 'Unknown')

'Doug'

>>> r.get('middle', 'Unknown')

'Unknown

.insert(where, what)

与Python列表类型的.insert（）方法一样，此方法将在字符串列表中的位置之前插入字符串的值。

>>> r.insert(1, 'Bubbles')

>>> print r

['Doug', 'Bubbles', 'Piranha']

1. items()

此方法的工作方式类似于Python的dict类型的.items（）方法，返回元组列表（键，值）。

>>> r.items()

[('last', 'Piranha'), ('first', 'Doug')]

R.keys()

返回命名结果的键列表。 继续上述Piranha的例子：

>>> r.keys()

['last', 'first']

## 5.27、QuotedString：匹配分隔的字符串

pp.QuotedString(quoteChar, escChar=None, multiline=False,

unquoteResults=True, endQuoteChar=None)

QuotedString的实例，可以匹配被单个字符或多个字符分隔的字符串。

quoteChar

此字符串参数定义开始分隔符，除非您传递endQuoteChar参数，否则还定义结束分隔符。 该值可能包含多个字符。

escChar

字符串内部通常不应包含结束分隔符，如果要允许结束分隔符出现在正文之中，需要设置escChar=c，c是转意字符，表示后面的字符将被视为文本而不是分隔符。

Multiline

默认情况下，字符串可能不包含换行符。 如果要允许解析器匹配扩展到多行的带引号的字符串，请传递参数“multiline = True”。

>>> qs = pp.QuotedString('"')

>>> print qs.parseString('"semprini"')

['semprini']

>>> cc = pp.QuotedString('/\*', endQuoteChar='\*/')

>>> print cc.parseString("/\* Attila the Bun \*/")

[' Attila the Bun ']

>>> pat = pp.QuotedString('"', escChar='\\')

>>> print pat.parseString(r'"abc\"def"')

['abc"def']

>>> text = """'Ken

... Obvious'"""

>>> print text

'Ken

Obvious'

>>> pat = pp.QuotedString("'")

>>> print pat.parseString(text)

pyparsing.ParseException: Expected quoted string, starting with ' ending

with ' (at char 0), (line:1, col:1)

>>> pat = pp.QuotedString("'", multiline=True)

>>> print pat.parseString(text)

['Ken\nObvious']

>>> pat = pp.QuotedString('|')

>>> print pat.parseString('|clever sheep|')

['clever sheep']

>>> pat = pp.QuotedString('|', unquoteResults=False)

>>> print pat.parseString('|clever sheep|')

['|clever sheep|']

## 5.28、Regex：匹配正则表达式

pp.Regex(r, flags=0)

Regex的实例可以匹配正则表达式，表达式的形式应该满足Python re模块的规范。参数r可以是包含正则表达式的字符串，也可以是编译后的re.RegexObject实例。

如果参数r是一个字符串，您可以提供一个flags参数，该参数将作为其flags参数传递给re.match()函数。

>>> r1 = '[a-e]+'

>>> pat1 = pp.Regex(r1)

>>> print pat1.parseString('aeebbaecd', parseAll=True)

['aeebbaecd']

>>> pat2 = pp.Regex(re.compile(r1))

>>> print pat2.parseString('dcbaee', parseAll=True)

['dcbaee']

>>> vowels = r'[aeiou]+'

>>> pat1 = pp.Regex(vowels)

>>> print pat1.parseString('eauoouEAUUO')

['eauoou']

>>> pat2 = pp.Regex(vowels, flags=re.IGNORECASE)

>>> print pat2.parseString('eauoouEAUUO')

['eauoouEAUUO']

## 5.29、SkipTo：提前搜索模式

pp.SkipTo(target, include=False, ignore=None, failOn=None)

该类的实例会在输入中向前搜索，直到找到与target参数匹配的文本。

Include

默认情况下，当找到与目标模式匹配的文本时，该位置将保留在该文本的开头。 如果指定include = True，则位置将保留在匹配文本的末尾，ParseResult将包含一个两元素列表，其第一个元素是跳过的文本，第二个元素是与目标匹配的文本解析器（译注：从示例代码推测，不加include=True，skipTo会停留在target开始的位置，指定了include=True，skipTo停留在target的末尾）。

Ignore

您可以通过指定参数ignore = p来指定在搜索目标时要忽略的模式，其中p是与要忽略的模式匹配的解析器。

failOn

您可以通过传递参数failOn = p来指定不得跳过的模式，其中p是与该模式匹配的解析器。 如果指定了fail0n=p，则SkipTo解析器识别出与p匹配的输入时，解析过程将会fail。

>>> digits = pp.Word(pp.nums)

>>> name = pp.Word(pp.alphas)

>>> ndn = name + pp.SkipTo(digits) + digits + name

>>> print ndn.parseString('Basil%@@^(@^\*(83Fawlty')

['Basil', '%@@^(@^\*(', '83', 'Fawlty']

>>> nn = name + pp.SkipTo(digits, include=True) + name

>>> print nn.parseString('Basil%@@^(@^\*(83Fawlty')

['Basil', ['%@@^(@^\*(', '83'], 'Fawlty']

## 5.30、StringEnd：匹配文本的结尾

pp.StringEnd()

仅当文本位置位于字符串的末尾时，此类的实例才匹配。

>>> noEnd = pp.Word(pp.alphas)

>>> print noEnd.parseString('Dorking...')

['Dorking']

>>> withEnd = pp.Word(pp.alphas) + pp.StringEnd()

>>> print withEnd.parseString('Dorking...')

pyparsing.ParseException: Expected end of text (at char 7), (line:1,

col:8)

## 5.31、StringStart：匹配文本的开头

pp.StringStart()

仅当文本位置位于字符串的开头时，此类的实例才匹配。

>>> number = pp.Word(pp.nums)

>>> name = pp.Word(pp.alphas)

>>> pat1 = number + name

>>> print pat1.parseString(' 7brothers')

['7', 'brothers']

>>> startName = pp.StringStart() + name

>>> pat2 = number + startName

>>> print pat2.parseString(' 7brothers')

pyparsing.ParseException: Expected start of text (at char 4), (line:1,

col:5)

## 5.32、Suppress：忽略结果中的匹配文本

pp.Suppress(p)

此类的实例是与解析器p匹配相同内容的解析器，但是当它与文本匹配时，匹配的文本不会存入返回的ParseResult实例。

>>> name = pp.Word(pp.alphas)

>>> lb = pp.Literal('[')

>>> rb = pp.Literal(']')

>>> pat1 = lb + name + rb

>>> print pat1.parseString('[Pewty]')

['[', 'Pewty', ']']

>>> pat2 = pp.Suppress(lb) + name + pp.Suppress(rb)

>>> print pat2.parseString('[Pewty]')

['Pewty']

另请参见第5.1节“ParserElement：基本解析器构建块”（第11页）中的.suppress（）方法。

## 5.33、Upcase：大写结果

pp.Upcase(p)

此类的实例与解析器匹配的内容匹配，但是当匹配的文本存保存到返回的ParseResults实例中时，所有小写字符都将转换为大写。

>>> name = pp.Upcase(pp.Word(pp.alphas))

>>> print name.parseString('ConfuseACat')

['CONFUSEACAT']

## 5.34、White：匹配空白

pp.White(ws=' \t\r\n', min=1, max=0, exact=0)

这个类的实例匹配一个或多个空白字符。

ws

该参数定义哪些字符被视为空白。

min

此参数定义匹配所需的最小字符数。

max

此参数定义将匹配的最大字符数。

exact

如果指定，则此数字定义将匹配的空白字符的确切数量。

>>> text = ' '

>>> print pp.White().parseString(text)

[' ']

>>> print pp.White(exact=1).parseString(text)

[' ']

>>> print pp.White(max=2).parseString(text)

[' ']

## 5.35、Word：匹配指定集合中的字符

pp.Word(initChars, bodyChars=None, min=1, max=0,

exact=0, asKeyword=False, excludeChars=None)

此类的实例将匹配由参数指定的一组字符中的多个字符。

initChars

如果没有给出bodyChars参数，则此参数指定将匹配的所有字符。 如果提供了bodyChars字符串，则initChars指定有效的初始字符，而bodyChars字符串中第一个字符之后的字符也将匹配

bodyChars

参见initChars

min

要匹配的最小长度

max

要匹配的最大长度。

exact

如果为此参数提供某个数字n的值，则此解析器将恰好匹配n个字符

asKeyword

默认情况下，此解析器将忽略匹配部分后面的文本。 如果指定asKeyword = True，如果匹配部分之后的下一个字符是匹配字符之一（如果没有bodyChars参数，则为initChars中的字符，或者如果存在该关键字参数，则为bodyChars中的字符），匹配将失败。

excludeChars

如果提供，则此参数指定不考虑匹配的字符，即使这些字符被认为匹配。

>>> name = pp.Word('abcdef')

>>> print name.parseString('fadedglory')

['faded']

>>> pyName = pp.Word(pp.alphas+'\_', bodyChars=pp.alphanums+'\_')

>>> print pyName.parseString('\_crunchyFrog13')

['\_crunchyFrog13']

>>> name4 = pp.Word(pp.alphas, exact=4)

>>> print name4.parseString('Whizzo')

['Whiz']

>>> noXY = pp.Word(pp.alphas, excludeChars='xy')

>>> print noXY.parseString('Sussex')

['Susse']

## 5.36、WordEnd：仅在单词的结尾处匹配

pp.WordEnd(wordChars=pp.printables)

仅当前一个字符（如果有的话）是单词字符且下一个字符不是单词字符时，此类的实例才匹配。 可选的wordChars参数指定哪些字符被视为字符; 默认值是所有可打印的非空白字符的集合。

>>> name4 = pp.Word(pp.alphas, exact=4)

>>> name = pp.Word(pp.alphas)

>>> pat = name4 + pp.WordEnd() + name

>>> print pat.parseString('fire truck')

['fire', 'truck']

>>> print pat.parseString('firetruck')

pyparsing.ParseException: Not at the end of a word (at char 4), (line:1,

col:5)

>>> pat2 = name4 + pp.WordEnd(pp.alphas) + pp.Word(pp.nums)

>>> print pat2.parseString('Doug86')

['Doug', '86']

## 5.37、WordStart：仅在单词的开头匹配

pp.WordStart(wordChars=pp.printables)

此类的实例仅在当前位置位于单词的开头时匹配，而前一个字符（如果有的话）不是单词字符。 可选的wordChars参数指定哪些字符被视为字符; 默认值是所有可打印的非空白字符的集合。

>>> goFour = pp.GoToColumn(5)

>>> letters = pp.Word(pp.alphas)

>>> pat = goFour + pp.WordStart(pp.alphas) + letters

>>> print pat.parseString('1234abcd')

['1234', 'abcd']

>>> print pat.parseString('123zabcd')

pyparsing.ParseException: Not at the start of a word (at char 4),

(line:1, col:5)

>>> firstName = pp.WordStart() + pp.Word(pp.alphas)

>>> print firstName.parseString('Lambert')

['Lambert']

>>> badNews = pp.Word(pp.alphas) + firstName

>>> print badNews.parseString('MrLambert')

pyparsing.ParseException: Not at the start of a word (at char 9),

(line:1, col:10)

>>> print badNews.parseString('Mr Lambert')

['Mr', 'Lambert']

## 5.38、ZeroOrMore：匹配任意数量的重复，包括None

pp.ZeroOrMore(p)

此类的实例匹配任意数量的文本项，每个文本项都与解析器p匹配，即使没有匹配的项也是如此。

>>> someWords = pp.ZeroOrMore(pp.Word(pp.alphas))

>>> print someWords.parseString('Comfidown Majorette')

['Comfidown', 'Majorette']

>>> print someWords.parseString('')

[]

>>> print someWords.parseString(' ')

[]

# 6、函数

这些功能在pyparsing模块中定义。

## 6.1、col()：将位置转换为列号

pp.col(loc, s)

此函数的loc参数是字符串s中某个位置的位置（Python索引，从0开始计算）。 返回值是其行内该位置的列号，从1开始计数。换行符（'\ n'）被视为行分隔符。

>>> text = 'abc\nde\n'

>>> for k in range(len(text)):

... print "Position {0}: col {1}, lineno {2}".format(

... k, pp.col(k, text), pp.lineno(k, text))

...

Position 0: col 1, lineno 1

Position 1: col 2, lineno 1

Position 2: col 3, lineno 1

Position 3: col 1, lineno 1

Position 4: col 1, lineno 2

Position 5: col 2, lineno 2

Position 6: col 1, lineno 2

## 6.2、countedArray：解析N后跟N个事物

pp.countedArray(parser, intExpr=None)

这个相当专业的函数创建一个匹配某些计数的解析器，然后是匹配某个解析器的模式的多次出现，如“3 Moe Larry Curly”。 此函数将值列表存入返回的ParseResults，省略计数本身。 请注意，此示例中的整数返回为str类型，而不是int类型。

（译注：这个解析器的匹配过程，有点像早年在UNIX网络编程中常见的TLV格式数据，网络上传来的报文，格式这样的：Type -- 两个字节、 Length -- 两个字节、 Value -- 长度由Length字段指定。想要正确的读取Value，需要先确定Length的值。）

>>> number = pp.Word(pp.nums)

>>> print pp.countedArray(number).parseString('3 18 37 33')

[['18', '37', '33']]

>>> countedQuoted = pp.countedArray(pp.QuotedString('"'))

>>> print countedQuoted.parseString('3 "Moe" "Larry" "Curly Joe"')

[['Moe', 'Larry', 'Curly Joe']]

如果计数由于某种原因而不是通常形式的整数，则可以提供intExpr关键字参数，该参数指定将与计数匹配的解析器并将其作为Python int返回。

>>> def octInt(toks): # Convert octal to int

... return int(toks[0], 8)

...

>>> octal = pp.Word('01234567').setParseAction(octInt)

>>> print octal.parseString('77')

[63]

>>> print pp.countedArray(number, intExpr=octal).parseString(

... '11 1 2 3 4 5 6 7 8 9')

[['1', '2', '3', '4', '5', '6', '7', '8', '9']]

## 6.3、delimitedList()：为分隔列表创建解析器

delimitedList(parser, delim=",", combine=False)

此函数为序列P D P D ... D P创建解析器，其中P匹配某个解析器，D是某个分隔符，默认为“，”。

默认情况下，结果是省略了分隔符（D项）的P项列表。

>>> text = "Arthur, Bedevere, Launcelot, Galahad, Robin"

>>> name = pp.Word(pp.alphas)

>>> nameList = pp.delimitedList(name)

>>> print nameList.parseString(text)

['Arthur', 'Bedevere', 'Launcelot', 'Galahad', 'Robin']

要包含分隔符并将整个结果融合到一个字符串中，请传入参数com bine = True。

>>> allNames = pp.delimitedList(name, delim=', ', combine=True)

>>> print allNames.parseString(text)

['Arthur, Bedevere, Launcelot, Galahad, Robin']

>>> badExample = pp.delimitedList(name, combine=True)

>>> print badExample.parseString(text)

['Arthur']

最后一个示例仅匹配一个名称，因为Combine类禁止在其内部片段中跳过空格。

## 6.4、dictOf()：从键/值对构建字典

pp.dictOf(keyParser, valueParser)

这个函数构建的解析器，可以匹配键和值交替出现的文本序列。匹配到键值对之后，会以字典的形式保存到返回的ParseResult中。

参数keyParser是与键文本匹配的解析器，参数valueParser是与值文本匹配的解析器。

下文这个简单的例子，可以让你了解dictOf大致是怎么用的。要匹配的文本是一系列五个字符的项目，每个项目都是一个单字母的颜色代码，后跟一个四字符的颜色名称。

>>> colorText = 'R#F00 G#0F0 B#00F'

>>> colorKey = pp.Word(pp.alphas, exact=1) # Matches 1 letter

>>> rgbValue = pp.Word(pp.printables, exact=4) # Matches 4 characters

>>> rgbPat = pp.dictOf(colorKey, rgbValue)

>>> rgbMap = rgbPat.parseString(colorText)

>>> rgbMap.keys()

['B', 'R', 'G']

>>> rgbMap['G']

'#0F0

下文是个略显微妙的例子。文本的格式为“degree：name; ...”，其中degree部分是数字形式的音阶，name部分是该音符的名称。这是第一次尝试：

>>> text = '1, do; 2, re; 3, mi; 4, fa; 5, sol; 6, la; 7, ti'

>>> key = pp.Word(pp.nums) + pp.Suppress(',')

>>> value = pp.Word(pp.alphas) + pp.Suppress(';')

>>> notePat = pp.dictOf(key, value)

>>> noteNames = notePat.parseString(text)

>>> noteNames.keys()

['1', '3', '2', '5', '4', '6']

>>> noteNames['4']

'fa'

>>> noteNames['7']

KeyError: '7

注意，最后一个键值对缺失了。这是因为value的模式要求以分号结尾,而text的结尾没有分号。要是不仔细检查匹配的结果，很可能不会注意到最后一个键值对没被匹配到。这是在调用 .parseString（）时总是使用parseAll = True选项的一个好理由。 注意parseAll=True是如何揭示错误的：

>>> noteNames = notePat.parseString(text, parseAll=True)

pyparsing.ParseException: Expected end of text (at char 43), (line:1,

col:44)

修改文本的定义很容易，但是让我们修改解析器，以便将值定义为以分号或字符串结尾结束。

>>> value = pp.Word(pp.alphas) + (pp.StringEnd() | pp.Suppress(';'))

>>> notePat = pp.dictOf(key, value)

>>> noteNames = notePat.parseString(text)

>>> noteNames.keys()

['1', '3', '2', '5', '4', '7', '6']

>>> noteNames['7']

'ti

## 6.5、downcaseTokens()：转换成小写的parse action

将此函数作为解析操作的效果，是解析器在其ParseResults中返回的所有字母都统一成小写。

>>> sameName = pp.Word(pp.alphas)

>>> print sameName.parseString('SpringSurprise')

['SpringSurprise']

>>> lowerName = sameName.setParseAction(pp.downcaseTokens)

>>> print lowerName.parseString('SpringSurprise')

['springsurprise']

## 6.6、getTokensEndLoc()：找到tokens的结尾

如果在解析操作中使用，则此函数不接受任何参数，返回tokens结束的位置，从0开始计数。

>>> def findEnd(s, loc, toks):

... print pp.getTokensEndLoc()

...

>>> letters = pp.Word(pp.alphas).setParseAction(findEnd)

>>> print letters.parseString('pepperpot')

9

['pepperpot']

## 6.7、line()：一个位置出现在哪一行？

pp.line(loc, text)

给定一个字符串text和一个代表字符串中某个位置的数字loc，line函数返回该位置所在的行，不包含换行符（译注：从下述例子看，这个函数返回的是去掉换行符的‘一行’）。

>>> text = 'abc\nde\nf\n'

>>> for loc in range(len(text)):

... print "{0:2d} '{1}'".format(loc, pp.line(loc, text))

...

0 'abc'

1 'abc'

2 'abc'

3 'abc'

4 'de'

5 'de'

6 'de'

7 'f'

8 'f

## 6.8、lineno()：将位置转换为行号

pp.lineno(loc, s)

loc参数是字符串s中的某个位置（Python索引，从0开始计算）。 返回值是该位置的行号，从1开始计算。换行符（'\ n'）被视为行分隔符。演示此函数的示例，请参见第6.1节。

## 6.9、matchOnlyAtCol()：从指定的列开始匹配

pp.matchOnlyAtCol(col)

使用此函数作为解析操作，强制解析器仅匹配行中的特定列号，从1开始计数。

>>> pound2 = pp.Literal('#').setParseAction(pp.matchOnlyAtCol(1))

>>> colorName = pp.Combine(pound2 + pp.Word(pp.hexnums, exact=6))

>>> print colorName.parseString('#00ff88')

['#00ff88']

>>> offColor = pp.Optional(pp.Literal('-')) + colorName

>>> print offColor.parseString('#ff0044')

['#ff0044']

>>> print offColor.parseString('-#ff0044')

pyparsing.ParseException: matched token not at column 1 (at char 1),

(line:1, col:2)

6.10、matchPreviousExpr()：匹配上一个表达式匹配的文本

pp.matchPreviousExpr(parser)

这函数返回一个新的解析器，它与参数parser匹配的模式，相同的值。

>>> name = pp.Word(pp.alphas)

>>> name2 = pp.matchPreviousExpr(name)

>>> dash2 = name + pp.Literal('-') + name2

>>> print dash2.parseString('aye-aye')

['aye', '-', 'aye']

>>> print dash2.parseString('aye-nay')

pyparsing.ParseException: (at char 0), (line:1, col:1)

>>> print dash2.parseString('no-now')

pyparsing.ParseException: (at char 0), (line:1, col:1)

上面的最后一个示例失败了，因为即使在连字符之前和之后都出现字符串“no”，name2解析器先匹配整个字符串“now”，再查看它（now）是否与之前出现的“no”相匹配。可以与第6.11节 --matchPreviousLiteral()：（TODO：待确定6.11的标题该如何翻译） -- 行为比较。

## 6.11、matchPreviousLiteral()：

pp.matchPreviousLiteral(parser)

这个函数的工作方式类似于第6.10节“matchPreviousExpr（）：匹配上一个表达式匹配的文本”（参见第39页），但返回的解析器与精确匹配解析器匹配的字符，不考虑任何后续上下文【译注：没看懂这两个函数的区别】。 将下面的示例与6.10节中的示例进行比较：

>>> name = pp.Word(pp.alphas)

>>> name2 = pp.matchPreviousLiteral(name)

>>> dash2 = pp.Combine(name + pp.Literal('-') + name2)

>>> print dash2.parseString('foo-foofaraw')

['foo-foo']

>>> print dash2.parseString('foo-foofaraw', parseAll=True)

pyparsing.ParseException: Expected end of text (at char 7), (line:1, col:8)

译注：

即使换成matchPreviousExpr函数，匹配'foo-foofaraw'时也会抛出异常

>>> import pyparsing as pp

>>> name = pp.Word(pp.alphas)

>>> name2 = pp.matchPreviousExpr(name)

>>> dash2 = name + pp.Literal('-') + name2

>>> print(dash2.parseString('foo-foofaraw', parseAll=True))

File "/home/superset\_master/lib/python3.6/site-packages/pyparsing.py", line 5040, in mustMatchTheseTokens

raise ParseException("",0,"")

pyparsing.ParseException: (at char 0), (line:1, col:1)

【译注：6.10和6.11两个函数提供了上下文相关（context-sensitive）的能力。两者的区别，似乎在于上下文的范围，但具体有什么区别，没找到合适的例子。在stackoverflos上有更详细的讨论：[Does Pyparsing Support Context-Sensitive Grammars?](https://stackoverflow.com/questions/15103106/does-pyparsing-support-context-sensitive-grammars" \t "https://zhuanlan.zhihu.com/p/61077860/_blank)】

## 6.12、nestedExpr()：嵌套列表的解析器

pp.nestedExpr(opener='(', closer=')', content=None, ignoreExpr=I)

该函数返回一个解析嵌套列表的解析器，嵌套列表，也就是所谓的“LCR”序列：

* opener参数，即L，是分隔符开始的部分，默认为“（”。
* closer参数，即R，是分隔符结束的部分，默认为“（。
* content参数，即C，是上述开始分隔符、结束分隔符之间的内容。在C的任何位置，可能会包含任意数量的另一层级的LCR。如果没有指定content参数，每一层LCR会以列表的形式保存在ParseResults中（译注：句子很复杂，没想出来该怎么翻译，只好简化处理了）。
* （TODO：此处还有一点，待补充）

If the content part may contain the L or R delimiter strings inside quote strings, you can specify an ignoreExpr parser that describes what a quoted string looks like in your context, and the parsing processwill not treat those occurrences as delimiters. The default value I is an instance of Section 5.27, “QuotedString: Match a delimited string” (p. 29). Ifyou specify ignoreExpr=None, no occurrences of the delimiter characters will be ignored.

>>> text = '{They {mean to {win}} Wimbledon}'

>>> print pp.nestedExpr(opener='{', closer='}').parseString(text)

[['They', ['mean', 'to', ['win']], 'Wimbledon']]

>>> text = '''(define (factorial n)

... (fact-iter 1 1 n))'''

>>> print pp.nestedExpr().parseString(text)

[['define', ['factorial', 'n'], ['fact-iter', '1', '1', 'n']]]

## 6.13、oneOf()：匹配多个文字（中的一个），优先检查最长的

pp.oneOf(alternatives, caseless=False)

该函数返回的解析器，可以与一组字符串中的一个相匹配。 特别地，如果任何字符串是另一个的子字符串，这个解析器始终首先检查较长的字符串; 例如，在解析一组关键字时，这样的行为很有用。

* alternatives参数指定解析器将匹配的不同字符串。该参数既可以是字符串列表，也可以是由空格分隔的字符串。
* 默认情况下，匹配区分大小写。 要指定不区分大小写的匹配，请传递参数“caseless = True”。

>>> keyList = pp.oneOf('as assert break')

>>> print keyList.parseString('assert yes')

['assert']

>>> print keyList.parseString('as read')

['as']

## 6.14、srange()：指定字符范围

pp.srange("[ranges]")

可以用该函数创建一个字符串，再将字符串传递给pp.Word（），这样就可以创建一个解析器，用于匹配[ranges]中出现的字符集。 [ranges]参数中可以使用字符代码范围，这样就不必指定每个字符。 [ranges]参数的语法类似于通用正则表达式的“[...]”结构。

ranges参数可以由以下一个或多个实例组成：

* 单个字符。
* 反斜杠后跟单个字符，这样解析器可以匹配一些“元字符”，如“ - ”、“]”等（分别写作'\-'和'\]'）。
* 十六进制数字，格式为“\xHH”。
* 八进制数字，格式为"\0N..."，其中N可以是一位、两位、或三位八进制数字。
* 两个上述形式的字符，用'-'隔开，代表这两个字符，以及两者之间的所有字符。例如，pp.srange('[a-z]')会返回一个匹配小写字母的解析器。

以下示例，演示如何使用此函数创建Python标识符的解析器。

>>> first = pp.Word(pp.srange('[\_a-zA-Z]'), exact=1)

>>> rest = pp.Optional(pp.Word(pp.srange('[\_0-9a-zA-Zz]')))

>>> ident = pp.Combine(first + rest)

>>> print ident.parseString('runcorn\_Abbey')

['runcorn\_Abbey']

>>> print ident.parseString('N23')

['N23']

>>> print ident.parseString('0xy')

pyparsing.ParseException: Expected W:(\_abc...) (at char 0), (line:1,

col:1)

## 6.15、removeQuotes()：去掉开头结尾的引号

要从结果中删除匹配文本的第一个和最后一个字符，请将此函数用作解析操作。

>>> slashPat = pp.Combine(pp.Literal('/') + pp.Word(pp.alphas) +

... pp.Literal('/'))

>>> print slashPat.parseString('/Llamas/')

['/Llamas/']

>>> slash2 = slashPat.addParseAction(pp.removeQuotes)

>>> print slash2.parseString('/Llamas/')

['Llamas']

6.16、replaceWith()：替换匹配文本的常量值

pp.replaceWith(literal)

如果将此函数作为解析操作附加到解析器，则当解析器匹配某些文本时，本来会被保存到ParseResults中的被匹配的文本，将会被literal参数取代。

>>> password = pp.Word(pp.printables).setParseAction(

... pp.replaceWith('\*\*\*\*\*\*\*\*'))

>>> print password.parseString('shazam')

['\*\*\*\*\*\*\*\*']

>>> print password.parseString('abracadabra')

['\*\*\*\*\*\*\*\*']

## 6.17、traceParseAction()：可以输出调试信息的parse action装饰器

您可以使用此装饰器来包装解析操作（parse action），这样，无论何时使用该parse action，sys.stderr流上都会显示两条消息，显示入口参数和返回值。

>>> @pp.traceParseAction

... def basil(toks):

... '''Dummy parse action

... '''

... return None

...

>>> number = pp.Word(pp.nums).setParseAction(basil)

>>> print number.parseString('575')

>>entering wrapper(line: '575', 0, ['575'])

<<leaving wrapper (ret: None)

['575']

## 6.18、upcaseTokens()：转换成大写的解析动作

此函数用作parse action的效果，是ParseResults中的所有字母会被转换成大写。

>>> sameName = pp.Word(pp.alphas)

>>> print sameName.parseString('SpringSurprise')

['SpringSurprise']

>>> upperName = pp.Word(pp.alphas).setParseAction(pp.upcaseTokens)

>>> print upperName.parseString('SpringSurprise')

['SPRINGSURPRISE']

# 7、变量

这些变量在pyparsing模块中定义。

## 7.1、alphanums：字母数字字符

变量pp.alphanums是一个包含小写和大写字母和十进制数字的字符串。

## 7.2、alphas：字母

变量pp.alphas是一个包含所有小写和大写字母的字符串。

## 7.3、alphas8bit：补充Unicode字母

变量pp.alphas8bit是一个Unicode字符串，包含Latin-1 Supplement（范围U000C0-U000FF）中被视为字母的字符。更多信息，请参阅ISO 9573-2003 Unicode实体组。

## 7.4、cStyleComment：匹配C语言注释

变量pp.cStyleComment是一个与C语言中的注释匹配的解析器。

>>> print pp.cStyleComment.parseString('''/\* First line.

... Second line.

... Third line. \*/''')

['/\* First line.\n Second line.\n Third line. \*/']

## 7.5、commaSeparatedList：解析以逗号分隔的列表

变量pp.commaSeparatedList是一个匹配任何文本的解析器，只要有逗号（“，”）就会将其分成几部分。

>>> print pp.commaSeparatedList.parseString('spam, spam, spam, and spam')

['spam', 'spam', 'spam', 'and spam']

## 7.6、cppStyleComment：解析C++注释

变量pp.cppStyleComment是一个与C ++语言中的注释匹配的解析器。 注释可以是C风格的（参见第7.4节“cStyleComment：匹配C语言注释”（第43页）），也可以是C ++风格（参见第7.8节“dblSlashComment：Parser，以”//“开头的注释）（第44页））。

>>> text = '// Look out of the yard? What will we see?'

>>> print pp.cppStyleComment.parseString(text)

['// Look out of the yard? What will we see?']

>>> print pp.cppStyleComment.parseString('/\* Author: R. J. Gumby \*/')

['/\* Author: R. J. Gumby \*/']

## 7.7、dblQuotedString：用“......”括起来的字符串

变量pp.dblQuotedString是一个解析器，它匹配用双引号（“”“）字符括起来的字符串。文本中的序列”\“”不会被解释为内部双引号字符。

>>> text = '"Gannet on a stick"'

>>> print pp.dblQuotedString.parseString(text)

['"Gannet on a stick"']

译注：

>>> text = '"Gannet on a stick\" inernal quote"'

>>> print(pp.dblQuotedString.parseString(text))

['"Gannet on a stick"']

## 7.8、dblSlashComment：解析以“//”开头的注释

变量pp.dblSlashComment是一个解析器，它匹配C ++或Java程序中以“//”开头的注释（从“//”到行尾）。

>>> text = '// Comment\nNext line'

>>> print pp.dblSlashComment.parseString(text)

['// Comment']

## 7.9、empty：匹配空内容

变量pp.empty是第5.9节--“Empty：匹配空内容”--的实例（参见第18页）。

>>> print pp.empty.parseString('Matches nothing')

[]

## 7.10、hexnums：全部十六进制数字

变量pp.hexnums是一个包含有效十六进制字符的字符串：十进制数字加上大写字母和小写字母的A到F。

## 7.11、javaStyleComment：Java语法的注释

变量pp.javaStyleComment可以匹配两种格式的注释，或者是第7.4节“cStyleComment：匹配C语言注释”的格式（参见第43页），或者是第7.6节“cppStyleComment：匹配 C ++注释”（参见第44页）。 请注意，“//”样式注释始终在行的末尾结束。

>>> print pp.javaStyleComment.parseString('''/\*

... multiline comment \*/''')

['/\*\nmultiline comment \*/']

>>> print pp.javaStyleComment.parseString('''// This comment

... intentionally left almost blank\n''')

['// This comment']

## 7.12、lineEnd：LineEnd的一个实例

变量pp.lineEnd是第5.15 -- “LineEnd：匹配行尾”-- 的实例（参见第24页）。

>>> print pp.lineEnd.parseString('\nThis remains unparsed')

['\n']

## 7.13、lineStart：LineStart的一个实例

变量pp.lineStart是Section 5.16--“LineStart：匹配行的开始”--的实例（参见第24页）。

在下文的示例中，模式initialWord匹配行开头的字符串。 示例文本能被匹配，是因为字符串的开头也被视为行的开头。

>>> initialWord = pp.lineStart + pp.Word(pp.alphas)

>>> print initialWord.parseString('Silly party\n')

['Silly']

在下一个示例中，模式pat1匹配一个单词，后跟一行的结尾，后跟模式initialWord。 第一个示例文本匹配，因为'\ n'之后的位置被视为一行的开头。

>>> pat1 = pp.Word(pp.alphas) + pp.lineEnd + initialWord

>>> print pat1.parseString('Very\nSilly Party')

['Very', '\n', 'Silly']

下文是一个匹配失败的例子。pat2的第一部分与'Harpenden'一词相匹配，之后的空格被自动跳过，接下来处理“southeast”一词的开头， 然后pp.lineStart解析器匹配失败，因为“southeast”之前既不是换行符，也不是字符串的开头，而是一个空格。

>>> pat2 = pp.Word(pp.alphas) + initialWord

>>> print pat2.parseString('Harpenden southeast')

pyparsing.ParseException: Expected lineStart (at char 9), (line:1, col:10)

## 7.14、nums：十进制数字

变量pp.nums是一个包含十位十进制数字的字符串。

>>> pp.nums

'0123456789'

## 7.15、printables：所有可打印的非空白字符

变量pp.printables是一个包含所有可打印、非空白字符的字符串。 在128个7位ASCII字符中，前32个是控制字符、counting space（译注：在ASCII码表中没找到能和counting space对应的描述，从实际输出看，应该是指空格）和DEL，剩下的94个是就是printables变量对应的可打印字符。

>>> len(pp.printables)

94

>>> print pp.printables

0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()\*+,-

./:;<=>?@[\]^\_`{|}~

译注：把printables放在中括号里面，能看到不包含空格：

>>> print('[%s]' % (pp.printables))

[0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~]

## 7.16、punc8bit：一些Unicode标点符号

变量pp.punc8bit是一个unicode字符串，包含许多被认为是标点符号的字符：具体来说，代码点U000A1-U000BF加上U000D7（×）和U000F7（÷）。

>>> pp.punc8bit

u'\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2

\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xd7\xf7'

## 7.17、pythonStyleComment：Python语言风格的注释

变量pythonStyleComment是一个解析python风格注释的解析器：注释以“＃”开头，结束于行尾（或整个文本字符串的结尾，以先到者为准）。

>>> print pp.pythonStyleComment.parseString('# The Silly Party take Luton.')

['# The Silly Party take Luton.']

>>> codePart = pp.ZeroOrMore(pp.CharsNotIn('#\n'))

>>> commentPart = pp.pythonStyleComment

>>> line = codePart + pp.Optional(commentPart)

>>> print line.parseString(' nVikings = 0')

[' nVikings = 0']

>>> print line.parseString(' nVikings = 0 # Reset Viking count')

[' nVikings = 0 ', '# Reset Viking count']

## 7.18、quotedString：解析默认的字符串

常见的字符串，无论是双引号("...") 还是单引号('...') 括起来的，变量pp.quotedString是第5.27节“QuotedString：匹配分隔字符串”的一个实例，可以解析符合该规范的“括起来”的字符串。

>>> print pp.quotedString.parseString('"Ftang ftang"')

['"Ftang ftang"']

>>> print pp.quotedString.parseString("'Ole Biscuitbarrel'")

["'Ole Biscuitbarrel'"]

需要注意的是，引号也被放到解析结果中了。如果不需要引号，可以附加一个action去掉前导、收尾的引号，参见6.15节，“removeQuotes()：去掉前导、收尾的引号”（第42页）。

>>> justTheGuts = pp.quotedString.addParseAction(pp.removeQuotes)

>>> print justTheGuts.parseString("'Kevin Phillips Bong'")

['Kevin Phillips Bong']

## 7.19、restOfLine：匹配当前行的剩余部分

变量restOfLine是一个解析器，可以匹配零个或多个字符，但不包括换行符（或者字符串的结尾，以先到者为准）。

>>> text = 'Wolverhampton 3\nBristol nought\n'

>>> print pp.restOfLine.parseString(text)

['Wolverhampton 3']

要匹配包括换行符的剩余部分（如果有的话），可以把restOfLine和7.12节的lineEnd一起使用。

>>> toNextLine = pp.Combine(pp.restOfLine + pp.lineEnd)

>>> print toNextLine.parseString(text)

['Wolverhampton 3\n']

## 7.20、sglQuotedString：用'......'括起来的字符串

变量pp.sglQuotedString是一个解析器，它匹配用单引号（“'”）字符括起来的字符串。 文本中的序列“\”不会被解释为内部单引号字符。

>>> text = "'Do I get wafers with it?'"

>>> print pp.sglQuotedString.parseString(text)

["'Do I get wafers with it?'"]

>>> escaper = "'Don\\'t'"

>>> print escaper

'Don\'t'

>>> result = pp.sglQuotedString.parseString(escaper)

>>> print result[0]

'Don\'t

如果内部“\”“序列被解释为转义符，则上面的最后一行将显示为：

"Don't"

## 7.21、stringEnd：匹配字符串的结尾

变量pp.stringEnd包含5.30节的实例，“StringEnd：匹配文本的结尾”（第31页）。

>>> wordAtEnd = pp.Word(pp.alphas) + pp.stringEnd

>>> print wordAtEnd.parseString("Leicester")

['Leicester']

>>> print wordAtEnd.parseString("West Byfleet")

pyparsing.ParseException: Expected stringEnd (at char 5), (line:1, col:6)

## 7.22、unicodeString：匹配Python样式的Unicode字符串

变量unicodeString是一个解析器，用来匹配Python样式的Unicode字符串：字母'u'，后面是被单引号或双引号括起来的字符串。这个解析器不支持三引号字符串。

>>> print pp.unicodeString.parseString("u'Tarquin'")

["u'Tarquin'"]

>>> print pp.unicodeString.parseString('u"Jethro"')

['u"Jethro"']

>>> print pp.unicodeString.parseString('u"""Two\nSheds\nJackson"""')

['u""']

参考

【1】

pyparsing的作者Paul在stackoverflow上回答过一个combine的问题：

PyParsing: What does Combine() do?​stackoverflow.com

