pyparsing quick reference: A Python text processing tool

John W. Shipman

2013-03-05 12:52

Abstract

A quick reference guide for *pyparsing*, a recursive descent parser framework for the Python programming language.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

[1] http://www.nmt.edu/tcc/help/pubs/pyparsing/
[2] http://www.nmt.edu/tcc/help/pubs/pyparsing/pyparsing.pdf

1. *pyparsing*: A tool for extracting information from text

The purpose of the *pyparsing* module is to give programmers using the Python programming language[3] a tool for extracting information from structured textual data.

In terms of power, this module is more powerful than regular expressions[4], as embodied in the Python `re` module[5], but not as general as a full-blown compiler.

In order to find information within structured text, we must be able to describe that structure. The *pyparsing* module builds on the fundamental syntax description technology embodied in Backus-Naur Form[6], or BNF. Some familiarity with the various syntax notations based on BNF will be most helpful to you in using this package.

The way that the *pyparsing* module works is to match patterns in the input text using a recursive descent[7] parser: we write BNF-like syntax productions, and *pyparsing* provides a machine that matches the input text against those productions.

The *pyparsing* module works best when you can describe the exact syntactic structure of the text you are analyzing. A common application of *pyparsing* is the analysis of log files. Log file entries generally have a predictable structure including such fields as dates, IP addresses, and such. Possible applications of the module to natural language work are not addressed here.

Useful online references include:

- The *pyparsing* homepage[8].
- Complete online reference documentation[9] for each class, function, and variable.
- See the tutorial[10] at `ONLamp.com`.
- The package author's 2004 tutorial[11] is slightly dated but still useful.
- For a small example (about ten syntax productions) of an application that uses this package, see *icalparse: A pyparsing parser for `.calendar` files*[12].
- For a modest example, *abaraw: A shorthand notation for bird records*[13] describes a file format with about thirty syntax productions. The actual implementation appears in a separate document, *abaraw internal maintenance specification*[14], which is basically a *pyparsing* core with a bit of application logic that converts it to XML to be passed to later processing steps.

[3] http://www.python.org/
[4] http://en.wikipedia.org/wiki/Regular_expression
[5] http://docs.python.org/2/library/re.html
[6] http://en.wikipedia.org/wiki/Backus-Naur_Form
[7] http://en.wikipedia.org/wiki/Recursive_descent
[8] http://pyparsing.wikispaces.com/
[9] http://packages.python.org/pyparsing/
[10] http://onlamp.com/lpt/a/6435
[11] http://www.ptmcg.com/geo/python/howtousepyparsing.html
[12] http://www.nmt.edu/tcc/help/lang/python/examples/icalparse/
[13] http://www.nmt.edu/~shipman/aba/raw/doc/
[14] http://www.nmt.edu/~shipman/aba/raw/doc/ims/

Note

Not every feature is covered here; this document is an attempt to cover the features most people will use most of the time. See the reference documentation for all the grisly details.

In particular, the author feels strongly that *pyparsing* is not the right tool for parsing XML and HTML, so numerous related features are not covered here. For a much better XML/HTML tool, see *Python XML processing with lxml*[15].

2. Structuring your application

Here is a brief summary of the general procedure for writing a program that uses the *pyparsing* module.

1. Write the BNF that describes the structure of the text that you are analyzing.
2. Install the *pyparsing* module if necessary. Most recent Python install packages will include it. If you don't have it, you can download it from the *pyparsing* homepage[16].
3. In your Python script, import the *pyparsing* module. We recommend that you import it like this:

   ```
   import pyparsing as pp
   ```

   The examples in this document will refer to the module through as `pp`.
4. Your script will assemble a *parser* that matches your BNF. A parser is an instance of the abstract base class `pp.ParserElement` that describes a general pattern.

   Building a parser for your input file format is a bottom-up process. You start by writing parsers for the smallest pieces, and assemble them into larger and larger pieces into you have a parser for the entire file.
5. Build a Python string (type `str` or `unicode`) containing the input text to be processed.
6. If the parser is *p* and the input text is *s*, this code will try to match them:

   ```
   p.parseString(s)
   ```

   If the syntax of *s* matches the syntax descriped by *p*, this expression will return an object that represents the parts that matched. This object will be an instance of class `pp.ParseResults`.

   If the input does not match your parser, it will raise an exception of class `pp.ParseException`. This exception will include information about where in the input the parse faiiled.

   The `.parseString()` method proceeds in sequence through your input text, using the pieces of your parser to match chunks of text. The lowest-level parsers are sometimes called *tokens*, and parsers at higher levels are called *patterns*.

   You may attach *parse actions* to any component parser. For example, the parser for an integer might have an attached parse action that converts the string representation into a Python `int`.

[15] http://www.nmt.edu/tcc/help/pubs/pylxml/
[16] http://pyparsing.wikispaces.com/

> **Note**
>
> White space (non-printing) characters such as space and tab are normally skipped between tokens, although this behavior can be changed. This greatly simplifies many applications, because you don't have to clutter the syntax with a lot of patterns that specify where white space can be skipped.

7. Extract your application's information from the returned `ParseResults` instance. The exact structure of this instance depends on how you built the parser.

To see how this all fits together:

- Section 3, "A small, complete example" (p. 5).
- Section 4, "How to structure the returned `ParseResults`" (p. 7).

3. A small, complete example

Just to give you the general idea, here is a small, running example of the use of *pyparsing*.

A Python identifier name consists of one or more characters, in which the first character is a letter or the underbar ("_") character, and any additional characters are letters, underbars, or digits. In extended BNF we can write it this way:

```
first       ::=  letter | "_"
letter      ::=  "a" | "b" | ... "z" | "A" | "B" | ... | "Z"
digit       ::=  "0" | "1" | ... | "9"
rest        ::=  first | digit
identifier  ::=  first rest*
```

That last production can be read as: "an `identifier` consists of one `first` followed by zero or more `rest`".

Here is a script that implements that syntax and then tests it against a number of strings.

trivex

```
#!/usr/bin/env python
#==============================================================
# trivex: Trivial example
#--------------------------------------------------------------

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

import sys
```

The next line imports the *pyparsing* module and renames it as `pp`.

trivex

```
import pyparsing as pp

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

In the next line, the `pp.alphas` variable is a string containing all lowercase and uppercase letters. The `pp.Word()` class produces a parser that matches a string of letters defined by its first argument; the `exact=1` keyword argument tells that parser to accept exactly one character from that string. So `first` is a parser (that is, a `ParserElement` instance) that matches exactly one letter or an underbar.

trivex

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

The `pp.alphanums` variable is a string containing all the letters and all the digits. So the `rest` pattern matches one or more letters, digits, or underbar characters.

trivex

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

The Python "+" operator is overloaded for instances of the `pp.ParserElement` class to mean sequence: that is, the `identifier` parser matches what the `first` parser matches, followed optionally by what the `rest` parser matches.

trivex

```
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)
```

When you call the `.parseString()` method on an instance of the `pp.ParserElement` class, either it returns a list of the matched elements or raises a `pp.ParseException`.

trivex

```
    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()
```

Here is the output from this script:

```
---Test for 'a'
  Matches: ['a']
```

```
---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', '_']
```

The return value is an instance of the `pp.ParseResults` class; when printed, it appears as a list of the matched strings. You will note that for single-letter test strings, the resulting list has only a single element, while for multi-letter strings, the list has two elements: the first character (the part that matched `first`) followed by the remaining characters that matched the `rest` parser.

If we want the resulting list to have only one element, we can change one line to get this effect:

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

The `pp.Combine()` class tells *pyparsing* to combine all the matching pieces in its argument list into a single result. Here is an example of two output lines from the revised script:

```
---Test for '_bride_of_mothra'
  Matches: ['_bride_of_mothra']
```

4. How to structure the returned `ParseResults`

When your input matches the parser you have built, the `.parseString()` method returns an instance of class `ParseResults`.

For a complex structure, this instance may have many different bits of information inside it that represent the important pieces of the input. The exact internal structure of a `ParseResults` instance depends on how you build up your top-level parser.

You can access the resulting `ParseResults` instance in two different ways:

- As a list. A parser that matches *n* internal components will return a result *r* that can act like a list of the *n* strings that matched those components. You can extract the *n*th element as "*r*`[`*n*`]`"; or you can convert it to an actual list of strings with the `list()` function.

```
>>> 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']
```

- As a dictionary. You can attach a *results name* `r` to a parser by calling its `.setResultsName(s)` method (see Section 5.1, "`ParserElement`: The basic parser building block" (p. 11)). Once you have done that, you can extract the matched string from the `ParseResults` instance `r` as "`r[s]`".

```
>>> number = pp.Word(pp.nums).setResultsName('nVache')
>>> result = number.parseString('17')
>>> print result
['17']
>>> result['nVache']
'17'
```

Here are some general principles for structuring your parser's `ParseResults` instance.

4.1. Use `pp.Group()` to divide and conquer

Like any nontrivial program, structuring a parser with any complexity at all will be more tractable if you use the "divide and conquer" principle, also known as stepwise refinement[17].

In practice, this means that the top-level `ParseResults` should contain no more than, say, five or seven components. If there are too many components at this level, look at the total input and divide it into two or more subparsers. Then structure the top level so that it contains just those pieces. If necessary, divide the smaller parsers into smaller parsers, until each parser is clearly defined in terms of built-in primitive functions or other parsers that you have built.

Section 5.13, "`Group`: Group repeated items into a list" (p. 23) is the basic tool for creating these levels of abstraction.

- Normally, when your parser matches multiple things, the result is a `ParseResults` instance that acts like a list of the strings that matched. For example, if your parser matches a list of words, it might return a `ParseResults` that *prints* as if it were a list. Using the `type()` function we can see the actual type of the result, and that the components are Python strings.

```
>>> 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'>
```

- However, when you apply `pp.Group()` to some parser, all the matching pieces are returned in a single `pp.ParseResults` that acts like a list.

[17] http://en.wikipedia.org/wiki/Stepwise_refinement

For example, suppose your program is disassembling a sequence of words, and you want to treat the first word one way and the rest of the words another way. Here's our first attempt.

```
>>> ungrouped = word + phrase
>>> result = ungrouped.parseString('imaginary farcical aquatic ceremony')
>>> print result
['imaginary', 'farcical', 'aquatic', 'ceremony']
```

That result doesn't really match our concept that the parser is a sequence of two things: a single word, followed by a sequence of words.

By applying `pp.Group()` like this, we get a parser that will return a sequence of two items that match our concept.

```
>>> 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. The `grouped` parser has two components: a `word` and a `pp.Group`. Hence, the result returned acts like a two-element list.
2. The first element is an actual string, `'imaginary'`.
3. The second part is another `pp.ParseResults` instance that acts like a list of strings.

So for larger grammars, the `pp.ParseResults` instance, which the top-level parser returns when it matches, will typically be a many-layered structure containing this kind of mixture of ordinary strings and other instances of `pp.ParseResults`.

The next section will give you some suggestions on manage the structure of these beasts.

4.2. Structuring with results names

For parsers that appear only once at a specific level, consider using `.setResultsName()` to associate a results name with that parser. This allows you to retrieve the matched text by treating the returned `ParseResults` instance as a dictionary, and the results name as the key.

Design rules for this option:

- Access by name is more robust than access by position. The structures you are working on may change over time. If you access the results as a list, what was element `[3]` now may suddenly become element `[4]` when the underlying structure changes.

  If, however, you give the results some name like `'swampName'`, the access code `result['swampName']` will probably continue to work even if other names are added later to the same result.
- In one `ParseResults`, use either access by position or access by key (that is, by results name), not both. If some subelement of a parser has a results name, and some subelements do not have a results name, the matching text for all those subelements will be mixed together in the result.

Here's an example showing what happens when you mix positional and named access at the same level: in bull-riding, the total score is a combination of the rider's score and the bull's score.

```
>>> 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']
```

In the four-element list shown above, you can access the first and third elements by name, but the second and fourth would be accessible only by position.

A more sensible way to structure this parser would be to write a parser for the combination of a name and a score, and then combine two of those for the overall parser.

```
>>> 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'
```

- Don't use a results name for a repeated element. If you do, only the last one will be accessible by results name in the `ParseResults`.

```
>>> 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']
```

A better approach is to wrap the entire name in a `pp.Group()` and then apply the results name to that.

```
>>> 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. Classes

Here are the classes defined in the *pyparsing* module.

5.1. `ParserElement`: The basic parser building block

Definition: A *parser* is an instance of some subclass of `pp.ParserElement`.

In the process of building a large syntax out of small pieces, define a parser for each piece, and then combine the pieces into larger and large aggregations until you have a parser that matches the entire input.

To assemble parsers into larger configurations, you will use *pyparsing*'s built-in classes such as `pp.And`, `pp.Or`, and `pp.OneOrMore`. Each of these class constructors returns a parser, and many of them accept one or more parsers as arguments.

For example, if a certain element of the syntax described by some parser `p` is optional, then `pp.Optional(p)` returns another parser – that is, another instance of a subclass of `pp.ParserElement` – that will match pattern `p` if it occurs at that point in the input, and do nothing if the input does not match `p`.

Here are the methods available on a parser instance *p* that subclasses `pp.ParserElement`.

`p.addParseAction(`*f_1*`, `*f_2*`, ...)`
Returns a copy of *p* with one or more additional parse actions attached. See the `p.setParseAction()` method below for a discussion of parse actions.

`p.copy()`
Returns a copy of *p*.

`p.ignore(`*q*`)`
This method modifies *p* so that it ignores any number of occurrences of text that matches pattern *q*. This is a useful way to instruct your parser to ignore comments.

```
>>> number = pp.Word(pp.nums)
>>> name = pp.Word(pp.alphas).ignore(number)
>>> print name.parseString('23 84 98305478 McTeagle')
['McTeagle']
```

`p.leaveWhitespace()`
This method instructs *p* *not* to skip whitespace before matching the input text. The method returns *p*.

When used on a parser that includes multiple pieces, this method suppresses whitespace skipping for all the included pieces. Here is an example:

```
>>> 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)
```

Note

To save space, in subsequent examples we will omit all the "`Traceback`" lines except the last.

```
>>> print nwn.parseString('23xy 47')
pyparsing.ParseException: Expected W:(0123...) (at char 4), (line:1,
col:5)
```

You will note that even though the `num` parser does not skip whitespace, whitespace is still disallowed for the string `' 47'` because the `wn` parser disabled automatic whitespace skipping.

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

Try to match the contents of a file against parser *p*. The argument *f* may be either the name of a file or a file-like object.

If the entire contents of the file does not match *p*, it is not considered an error unless you pass the argument `parseAll=True`.

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

Try to match string *s* against parser *p*. If there is a match, it returns an instance of Section 5.26, "`ParseResults`: Result returned from a match" (p. 28). If there is no match, it will raise a `pp.ParseException`.

By default, if the entirety of *s* does not match *p*, it is not considered an error. If you want to insure that all of *s* matched *p*, pass the keyword argument `parseAll=True`.

`p.scanString(s)`

Search through string *s* to find regions that match *p*. This method is an iterator that generates a sequence of tuples `(r, start, end)`, where *r* is a `pp.ParseResults` instance that represents the matched part, and *start* and *end* are the beginning and ending offsets within *s* that bracket the position of the matched text.

```
>>> 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()`

When this parser is about to be used, call up the Python debugger `pdb`.

`p.setFailAction(f)`

This method modifies *p* so that it will call function *f* if it fails to parse. The method returns *p*.

Here is the calling sequence for a fail action:

```
f(s, loc, expr, err)
```

`s`

The input string.

`loc`

The location in the input where the parse failed, as an offset counting from 0.

expr
The name of the parser that failed.

err
The exception instance that the parser raised.

Here is an example.

```
>>> 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)

Attaches a name to this parser for debugging purposes. The argument is a string. The method returns *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)
```

In the above example, if you convert a parser to a string, you get a generic description of it: the string "`W:(0123...)`" tells you it is a `Word` parser and shows you the first few characters in the set. Once you have attached a name to it, the string form of the parser is that name. Note that when the parse fails, the error message identifies what it expected by naming the failed parser.

p.setParseAction(f_1, f_2, ...)

This method returns a copy of *p* with one or more parse actions attached. When the parser matches the input, it then calls each function f_i in the order specified.

The calling sequence for a parse action can be any of these four prototypes:

```
f()
f(toks)
f(loc, toks)
f(s, loc, toks)
```

These are the arguments your function will receive, depending on how many arguments it accepts:

s
The string being parsed. If your string contains *tab* characters, see the reference documentation[18] for notes about tab expansion and its effect on column positions.

[18] http://packages.python.org/pyparsing/

`loc`
The location of the matching substring as an offset (index, counting from 0).

`toks`
A `pp.ParseResults` instance containing the results of the match.

A parse action can modify the result (the `toks` argument) by returning the modified list. If it returns `None`, the result is not changed. Here is an example parser with two parse actions.

```
>>> 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']
```

***p*`.setResultsName(name)`**
For parsers that deposit the matched text into the `ParseResults` instance returned by `.parseString()`, you can use this method to attach a name to that matched text. Once you do this, you can retrieve the matched text from the `ParseResults` instance by using that instance as if it were a Python `dict`.

```
>>> count = pp.Word(pp.nums)
>>> beanCounter = count.setResultsName('beanCount')
>>> r = beanCounter.parseString('7388')
>>> r.keys()
['beanCount']
>>> r['beanCount']
'7388'
```

The result of this method is a *copy* of *p*. Hence, if you have defined a useful parser, you can create several instances, each with a different results name. Continuing the above example, if we then use the `count` parser, we find that it does not have the results name that is attached to its copy `beanCounter`.

```
>>> r2 = count.parseString('8873')
>>> r2.keys()
[]
>>> print r2
['8873']
```

***p*`.setWhitespaceChars(`*s*`)`**
For parser *p*, change its definition of whitespace to the characters in string *s*.

***p*.suppress()**

This method returns a copy of *p* modified so that it does not add the matched text to the `ParseResult`. This is useful for omitting punctuation. See also Section 5.32, "`Suppress`: Omit matched text from the result" (p. 32).

```
>>> 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']
```

Additionally, these ordinary Python operators are overloaded to work with `ParserElement` instances.

p1*+*p2

Equivalent to "`pp.And(`*p1*`, `*p2*`)`".

p* * *n

For a parser *p* and an integer *n*, the result is a parser that matches *n* repetitions of *p*. You can give the operands in either order: for example, "`3 * p`" is the same as "`p * 3`".

```
>>> threeWords = pp.Word(pp.alphas) * 3
>>> text = "Lady of the Lake"
>>> print threeWords.parseString(text)
['Lady', 'of', 'the']
>>> print threeWords.parseString(text, parseAll=True)
pyparsing.ParseException: Expected end of text (at char 12), (line:1,
col:13)
```

p1* | *p2

Equivalent to "`pp.MatchFirst(`*p1*`, `*p2*`)`".

p1* ^ *p2

Equivalent to "`pp.Or(`*p1*`, `*p2*`)`".

p1* & *p2

Equivalent to "`pp.Each(`*p1*`, `*p2*`)`".

~ *p*

Equivalent to "`pp.NotAny(`*p*`)`".

Class `pp.ParserElement` also supports one static method:

pp.ParserElement.setDefaultWhitespaceChars(*s*)

This static method changes the definition of whitespace to be the characters in string *s*. Calling this method has this effect on all subsequent instantiations of any `pp.ParserElement` subclass.

```
>>> 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`: Sequence

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

The argument is a sequence of `ParseExpression` instances. The resulting parser matches a sequence of items that match those expressions, in exactly that order. You may also use the Python "`+`" operator to get this functionality. Here are some examples:

```
>>> 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`: Case-insensitive keyword match

```
pp.CaselessKeyword(matchString, identChars=I)
```

A variant of Section 5.14, "`Keyword`: Match a literal string not adjacent to specified context" (p. 23) that treats uppercase and lowercase characters the same.

5.4. `CaselessLiteral`: Case-insensitive string match

```
pp.CaselessLiteral(matchString)
```

The argument is a literal string to be matched. The resulting parser matches that string, except that it treats uppercase and lowercase characters the same.

The matched value will always have the same case as the *`matchString`* argument, not the case of the matched text.

```
>>> 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`: Match characters not in a given set

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

A parser of this class matches one or more characters that are *not* in the *`notChars`* argument. You may specify a minimum count of such characters using the `min` keyword argument, and you may specify a maximum count as the `max` argument. To create a parser that matches exactly *`N`* characters that are not in `notChars`, use the `exact=`*`N`* keyword argument.

```
>>> nonDigits = pp.CharsNotIn(pp.nums)
>>> print nonDigits.parseString('zoot86')
['zoot']
>>> fourNonDigits = pp.CharsNotIn(pp.nums, exact=4)
>>> print fourNonDigits.parseString('a$_/#')
['a$_/']
```

5.6. `Combine`: Fuse components together

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

The purpose of this class is to modify a parser containing several pieces so that the matching string will be returned as a single item in the returned `ParseResults` instance. The return value is another `ParserElement` instance that matches the same syntax as *`parser`*, but combines the pieces in the result.

`parser`
A parser, as a `ParserElement` instance.

`joinString`
A string that will be inserted between the pieces of the matched text when they are concatenated in the result.

`adjacent`
In the default case, `adjacent=True`, the text matched by components of the *`parser`* must be adjacent. If you pass `adjacent=False`, the result will match text containing the components of *`parser`* even if they are separated by other text.

```
>>> 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']
```

In the example above, `hiwayPieces` matches one or more letters (`pp.Word(pp.alphas)`) followed by one or more digits (`pp.Word(pp.nums)`). Because it has two components, a match on `hiwayPieces` will always return a list of two strings. The `hiway` parser returns a list containing one string, the concatenation of the matched pieces.

```
>>> 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`: A scanner for tables

```
pp.Dict(pattern)
```

The `Dict` class is a highly specialized pattern used to extract data from text arranged in rows and columns, where the first column contains labels for the remaining columns. The *`pattern`* argument must be a parser that describes a two-level structure such as a `Group` within a `Group`. Other group-like patterns such as the `delimitedList()` function may be used.

The constructor returns a parser whose `.parseString()` method will return a `ParseResults` instance like most parsers; however, in this case, the `ParseResults` instance can act like a dictionary whose keys are the row labels and each related value is a list of the other items in that row.

Here is an example.

catbird

```
#!/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])
```

Here is the output from that script:

```
result['bird']=['finch', 'verdin', 'siskin']
result['cat']=['Sandy', 'Mocha', 'Java']
```

5.8. `Each`: Require components in any order

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

This class returns a `ParserElement` that matches a given set of pieces, but the pieces may occur in any order. You may also construct this class using the "&" operator and the identity "`Each([p0, p1, p2, ...]) == p0 & p1 & p2 & ...`". Here is an example: a pattern that requires a string of letters and a string of digits, but they may occur in either order.

```
>>> 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`: Match empty content

```
pp.Empty()
```

The constructor returns a parser that always matches, and consumes no input. It can be used as a placeholder where a parser is required but you don't want it to match anything.

```
>>> 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`: Adding lookahead constraints

```
pp.FollowedBy(parser)
```

This class is used to specify a lookahead; that is, some content which must appear in the input, but you don't want to match any of it. Here is an example.

```
>>> 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)
```

The `name` pattern matches one or more letters; the `oneOrTwo` pattern matches either `'1'` or `'2'`; and the `number` pattern matches one or more digits. The expression "`pp.FollowedBy(oneOrTwo)`" requires that the next thing after the `name` matches the `oneOrTwo` pattern, but the input is not advanced past it.

Thus, the `number` pattern matches one or more digits, *including* the `'1'` or `'2'` just after the `name`. In the `'Robin88'` example, the match fails because the character just after `'Robin'` is neither `'1'` or `'2'`.

5.11. `Forward`: The parser placeholder

```
pp.Forward()
```

This one is going to a little complicated to explain.

There are certain parsing situations where you can't really write correct BNF that describes what is correctly structured and what is not. For example, consider the pattern "two copies of the same digit, one after the other." For example, the strings `'33'` and `'99'` match that description.

Writing a pattern that matches one digit is easy: "`pp.Word(pp.nums, exact=1)`" works perfectly well. And two of those in succession would be "`pp.Word(pp.nums, exact=1) + pp.Word(pp.nums, exact=1)`".

However, although that pattern matches `'33'`, it also matches `'37'`. So how would your script specify that both the pieces matched the same digit?

In *pyparsing*, we do this with an instance `Forward` class. Such an instance is basically an empty placeholder where a pattern will be added later, *during* execution of the parsing.

Let's look at a complete script that demonstrates use of the `Forward` pattern. For this example, we will reach back to ancient computing history for a feature of the early versions of the FORTRAN programming language: Hollerith string constants.

A Hollerith constant is a way to represent a string of characters. It consists of a count, followed by the letter `'H'`, followed by the number of characters specified by the count. Here are two examples, with their Python equivalents:

1HX	'X'
10H0123456789	'0123456789'

We'll write our pattern so that the `'H'` can be either uppercase or lowercase.

Here's the complete script. We start with the usual preliminaries: imports, some test strings, the main, and a function to run each test.

hollerith

```
#!/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))
```

Next we'll define the function `hollerith()` that returns a parse for a Hollerith string.

hollerith

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

def hollerith():
```

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

First we define a parser `intExpr` that matches the character count. It has a parse action that converts the number from character form to a Python `int`. The `lambda` expression defines a nameless function that takes a list of tokens and converts the first token to an `int`.

hollerith

```
    #--
    # Define a recognizer for the character count.
    #--
    intExpr = pp.Word(pp.nums).setParseAction(lambda t: int(t[0]))
```

Next we create an empty `Forward` parser as a placeholder for the logic that matches the `'H'` and the following characters.

hollerith

```
    #--
    # Allocate a placeholder for the rest of the parsing logic.
    #--
    stringExpr = pp.Forward()
```

Next we define a *closure*[19] that will be added to `intExpr` as a second parse action. Notice that we are defining a function within a function. The `countedParseAction` function will retain access to an external name (`stringExpr`, which is defined in the outer function's scope) after the function is defined.

hollerith

```
    #--
    # Define a closure that transfers the character count from
    # the intExpr to the stringExpr.
    #--
    def countedParseAction(toks):
        '''Closure to define the content of stringExpr.
        '''
```

The argument is the list of tokens that was recognized by `intExpr`; because of its parse action, this list contains the count as a single `int`.

hollerith

```
        n = toks[0]
```

The `contents` parser will match exactly `n` characters. We'll use Section 5.5, "`CharsNotIn`: Match characters not in a given set" (p. 16) to do this match, specifying the excluded characters as an empty string so that any character will be included. Incidentally, this does not for `n==0`, but `'0H'` is not a valid Hollerith literal. A more robust implementation would raise a `pp.ParseException` in this case.

hollerith

```
        #--
        # Create a parser for any (n) characters.
        #--
        contents = pp.CharsNotIn('', exact=n)
```

This next line inserts the final pattern into the placeholder parser: an `'H'` in either case followed by the `contents` pattern. The `'<<'` operator is overloaded in the `Forward` class to perform this operation: for any `Forward` recognizer *F* and any parser *p*, the expression "*F* `<<` *p*" modifies *F* so that it matches pattern *p*.

[19] http://en.wikipedia.org/wiki/Closure_(computer_science)

hollerith

```
        #--
        # Store a recognizer for 'H' + contents into stringExpr.
        #--
        stringExpr << (pp.Suppress(pp.CaselessLiteral('H')) + contents)
```

Parse actions may elect to modify the recognized tokens, but we don't need to do that, so we return `None` to signify that the tokens remain unchanged.

hollerith

```
        return None
```

That is the end of the `countedParseAction` closure. We are now back in the scope of `hollerith()`. The next line adds the closure as the second parse action for the `intExpr` parser.

hollerith

```
    #--
    # Add the above closure as a parse action for intExpr.
    #--
    intExpr.addParseAction(countedParseAction)
```

Now we are ready to return the completed `hollerith` parser: `intExpr` recognizes the count and `stringExpr` recognizes the `'H'` and string contents. When we return it, it is still just an empty `Forward`, but it will be filled in before it asked to parse.

hollerith

```
    #--
    # Return the completed pattern.
    #--
    return (pp.Suppress(intExpr) + stringExpr)

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

if __name__ == "__main__":
    main()
```

Here is the output of the script. Note that the last test fails because the `'999H'` is not followed by 999 more characters.

```
--- 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`: Advance to a specified position in the line

```
pp.GoToColumn(colNo)
```

This class returns a parser that causes the input position to advance to column number *`colNo`*, where column numbers are counted starting from 1. The value matched by this parser is the string of characters between the current position and position *`colNo`*. It is an error if the current position is past column *`colNo`*.

```
>>> 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']
```

In this example, `pat` is a parser with three parts. The first part matches one to four letters. The second part skips to column 5. The third part matches one or more digits.

In the first test, the `GoToColumn` parser returns `'@@'` because that was the text between the letters and column 5. In the second test, that parser returns the empty string because there are no characters between `'wxyz'` and `'987'`. In the third example, the part matched by the `GoToColumn` is empty because white space is ignored between tokens.

5.13. `Group`: Group repeated items into a list

```
pp.Group(parser)
```

This class causes the value returned from a match to be formed into a list. The *`parser`* argument is some parser that involves repeated tokens such as `ZeroOrMore` or a `delimitedList`.

```
>>> 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'], '}']
```

In the example above, both `pat1` and `pat2` match a sequence of words within `{braces}`. In the test of `pat1`, the result has five elements: the three words and the opening and closing braces. In the test of `pat2`, the result has three elements: the open brace, a list of the strings that matched the `OneOrMore` parser, and the closing brace.

5.14. `Keyword`: Match a literal string not adjacent to specified context

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

The *`matchString`* argument is a literal string. The resulting parser will match that exact text in the input. However, unlike the `Literal` class, the next input character must *not* be one of the characters in *`I`*. The default value of the `identChars` argument is a string containing all the letters and digits plus underbar ("`_`") and dollar sign ("`$`").

If you provide the keyword argument `caseless=True`, the match will be case-insensitive.

Examples:

```
>>> 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`: Match end of line

```
pp.LineEnd()
```

An instance of this class matches if the current position is at the end of a line or the end of a string. If it matches at the end of a line, it returns a newline (`'\n'`) in the result.

```
>>> 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']
```

In the next example, note that the end of the string does match the `pp.LineEnd()`, but in this case no value is added to the result.

```
>>> print an.parseString('Basingstoke')
['Basingstoke']
```

For more examples, see Section 7.12, "`lineEnd`: An instance of `LineEnd`" (p. 45).

5.16. `LineStart`: Match start of line

```
pp.LineStart()
```

An instance of this class matches if the current position is at the beginning of a line; that is, if it is either the beginning of the text or preceded by a newline. It does not advance the current position or contribute any content to the result.

Here are some examples. The first pattern matches a name at the beginning of a line.

```
>>> name = pp.Word(pp.alphas)
>>> sw = pp.LineStart() + name
>>> print sw.parseString('Dinsdale')
['Dinsdale']
```

The `ansb` pattern here matches a name followed by a newline followed by another name. Note that although there are four components, there are only three strings in the result; the `pp.LineStart()` does not contribute a result string.

```

>>> ansb = name + pp.LineEnd() + pp.LineStart() + name
>>> print ansb.parseString('Spiny\nNorman')
['Spiny', '\n', 'Norman']
```

Here is an example of `pp.LineStart()` failing to match.

```
>>> 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)
```

For more examples, see Section 7.13, "`lineStart`: An instance of `LineStart`" (p. 45).

5.17. `Literal`: Match a specific string

```
pp.Literal(text)
```

Matches the exact characters of the `text` argument. Here are some examples.

```
>>> 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`: Try multiple matches in a given order

```
pp.MatchFirst(parserList)
```

Use an instance of this class when you want to try to match two or more different parser, but you want to specify the order of the tests. The *`parserList`* argument is a list of parsers.

A typical place to use this is to match the text against a set of strings in which some strings are substrings of others. For example, suppose your input text has two different command names `CATCH` and `CAT`: you should test for `CATCH` first. If you test for `CAT` first, it will match the first three characters of `CATCH`, which is probably not what you want.

```
>>> 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']
```

You may also get the effect of this class by combining the component parsers with the "`|`" operator. The definition of `keySet` in the above example could also have been done this way:

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

5.19. `NoMatch`: A parser that never matches

```
pp.NoMatch()
```

This parser will always raise a `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)
```

This is similar to Section 5.10, "`FollowedBy`: Adding lookahead constraints" (p. 19) in that it looks to see if the current text position does *not* match something, but it does not advance that position. In this case the *`parser`* argument is any parser. The match succeeds if and only if the text at the current position does not match *`parser`*. The current position is not advanced whether `parser` matches it or not.

In the example below, the pattern matches a sequence of letters, followed by a sequence of digits provided the first one is not `'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`: Repeat a pattern one or more times

```
pp.OneOrMore(parser)
```

An instance of this class matches one or more repetitions of the syntax described by the *`parser`* argument.

```
>>> 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`: Match an optional pattern

```
pp.Optional(parser, default=D)
```

Use this pattern when a syntactic element is optional. The *`parser`* argument is a parser for the optional pattern. By default, if the pattern is not present, no content is added to the `ParseResult`; if you would like to supply content to be added in that case, provide it as the `default` keyword option.

```
>>> 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`: Parse one of a set of alternatives

```
pp.Or(parserList)
```

An instance of this class matches one of a given set of parsers; the argument is a sequence containing those parsers. If more than one of the parsers match, the parser used will be the one that matches the longest string of text.

```
>>> 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', '!']
```

You may also use the "^" operator to construct a set of alternatives. This line is equivalent to the third line of the example above:

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

5.24. `ParseException`

This is the exception thrown when the parse fails. These attributes are available on an instance:

`.lineno`
The line number where the parse failed, counting from 1.

`.col`
The column number where the parse failed, counting from 1.

`.line`
The text of the line in which the parse failed.

```
>>> 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`: Get me out of here!

If one of your parsers finds that it cannot go on, it can raise this exception to terminate parsing right away.

5.26. `ParseResults`: Result returned from a match

All parsers return an instance of this class. It can act like a list containing the strings that matched each piece of the corresponding parser. You may also use it like a dictionary, if you have used the `.setResultsName()` method on any of the component parsers to assign names to the text that matched that parser.

For an instance *R*, operations include:

***R*[*index*]**

To retrieve one of the matched tokens, you can treat the instance as a list.

You may also perform the usual list operations on a `ParseResult` such as replacing or deleting one of the values.

```
>>> 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']
```

If you have assigned names to any of the components of your parser, you can use the `ParseResults` instance as if it were a dictionary: the keys are the names, and each related value is the string that matched that component.

```
>>> 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'
```

Here are the methods available on a `ParseResults` instance.

***R*.asDict()**

This method returns the named items of *R* as a normal Python `dict`. Continuing the example above:

```
>>> r.asDict()
{'last': 'Piranha', 'first': 'Doug'}
```

***R*.asList()**

This method returns *R* as a normal Python `list`.

```
>>> r.asList()
['Doug', 'Piranha']
```

***R*.copy()**

Returns a copy of *R*.

.get(*key*, defaultValue=None)

Works like the `.get()` method on the standard Python `dict` type: if the `ParseResult` has no component named *`key`*, the `defaultValue` is returned.

```
>>> r.get('first', 'Unknown')
'Doug'
>>> r.get('middle', 'Unknown')
'Unknown'
```

.insert(*where*, *what*)

Like the `.insert()` method of the Python `list` type, this method will insert the value of the string *`what`* before position *`where`* in the list of strings.

```
>>> r.insert(1, 'Bubbles')
>>> print r
['Doug', 'Bubbles', 'Piranha']
```

***R*.items()**

This method works like the `.items()` method of Python's `dict` type, returning a list of tuples (*`key`*, *`value`*).

```
>>> r.items()
[('last', 'Piranha'), ('first', 'Doug')]
```

***R*.keys()**

Returns a list of the keys of named results. Continuing the Piranha example:

```
>>> r.keys()
['last', 'first']
```

5.27. `QuotedString`: Match a delimited string

```
pp.QuotedString(quoteChar, escChar=None, multiline=False,
    unquoteResults=True, endQuoteChar=None)
```

An instance of this class matches a string literal that is delimited by some quote character or characters.

quoteChar

This string argument defines the opening delimiter and, unless you pass an `endQuoteChar` argument, also the closing delimiter. The value may have multiple characters.

escChar
Strings may not normally include the closing quote character inside the string. To allow closing quote characters inside the string, pass an argument `escChar=c`, where *c* is an escape character that signifies that the following character is to be treated as text and not as a delimiter.

multiline
By default, a string may not include newline characters. If you want to allow the parser to match quoted strings that extend over multiple lines, pass an argument "`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: Match a regular expression

```
pp.Regex(r, flags=0)
```

An instance of this class matches a regular expression expressed in the form expected by the Python `re` module[20]. The argument *r* may be either a string containing a regular expression, or a compiled regular expression as an instance of `re.RegexObject`.

If the argument *r* is a string, you may provide a `flags` argument that will be passed to the `re.match()` function as its `flags` argument.

```
>>> r1 = '[a-e]+'
>>> pat1 = pp.Regex(r1)
>>> print pat1.parseString('aeebbaecd', parseAll=True)
['aeebbaecd']
```

[20] http://docs.python.org/2/library/re.html

```
>>> 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`: Search ahead for a pattern

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

An instance of this class will search forward in the input until it finds text that matches a parser *`target`*.

`include`
By default, when text matching the *`target`* pattern is found, the position is left at the beginning of that text. If you specify `include=True`, the position will be left at the end of the matched text, and the `ParseResult` will include a two-element list whose first element is the text that was skipped and the second element is the text that matched the *`target`* parser.

`ignore`
You can specify a pattern to be ignored while searching for the *`target`* by specifying an argument `ignore=`*`p`*, where *`p`* is a parser that matches the pattern to be ignored.

`failOn`
You can specify a pattern that must not be skipped over by passing an argument `failOn=`*`p`*, where *`p`* is a parser that matches that pattern. If you do this, the `SkipTo` parser will fail if it ever recognizes input that matches *`p`*.

```
>>> 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`: Match the end of the text

```
pp.StringEnd()
```

An instance of this class matches only if the text position is at the end of the string.

```
>>> 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`: Match the start of the text

```
pp.StringStart()
```

An instance of this class matches only if the text position is at the start of the string.

```
>>> 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`: Omit matched text from the result

```
pp.Suppress(p)
```

An instance of this class is a parser that matches the same content as parser *p*, but when it matches text, the matched text is not deposited into the returned `ParseResult` instance.

```
>>> 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']
```

See also the `.suppress()` method in Section 5.1, "`ParserElement`: The basic parser building block" (p. 11).

5.33. `Upcase`: Uppercase the result

```
pp.Upcase(p)
```

An instance of this class matches what parser *p* matches, but when the matching text is deposited in the returned `ParseResults` instance, all lowercase characters are converted to uppercase.

```
>>> name = pp.Upcase(pp.Word(pp.alphas))
>>> print name.parseString('ConfuseACat')
['CONFUSEACAT']
```

5.34. `White`: Match whitespace

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

An instance of this class matches one or more characters of whitespace.

`ws`
This string argument defines which characters are considered whitespace.

`min`
This argument defines the minimum number of characters that are required for a match.

`max`
This argument defines the maximum number of characters that will be matched.

`exact`
If specified, this number defines the exact number of whitespaces characters that will be matched.

```
>>> text = '   '
>>> print pp.White().parseString(text)
['   ']
>>> print pp.White(exact=1).parseString(text)
[' ']
>>> print pp.White(max=2).parseString(text)
['  ']
```

5.35. `Word`: Match characters from a specified set

```
pp.Word(initChars, bodyChars=None, min=1, max=0,
        exact=0, asKeyword=False, excludeChars=None)
```

An instance of this class will match multiple characters from a set of characters specified by the arguments.

`initChars`
If no `bodyChars` argument is given, this argument specifies all the characters that will be matched. If a `bodyChars` string is supplied, `initChars` specifies valid initial characters, and characters after the first that are in the `bodyChars` string will also match.

`bodyChars`
See `initChars`.

`min`
The minimum length to be matched.

`max`
The maximum length to be matched.

`exact`
If you supply this argument with a value of some number *n*, this parser will match exactly *n* characters.

`asKeyword`
By default, this parser will disregard the text following the matched part. If you specify `asKeyword=True`, the match will fail if the next character after the matched part is one of the matching characters (a character in `initChars` if there is no `bodyChars` argument, or a character in `bodyChars` if that keyword argument is present).

excludeChars

If supplied, this argument specifies characters *not* to be considered to match, even if those characters are otherwise considered to match.

```
>>> 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`: Match only at the end of a word

```
pp.WordEnd(wordChars=pp.printables)
```

An instance of this class matches only when the previous character (if there is one) is a word character and the next character is not a word character.

The optional `wordChars` argument specifies which characters are considered word characters; the default value is the set of all printable, non-whitespace characters.

```
>>> 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`: Match only at the start of a word

```
pp.WordStart(wordChars=pp.printables)
```

An instance of this class matches only when the current position is at the beginning of a word, and the previous character (if there is one) is *not* a word character.

The optional `wordChars` argument specifies which characters are considered word characters; the default value is the set of all printable, non-whitespace characters.

```
>>> 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`: Match any number of repetitions including none

```
pp.ZeroOrMore(p)
```

An instance of this class matches any number of text items, each of which matches parser *p*, even if there are no matching items.

```
>>> someWords = pp.ZeroOrMore(pp.Word(pp.alphas))
>>> print someWords.parseString('Comfidown Majorette')
['Comfidown', 'Majorette']
>>> print someWords.parseString('')
[]
>>> print someWords.parseString('   ')
[]
```

6. Functions

These functions are available in the *pyparsing* module.

6.1. `col()`: Convert a position to a column number

```
pp.col(loc, s)
```

The *loc* argument to this function is the location (Python index, counted from 0) of some position in a string *s*. The returned value is the column number of that position within its line, counting from 1. Newlines (`'\n'`) are treated as line separators.

```
>>> 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`: Parse *N* followed by *N* things

```
pp.countedArray(parser, intExpr=None)
```

This rather specialized function creates a parser that matches some count, followed by that many occurrences of a pattern matching some *`parser`*, like `"3 Moe Larry Curly"`. This function deposits a list of the values into the returned `ParseResults`, omitting the count itself. Note that the integers in this example are returned as type `str`, not type `int`.

```
>>> 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']]
```

If the count is for some reason not an integer in the usual form, you can provide an `intExpr` keyword argument that specifies a parser that will match the count and return it as a Python `int`.

```
>>> 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()`: Create a parser for a delimited list

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

This function creates a parser for a sequence *`P D P D ... D P`*, where *`P`* matches some *`parser`* and *`D`* is some delimiter, defaulting to "`,`".

By default, the result is a list of the *`P`* items with the delimiters (*`D`* items) omitted.

```
>>> text = "Arthur, Bedevere, Launcelot, Galahad, Robin"
>>> name = pp.Word(pp.alphas)
>>> nameList = pp.delimitedList(name)
>>> print nameList.parseString(text)
['Arthur', 'Bedevere', 'Launcelot', 'Galahad', 'Robin']
```

To include the delimiters and fuse the entire result into a single string, pass in the argument `combine=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']
```

The last example only matches one name because the `Combine` class suppresses the skipping of whitespace within its internal pieces.

6.4. `dictOf()`: Build a dictionary from key/value pairs

```
pp.dictOf(keyParser, valueParser)
```

This function builds a parser that matches a sequence of key text alternating with value text. When matched, this parser will deposit a dictionary-like value into the returned `ParseResults` with those keys and values.

The *`keyParser`* argument is a parser that matches the key text and the *`valueParser`* is a parser that matches the value text.

Here is a very simple example to give you the idea. The text to be matched is a sequence of five-character items, each of which is a one-letter color code followed by a four-character color name.

```
>>> 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'
```

Here's a slightly more subtle example. The text has the form "*`degree`*`: `*`name`*`; ...`", where the *`degree`* part is the degree of the musical scale as a number, and the *`name`* part is the name of that note. Here's a first attempt.

```
>>> 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'
```

Note that the last key-value pair is missing. This is because the `value` pattern requires a trailing semicolon, and the text string does not end with one of those. Unless you were careful to check your work, you might not notice that the last item is missing. This is one reason that it is good practice always to use the `parseAll=True` option when calling `.parseString()`. Notice how that reveals the error:

```
>>> noteNames = notePat.parseString(text, parseAll=True)
pyparsing.ParseException: Expected end of text (at char 43), (line:1,
col:44)
```

It's easy enough to fix the definition of the `text`, but instead let's fix the parser so that it defines `value` as ending either with a semicolon or with the end of the string:

```
>>> 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()`: Lowercasing parse action

If you use this function as a parse action, the effect will be that all the letters in the values that the parser returns in its `ParseResults` will be lowercase.

```
>>> sameName = pp.Word(pp.alphas)
>>> print sameName.parseString('SpringSurprise')
['SpringSurprise']
>>> lowerName = sameName.setParseAction(pp.downcaseTokens)
>>> print lowerName.parseString('SpringSurprise')
['springsurprise']
```

6.6. `getTokensEndLoc()`: Find the end of the tokens

If used within a parse action, this function takes no arguments and returns the location just after its tokens, counting from 0.

```
>>> def findEnd(s, loc, toks):
...     print pp.getTokensEndLoc()
...
>>> letters = pp.Word(pp.alphas).setParseAction(findEnd)
>>> print letters.parseString('pepperpot')
9
['pepperpot']
```

6.7. `line()`: In what line does a location occur?

```
pp.line(loc, text)
```

Given a string *`text`* and a location *`loc`* (Python index) within that string, this function returns the line containing that location, without a line terminator.

```
>>> 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()`: Convert a position to a line number

```
pp.lineno(loc, s)
```

The *`loc`* argument to this function is the location (Python index, counted from 0) of some position in a string *`s`*. The returned value is the line number of that position, counting from 1. Newlines (`'\n'`) are treated as line separators. For an example demonstrating this function, see Section 6.1, "`col()`: Convert a position to a column number" (p. 35).

6.9. `matchOnlyAtCol()`: Parse action to limit matches to a specific column

```
pp.matchOnlyAtCol(col)
```

Use this function as a parse action to force a parser to match only at a specific column number within the line, counting from 1.

```
>>> 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()`: Match the text that the preceding expression matched

```
pp.matchPreviousExpr(parser)
```

This function returns a new parser that matches not only the same *pattern* as the given *`parser`*, but it matches the *value* that was matched by *`parser`*.

```
>>> 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)
```

The last example above failed because, even though the string `"no"` occurred both before and after the hyphen, the `name2` parser matched the entire string `"now"` before it tested to see if it matched the previous occurrence `"no"`. Compare the behavior of Section 6.11, "`matchPreviousLiteral()`: Match the literal text that the preceding expression matched" (p. 40).

6.11. `matchPreviousLiteral()`: Match the literal text that the preceding expression matched

```
pp.matchPreviousLiteral(parser)
```

This function works like the one described in Section 6.10, "`matchPreviousExpr()`: Match the text that the preceding expression matched" (p. 39), except that the returned parser matches the exact characters that *`parser`* matched, without regard for any following context. Compare the example below with the one in Section 6.10, "`matchPreviousExpr()`: Match the text that the preceding expression matched" (p. 39).

```
>>> 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)
```

6.12. `nestedExpr()`: Parser for nested lists

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

This function returns a parser that matches text that is structured as a nested list, that is, as a sequence *`LCR`* where:

- The `opener` argument *`L`* is some opening delimiter string, defaulting to "`(`".
- The `closer` argument *`R`* is some closing delimiter string, defaulting to "`(`".
- The `content` argument *`C`* is some content that can occur between these two delimiters. Anywhere in this content, another level of the *`LCR`* sequence may occur any number of times. If you don't specify a `content` argument, the corresponding value deposited into the returned `ParseResults` will be a list of the strings at each level that consist of non-whitespace groups separated by whitespace.
- 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 process will not treat those occurrences as delimiters. The default value *`I`* is an instance of Section 5.27, "`QuotedString`: Match a delimited string" (p. 29). If you 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()`: Check for multiple literals, longest first

```
pp.oneOf(alternatives, caseless=False)
```

This function returns a parser that matches one of a set of literals. In particular, if any literal is a substring of another, this parser will always check for the longer one first; this behavior is useful, for example, when you are parsing a set of keywords.

- The `alternatives` argument specifies the different literals that the parser will match. This may be either a list of strings, or one string with the alternatives separated by spaces.
- By default, the match will be case-sensitive. To specify a case-insensitive match, pass the argument "`caseless=True`".

```
>>> keyList = pp.oneOf('as assert break')
>>> print keyList.parseString('assert yes')
['assert']
>>> print keyList.parseString('as read')
['as']
```

6.14. `srange()`: Specify ranges of characters

```
pp.srange("[ranges]")
```

Use this function to create a string that you can pass to `pp.Word()` to create a parser that will match any one of a specified sets of characters. The argument allows you to use ranges of character codes so that you don't have to specify every single character. The syntax of the argument is similar to the "`[...]`" construct of general-purpose regular expressions.

The *`ranges`* argument string consists of one or more occurrences of:

- Single characters.
- A backslash followed by a single character, so that your parser can match characters such as `'-'` (as `'\-'`) or `']'` (as `'\]'`).
- A character specified by its hexadecimal character code as `'\x`*`HH`*`'`.
- A character specified by its octal character code as `'\0`*`N`*`...'`, where *`N`* can be one, two or three octal digits.
- Two of the above choices separated by `'-'`, meaning all the characters with codes between those values, including the endpoints. For example, `pp.srange('[a-z]')` will return a parser that will match any lowercase letter.

Here's an example that demonstrates the use of this function in creating a parser for a Python identifier.

```
>>> 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()`: Strip leading trailing quotes

To remove the first and last characters of the matched text from the result, use this function as a parse action.

```
>>> 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()`: Substitute a constant value for the matched text

```
pp.replaceWith(literal)
```

If you attach this function to a parser as a parse action, when the parser matches some text, the value that will be deposited in the `ParseResults` will be the *`literal`* string value.

```
>>> password = pp.Word(pp.printables).setParseAction(
...     pp.replaceWith('********'))
>>> print password.parseString('shazam')
['********']
>>> print password.parseString('abracadabra')
['********']
```

6.17. `traceParseAction()`: Decorate a parse action with trace output

You can use this decorator to wrap a parse action, so that whenever the parse action is used, two messages will appear on the `sys.stderr` stream, showing the arguments on entry and the value returned.

```
>>> @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()`: Uppercasing parse action

If you use this function as a parse action, the effect will be that all the letters in the values that the parser returns in its `ParseResults` will be uppercase.

```
>>> sameName = pp.Word(pp.alphas)
>>> print sameName.parseString('SpringSurprise')
['SpringSurprise']
>>> upperName = pp.Word(pp.alphas).setParseAction(pp.upcaseTokens)
>>> print upperName.parseString('SpringSurprise')
['SPRINGSURPRISE']
```

7. Variables

These variables are defined in the *pyparsing* module.

7.1. `alphanums`: The alphanumeric characters

The variable `pp.alphanums` is a string containing all lowercase and uppercase letters and decimal digits.

7.2. `alphas`: The letters

The variable `pp.alphas` is a string containing all lowercase and uppercase letters.

7.3. `alphas8bit`: Supplement Unicode letters

The variable `pp.alphas8bit` is a Unicode string containing the characters in the Latin-1 Supplement (range U000C0-U000FF) that are considered alphabetic. For more information, see *The ISO 9573-2003 Unicode entity group*[21].

7.4. `cStyleComment`: Match a C-language comment

The variable `pp.cStyleComment` is a parser that matches comments in the C language.

```
>>> print pp.cStyleComment.parseString('''/* First line.
...    Second line.
...    Third line. */''')
['/* First line.\n   Second line.\n   Third line. */']
```

7.5. `commaSeparatedList`: Parser for a comma-separated list

The variable `pp.commaSeparatedList` is a parser that matches any text, and breaks it into pieces wherever there is a comma (“,”).

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

[21] http://www.nmt.edu/tcc/help/pubs/docbook43/iso9573/

7.6. `cppStyleComment`: Parser for C++ comments

The variable `pp.cppStyleComment` is a parser that matches a comment in the C++ language. Comments may be either the old C style (see Section 7.4, "`cStyleComment`: Match a C-language comment" (p. 43)) or the newer C++ style (see Section 7.8, "`dblSlashComment`: Parser for a comment that starts with "`//`"" (p. 44)).

```
>>> 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`: String enclosed in `"..."`

The variable `pp.dblQuotedString` is a parser that matches a string enclosed in double-quote ("`"`") characters. The sequence "`\"`" within the text is *not* interpreted as an internal double-quote character.

```
>>> text = '"Gannet on a stick"'
>>> print pp.dblQuotedString.parseString(text)
['"Gannet on a stick"']
```

7.8. `dblSlashComment`: Parser for a comment that starts with "`//`"

The variable `pp.dblSlashComment` is a parser that matches comments in C++ or Java programs that start with "`//`" and extend to the end of the line.

```
>>> text = '// Comment\nNext line'
>>> print pp.dblSlashComment.parseString(text)
['// Comment']
```

7.9. `empty`: Match empty content

Variable `pp.empty` is an instance of Section 5.9, "`Empty`: Match empty content" (p. 18).

```
>>> print pp.empty.parseString('Matches nothing')
[]
```

7.10. `hexnums`: All hex digits

The variable `pp.hexnums` is a string containing all valid hexadecimal characters: the ten decimal digits plus the letters A through F in upper- and lowercase.

7.11. `javaStyleComment`: Comments in Java syntax

The variable `pp.javaStyleComment` matches a comment in one of two styles, either Section 7.4, "`cStyleComment`: Match a C-language comment" (p. 43) or Section 7.6, "`cppStyleComment`: Parser for C++ comments" (p. 44). Note that the "`//`" style comment always ends at the end of the line.

```
>>> print pp.javaStyleComment.parseString('''/*
... multiline comment */''')
['/*\nmultiline comment */']
>>> print pp.javaStyleComment.parseString('''// This comment
... intentionally left almost blank\n''')
['// This comment']
```

7.12. `lineEnd`: An instance of `LineEnd`

The variable `pp.lineEnd` is an instance of Section 5.15, "`LineEnd`: Match end of line" (p. 24).

```
>>> print pp.lineEnd.parseString('\nThis remains unparsed')
['\n']
```

7.13. `lineStart`: An instance of `LineStart`

The variable `pp.lineStart` is an instance of Section 5.16, "`LineStart`: Match start of line" (p. 24).

In this example, pattern `initialWord` matches a string of letters at the beginning of a line. The sample text matches because the beginning of the string is considered the beginning of a line.

```
>>> initialWord = pp.lineStart + pp.Word(pp.alphas)
>>> print initialWord.parseString('Silly party\n')
['Silly']
```

In the next example, pattern `pat1` matches a word, followed by the end of a line, followed by pattern `initialWord`. The first sample text matches because the position just after a `'\n'` is considered the start of a line.

```
>>> pat1 = pp.Word(pp.alphas) + pp.lineEnd + initialWord
>>> print pat1.parseString('Very\nSilly Party')
['Very', '\n', 'Silly']
```

This example fails to match. The first part of `pat2` matches the word `'Harpenden'`; automatic blank skipping moves to the beginning of the word `'southeast'`; and then the `pp.lineStart` parser fails because the space before `'southeast'` is not a newline or the beginning of the string.

```
>>> 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`: The decimal digits

The variable `pp.nums` is a string containing the ten decimal digits.

```
>>> pp.nums
'0123456789'
```

7.15. `printables`: All the printable non-whitespace characters

The variable `pp.printables` is a `str` containing all the printable, non-whitespace characters. Of the 128 7-bit ASCII characters, the first 32 are control characters; counting space and *DEL*, this leaves 94 printables.

```
>>> len(pp.printables)
94
>>> print pp.printables
0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&'()*+,-
./:;<=>?@[\]^_`{|}~
```

7.16. `punc8bit`: Some Unicode punctuation marks

The variable `pp.punc8bit` is a `unicode` string containing a number of characters that are considered punctuation marks: specifically, code points U000A1-U000BF plus U000D7 (×) and 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`: Comments in the style of the Python language

The variable `pp.pythonStyleComments` is a parser that matches comments in the style of the Python programming language: a comment begins with "`#`" and ends at the end of the line (or the end of the entire text string, whichever comes first).

```
>>> 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`: Parser for a default quoted string

For the common case of a literal string that is bounded by either double (`"..."`) or single (`'...'`) quotes, the variable `pp.quotedString` is an instance of Section 5.27, "`QuotedString`: Match a delimited string" (p. 29) that uses that specification for the quote characters.

```
>>> print pp.quotedString.parseString('"Ftang ftang"')
['"Ftang ftang"']
>>> print pp.quotedString.parseString("'Ole Biscuitbarrel'")
["'Ole Biscuitbarrel'"]
```

Note that the quotes are returned as part of the result. If you don't like that, you can attach the parse action described in Section 6.15, "`removeQuotes()`: Strip leading trailing quotes" (p. 42).

```
>>> justTheGuts = pp.quotedString.addParseAction(pp.removeQuotes)
>>> print justTheGuts.parseString("'Kevin Phillips Bong'")
['Kevin Phillips Bong']
```

7.19. `restOfLine`: Match the rest of the current line

The variable `pp.restOfLine` is a parser that matches zero or more characters up to, but not including, the next newline character (or the end of the string, whichever comes first).

```
>>> text = 'Wolverhampton 3\nBristol nought\n'
>>> print pp.restOfLine.parseString(text)
['Wolverhampton 3']
```

To match the rest of the line including the newline character at the end (if there is one), combine this with the parser described in Section 7.12, "`lineEnd`: An instance of `LineEnd`" (p. 45).

```
>>> toNextLine = pp.Combine(pp.restOfLine + pp.lineEnd)
>>> print toNextLine.parseString(text)
['Wolverhampton 3\n']
```

7.20. `sglQuotedString`: String enclosed in `'...'`

The variable `pp.sglQuotedString` is a parser that matches a string enclosed in single-quote ("`'`") characters. The sequence "`\'`" within the text is *not* interpreted as an internal single-quote character.

```
>>> 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'
```

If internal `"\'"` sequences were interpreted as escapes, the last line above would have displayed as:

```
"Don't"
```

7.21. `stringEnd`: Matches the end of the string

The variable `pp.stringEnd` contains an instance of Section 5.30, "`StringEnd`: Match the end of the text" (p. 31).

```
>>> 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`: Match a Python-style Unicode string

The variable `pp.unicodeString` is a parser that matches Python Unicode string literals: letter `'u'`, followed by a string in either single or double quote characters. This parser does not support triply-quoted strings.

```
>>> 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"""']
```