PyParsing Documentation

Release 2.4.2

Paul T. McGuire

Nov 04, 2019

Contents:

Release v2.4.2

CHAPTER 1

1 Using the pyparsing module

author Paul McGuire

address ptmcg@users.sourceforge.net

revision 2.0.1a

date July, 2013 (minor update August, 2018)

abstract This document provides how-to instructions for the pyparsing library, an easy-to-use Python module for constructing and executing basic text parsers. The pyparsing module is useful for evaluating user-definable expressions, processing custom application language commands, or extracting data from formatted reports.

Contents

Note: While this content is still valid, there are more detailed descriptions and examples at the online doc server at https://pythonhosted.org/pyparsing/pyparsing-module.html

1.1 1.1 Steps to follow

To parse an incoming data string, the client code must follow these steps:

1. First define the tokens and patterns to be matched, and assign this to a program variable. Optional results names or parsing actions can also be defined at this time.
2. Call `parseString()` or `scanString()` on this variable, passing in the string to be parsed. During the matching process, whitespace between tokens is skipped by default (although this can be changed). When token matches occur, any defined parse action methods are called.
3. Process the parsed results, returned as a list of strings. Matching results may also be accessed as named attributes of the returned results, if names are defined in the definition of the token pattern, using `setResultsName()`.

1.1.1 1.1.1 Hello, World!

The following complete Python program will parse the greeting "Hello, World!", or any other greeting of the form "<salutation>, <addressee>!":

```
from pyparsing import Word, alphas

greet = Word(alphas) + "," + Word(alphas) + "!"
greeting = greet.parseString("Hello, World!")
print greeting
```

The parsed tokens are returned in the following form:

```
['Hello', ',', 'World', '!']
```

1.1.2 1.1.2 Usage notes

- The pyparsing module can be used to interpret simple command strings or algebraic expressions, or can be used to extract data from text reports with complicated format and structure ("screen or report scraping"). However, it is possible that your defined matching patterns may accept invalid inputs. Use pyparsing to extract data from strings assumed to be well-formatted.
- To keep up the readability of your code, use *operators* such as +, |, ^, and ~ to combine expressions. You can also combine string literals with ParseExpressions - they will be automatically converted to Literal objects. For example:

```
integer  = Word(nums)             # simple unsigned integer
variable = Word(alphas, max=1)    # single letter variable, such as x, z, m, etc.
arithOp  = Word("+-*/", max=1)    # arithmetic operators
equation = variable + "=" + integer + arithOp + integer    # will match "x=2+2",␣
↪etc.
```

In the definition of `equation`, the string `"="` will get added as a `Literal("=")`, but in a more readable way.

- The pyparsing module's default behavior is to ignore whitespace. This is the case for 99% of all parsers ever written. This allows you to write simple, clean, grammars, such as the above `equation`, without having to clutter it up with extraneous `ws` markers. The `equation` grammar will successfully parse all of the following statements:

```
x=2+2
x = 2+2
a = 10   *   4
r= 1234/ 100000
```

Of course, it is quite simple to extend this example to support more elaborate expressions, with nesting with parentheses, floating point numbers, scientific notation, and named constants (such as `e` or `pi`). See `fourFn.py`, included in the examples directory.

- To modify pyparsing's default whitespace skipping, you can use one or more of the following methods:
  - use the static method `ParserElement.setDefaultWhitespaceChars` to override the normal set of whitespace chars (' tn'). For instance when defining a grammar in which newlines are significant, you should call `ParserElement.setDefaultWhitespaceChars(' \t')` to remove newline from the set of skippable whitespace characters. Calling this method will affect all pyparsing expressions defined afterward.
  - call `leaveWhitespace()` on individual expressions, to suppress the skipping of whitespace before trying to match the expression
  - use `Combine` to require that successive expressions must be adjacent in the input string. For instance, this expression:

    ```
    real = Word(nums) + '.' + Word(nums)
    ```

    will match "3.14159", but will also match "3 . 12". It will also return the matched results as ['3', '.', '14159']. By changing this expression to:

    ```
    real = Combine(Word(nums) + '.' + Word(nums))
    ```

    it will not match numbers with embedded spaces, and it will return a single concatenated string '3.14159' as the parsed token.

- Repetition of expressions can be indicated using `*` or `[]` notation. An expression may be multiplied by an integer value (to indicate an exact repetition count), or indexed with a tuple, representing min and max repetitions (with `...` representing no min or no max, depending whether it is the first or second tuple element). See the following examples, where n is used to indicate an integer value:
  - `expr*3` is equivalent to `expr + expr + expr`
  - `expr[2, 3]` is equivalent to `expr + expr + Optional(expr)`
  - `expr[n, ...]` or `expr[n,]` is equivalent to `expr*n + ZeroOrMore(expr)` (read as "at least n instances of expr")
  - `expr[... ,n]` is equivalent to `expr*(0, n)` (read as "0 to n instances of expr")

  - `expr[...]` and `expr[0, ...]` are equivalent to `ZeroOrMore(expr)`
  - `expr[1, ...]` is equivalent to `OneOrMore(expr)`

  Note that `expr[..., n]` does not raise an exception if more than n exprs exist in the input stream; that is, `expr[..., n]` does not enforce a maximum number of expr occurrences. If this behavior is desired, then write `expr[..., n] + ~expr`.

- `MatchFirst` expressions are matched left-to-right, and the first match found will skip all later expressions within, so be sure to define less-specific patterns after more-specific patterns. If you are not sure which expressions are most specific, use Or expressions (defined using the ^ operator) - they will always match the longest expression, although they are more compute-intensive.
- `Or` expressions will evaluate all of the specified subexpressions to determine which is the "best" match, that is, which matches the longest string in the input data. In case of a tie, the left-most expression in the `Or` list will win.
- If parsing the contents of an entire file, pass it to the `parseFile` method using:

```
expr.parseFile(sourceFile)
```

- `ParseExceptions` will report the location where an expected token or expression failed to match. For example, if we tried to use our "Hello, World!" parser to parse "Hello World!" (leaving out the separating comma), we would get an exception, with the message:

```
pyparsing.ParseException: Expected "," (6), (1,7)
```

  In the case of complex expressions, the reported location may not be exactly where you would expect. See more information under *ParseException* .

- Use the `Group` class to enclose logical groups of tokens within a sublist. This will help organize your results into more hierarchical form (the default behavior is to return matching tokens as a flat list of matching input strings).
- Punctuation may be significant for matching, but is rarely of much interest in the parsed results. Use the `suppress()` method to keep these tokens from cluttering up your returned lists of tokens. For example, `delimitedList()` matches a succession of one or more expressions, separated by delimiters (commas by default), but only returns a list of the actual expressions - the delimiters are used for parsing, but are suppressed from the returned output.
- Parse actions can be used to convert values from strings to other data types (ints, floats, booleans, etc.).
- Results names are recommended for retrieving tokens from complex expressions. It is much easier to access a token using its field name than using a positional index, especially if the expression contains optional elements. You can also shortcut the `setResultsName` call:

```
stats = ("AVE:" + realNum.setResultsName("average")
         + "MIN:" + realNum.setResultsName("min")
         + "MAX:" + realNum.setResultsName("max"))
```

  can now be written as this:

```
stats = ("AVE:" + realNum("average")
         + "MIN:" + realNum("min")
         + "MAX:" + realNum("max"))
```

- Be careful when defining parse actions that modify global variables or data structures (as in `fourFn.py`), especially for low level tokens or expressions that may occur within an `And` expression; an early element of an `And` may match, but the overall expression may fail.

1.2 1.2 Classes

1.2.1 1.2.1 Classes in the pyparsing module

`ParserElement` - abstract base class for all pyparsing classes; methods for code to use are:

- `parseString(sourceString, parseAll=False)` - only called once, on the overall matching pattern; returns a *ParseResults* object that makes the matched tokens available as a list, and optionally as a dictionary, or as an object with named attributes; if parseAll is set to True, then parseString will raise a ParseException if the grammar does not process the complete input string.
- `parseFile(sourceFile)` - a convenience function, that accepts an input file object or filename. The file contents are passed as a string to `parseString()`. `parseFile` also supports the `parseAll` argument.
- `scanString(sourceString)` - generator function, used to find and extract matching text in the given source string; for each matched text, returns a tuple of:
  - matched tokens (packaged as a *ParseResults* object)
  - start location of the matched text in the given source string
  - end location in the given source string

  `scanString` allows you to scan through the input source string for random matches, instead of exhaustively defining the grammar for the entire source text (as would be required with `parseString`).
- `transformString(sourceString)` - convenience wrapper function for `scanString`, to process the input source string, and replace matching text with the tokens returned from parse actions defined in the grammar (see *setParseAction*).
- `searchString(sourceString)` - another convenience wrapper function for `scanString`, returns a list of the matching tokens returned from each call to `scanString`.
- `setName(name)` - associate a short descriptive name for this element, useful in displaying exceptions and trace information
- `setResultsName(string, listAllMatches=False)` - name to be given to tokens matching the element; if multiple tokens within a repetition group (such as `ZeroOrMore` or `delimitedList`) the default is to return only the last matching token - if listAllMatches is set to True, then a list of all the matching tokens is returned. (New in 1.5.6 - a results name with a trailing '*' character will be interpreted as setting listAllMatches to True.) Note: `setResultsName` returns a *copy* of the element so that a single basic element can be referenced multiple times and given different names within a complex grammar.
- `setParseAction(*fn)` - specify one or more functions to call after successful matching of the element; each function is defined as `fn(s, loc, toks)`, where:
  - `s` is the original parse string
  - `loc` is the location in the string where matching started
  - `toks` is the list of the matched tokens, packaged as a *ParseResults* object

  Multiple functions can be attached to a ParserElement by specifying multiple arguments to setParseAction, or by calling setParseAction multiple times.

  Each parse action function can return a modified `toks` list, to perform conversion, or string modifications. For brevity, `fn` may also be a lambda - here is an example of using a parse action to convert matched integer tokens from strings to integers:

```
intNumber = Word(nums).setParseAction(lambda s,l,t: [int(t[0])])
```

  If `fn` does not modify the `toks` list, it does not need to return anything at all.

- `setBreak(breakFlag=True)` - if breakFlag is True, calls pdb.set_break() as this expression is about to be parsed
- `copy()` - returns a copy of a ParserElement; can be used to use the same parse expression in different places in a grammar, with different parse actions attached to each
- `leaveWhitespace()` - change default behavior of skipping whitespace before starting matching (mostly used internally to the pyparsing module, rarely used by client code)
- `setWhitespaceChars(chars)` - define the set of chars to be ignored as whitespace before trying to match a specific ParserElement, in place of the default set of whitespace (space, tab, newline, and return)
- `setDefaultWhitespaceChars(chars)` - class-level method to override the default set of whitespace chars for all subsequently created ParserElements (including copies); useful when defining grammars that treat one or more of the default whitespace characters as significant (such as a line-sensitive grammar, to omit newline from the list of ignorable whitespace)
- `suppress()` - convenience function to suppress the output of the given element, instead of wrapping it with a Suppress object.
- `ignore(expr)` - function to specify parse expression to be ignored while matching defined patterns; can be called repeatedly to specify multiple expressions; useful to specify patterns of comment syntax, for example
- `setDebug(dbgFlag=True)` - function to enable/disable tracing output when trying to match this element
- `validate()` - function to verify that the defined grammar does not contain infinitely recursive constructs
- `parseWithTabs()` - function to override default behavior of converting tabs to spaces before parsing the input string; rarely used, except when specifying whitespace-significant grammars using the *White* class.
- `enablePackrat()` - a class-level static method to enable a memoizing performance enhancement, known as "packrat parsing". packrat parsing is disabled by default, since it may conflict with some user programs that use parse actions. To activate the packrat feature, your program must call the class method ParserElement.enablePackrat(). For best results, call enablePackrat() immediately after importing pyparsing.

1.2.2 1.2.2 Basic ParserElement subclasses

- `Literal` - construct with a string to be matched exactly
- `CaselessLiteral` - construct with a string to be matched, but without case checking; results are always returned as the defining literal, NOT as they are found in the input string
- `Keyword` - similar to Literal, but must be immediately followed by whitespace, punctuation, or other non-keyword characters; prevents accidental matching of a non-keyword that happens to begin with a defined keyword
- `CaselessKeyword` - similar to Keyword, but with caseless matching behavior
- `Word` - one or more contiguous characters; construct with a string containing the set of allowed initial characters, and an optional second string of allowed body characters; for instance, a common Word construct is to match a code identifier - in C, a valid identifier must start with an alphabetic character or an underscore ('_'), followed by a body that can also include numeric digits. That is, `a`, `i`, `MAX_LENGTH`, `_a1`, `b_109_`, and `plan9FromOuterSpace` are all valid identifiers; `9b7z`, `$a`, `.section`, and `0debug` are not. To define an identifier using a Word, use either of the following:

```
- Word(alphas+"_", alphanums+"_")
- Word(srange("[a-zA-Z_]"), srange("[a-zA-Z0-9_]"))
```

If only one string given, it specifies that the same character set defined for the initial character is used for the word body; for instance, to define an identifier that can only be composed of capital letters and underscores, use:

```
- Word("ABCDEFGHIJKLMNOPQRSTUVWXYZ_")
- Word(srange("[A-Z_]"))
```

A Word may also be constructed with any of the following optional parameters:

- `min` - indicating a minimum length of matching characters
- `max` - indicating a maximum length of matching characters
- `exact` - indicating an exact length of matching characters

If `exact` is specified, it will override any values for `min` or `max`.

New in 1.5.6 - Sometimes you want to define a word using all characters in a range except for one or two of them; you can do this with the new `excludeChars` argument. This is helpful if you want to define a word with all printables except for a single delimiter character, such as '.'. Previously, you would have to create a custom string to pass to Word. With this change, you can just create `Word(printables, excludeChars='.')`.

- `CharsNotIn` - similar to *Word*, but matches characters not in the given constructor string (accepts only one string for both initial and body characters); also supports `min`, `max`, and `exact` optional parameters.
- `Regex` - a powerful construct, that accepts a regular expression to be matched at the current parse position; accepts an optional `flags` parameter, corresponding to the flags parameter in the re.compile method; if the expression includes named sub-fields, they will be represented in the returned *ParseResults*
- `QuotedString` - supports the definition of custom quoted string formats, in addition to pyparsing's built-in `dblQuotedString` and `sglQuotedString`. `QuotedString` allows you to specify the following parameters:
  - `quoteChar` - string of one or more characters defining the quote delimiting string
  - `escChar` - character to escape quotes, typically backslash (default=None)
  - `escQuote` - special quote sequence to escape an embedded quote string (such as SQL's "" to escape an embedded ") (default=None)
  - `multiline` - boolean indicating whether quotes can span multiple lines (default=False)
  - `unquoteResults` - boolean indicating whether the matched text should be unquoted (default=True)
  - `endQuoteChar` - string of one or more characters defining the end of the quote delimited string (default=None => same as quoteChar)
- `SkipTo` - skips ahead in the input string, accepting any characters up to the specified pattern; may be constructed with the following optional parameters:
  - `include` - if set to true, also consumes the match expression (default is false)
  - `ignore` - allows the user to specify patterns to not be matched, to prevent false matches
  - `failOn` - if a literal string or expression is given for this argument, it defines an expression that should cause the `SkipTo` expression to fail, and not skip over that expression
- `White` - also similar to *Word*, but matches whitespace characters. Not usually needed, as whitespace is implicitly ignored by pyparsing. However, some grammars are whitespace-sensitive, such as those that use leading tabs or spaces to indicating grouping or hierarchy. (If matching on tab characters, be sure to call *parseWithTabs* on the top-level parse element.)
- `Empty` - a null expression, requiring no characters - will always match; useful for debugging and for specialized grammars
- `NoMatch` - opposite of Empty, will never match; useful for debugging and for specialized grammars

1.2.3 1.2.3 Expression subclasses

- `And` - construct with a list of ParserElements, all of which must match for And to match; can also be created using the '+' operator; multiple expressions can be Anded together using the '*' operator as in:

```
ipAddress = Word(nums) + ('.' + Word(nums)) * 3
```

A tuple can be used as the multiplier, indicating a min/max:

```
usPhoneNumber = Word(nums) + ('-' + Word(nums)) * (1,2)
```

A special form of `And` is created if the '-' operator is used instead of the '+' operator. In the ipAddress example above, if no trailing '.' and Word(nums) are found after matching the initial Word(nums), then pyparsing will back up in the grammar and try other alternatives to ipAddress. However, if ipAddress is defined as:

```
strictIpAddress = Word(nums) - ('.'+Word(nums))*3
```

then no backing up is done. If the first Word(nums) of strictIpAddress is matched, then any mismatch after that will raise a ParseSyntaxException, which will halt the parsing process immediately. By careful use of the '-' operator, grammars can provide meaningful error messages close to the location where the incoming text does not match the specified grammar.

- `Or` - construct with a list of ParserElements, any of which must match for Or to match; if more than one expression matches, the expression that makes the longest match will be used; can also be created using the '^' operator
- `MatchFirst` - construct with a list of ParserElements, any of which must match for MatchFirst to match; matching is done left-to-right, taking the first expression that matches; can also be created using the 'l' operator
- `Each` - similar to And, in that all of the provided expressions must match; however, Each permits matching to be done in any order; can also be created using the '&' operator
- `Optional` - construct with a ParserElement, but this element is not required to match; can be constructed with an optional `default` argument, containing a default string or object to be supplied if the given optional parse element is not found in the input string; parse action will only be called if a match is found, or if a default is specified
- `ZeroOrMore` - similar to Optional, but can be repeated
- `OneOrMore` - similar to ZeroOrMore, but at least one match must be present
- `FollowedBy` - a lookahead expression, requires matching of the given expressions, but does not advance the parsing position within the input string
- `NotAny` - a negative lookahead expression, prevents matching of named expressions, does not advance the parsing position within the input string; can also be created using the unary '~' operator

1.2.4 1.2.4 Expression operators

- ~ - creates NotAny using the expression after the operator
- + - creates And using the expressions before and after the operator
- | - creates MatchFirst (first left-to-right match) using the expressions before and after the operator
- ^ - creates Or (longest match) using the expressions before and after the operator
- & - creates Each using the expressions before and after the operator

- * - creates And by multiplying the expression by the integer operand; if expression is multiplied by a 2-tuple, creates an And of (min,max) expressions (similar to "{min,max}" form in regular expressions); if min is None, intepret as (0,max); if max is None, interpret as expr*min + ZeroOrMore(expr)
- \- - like + but with no backup and retry of alternatives
- * - repetition of expression
- == - matching expression to string; returns True if the string matches the given expression
- <<= - inserts the expression following the operator as the body of the Forward expression before the operator

1.2.5 1.2.5 Positional subclasses

- `StringStart` - matches beginning of the text
- `StringEnd` - matches the end of the text
- `LineStart` - matches beginning of a line (lines delimited by `\n` characters)
- `LineEnd` - matches the end of a line
- `WordStart` - matches a leading word boundary
- `WordEnd` - matches a trailing word boundary

1.2.6 1.2.6 Converter subclasses

- `Combine` - joins all matched tokens into a single string, using specified joinString (default `joinString=""`); expects all matching tokens to be adjacent, with no intervening whitespace (can be overridden by specifying `adjacent=False` in constructor)
- `Suppress` - clears matched tokens; useful to keep returned results from being cluttered with required but uninteresting tokens (such as list delimiters)

1.2.7 1.2.7 Special subclasses

- `Group` - causes the matched tokens to be enclosed in a list; useful in repeated elements like `ZeroOrMore` and `OneOrMore` to break up matched tokens into groups for each repeated pattern
- `Dict` - like `Group`, but also constructs a dictionary, using the [0]'th elements of all enclosed token lists as the keys, and each token list as the value
- `SkipTo` - catch-all matching expression that accepts all characters up until the given pattern is found to match; useful for specifying incomplete grammars
- `Forward` - placeholder token used to define recursive token patterns; when defining the actual expression later in the program, insert it into the `Forward` object using the << operator (see `fourFn.py` for an example).

1.2.8 1.2.8 Other classes

- `ParseResults` - class used to contain and manage the lists of tokens created from parsing the input using the user-defined parse expression. ParseResults can be accessed in a number of ways:
  - as a list
    - total list of elements can be found using len()

    - individual elements can be found using [0], [1], [-1], etc.
    - elements can be deleted using `del`
    - the -1th element can be extracted and removed in a single operation using `pop()`, or any element can be extracted and removed using `pop(n)`
  - as a dictionary
    - if `setResultsName()` is used to name elements within the overall parse expression, then these fields can be referenced as dictionary elements or as attributes
    - the Dict class generates dictionary entries using the data of the input text - in addition to ParseResults listed as `[ [ a1, b1, c1, ...], [ a2, b2, c2, ...] ]` it also acts as a dictionary with entries defined as `{ a1 : [ b1, c1, ... ] }, { a2 : [ b2, c2, ... ] }`; this is especially useful when processing tabular data where the first column contains a key value for that line of data
    - list elements that are deleted using `del` will still be accessible by their dictionary keys
    - supports `get()`, `items()` and `keys()` methods, similar to a dictionary
    - a keyed item can be extracted and removed using `pop(key)`. Here key must be non-numeric (such as a string), in order to use dict extraction instead of list extraction.
    - new named elements can be added (in a parse action, for instance), using the same syntax as adding an item to a dict (`parseResults["X"] = "new item"`); named elements can be removed using `del parseResults["X"]`
  - as a nested list
    - results returned from the Group class are encapsulated within their own list structure, so that the tokens can be handled as a hierarchical tree

  ParseResults can also be converted to an ordinary list of strings by calling `asList()`. Note that this will strip the results of any field names that have been defined for any embedded parse elements. (The `pprint` module is especially good at printing out the nested contents given by `asList()`.)

  Finally, ParseResults can be viewed by calling `dump()`. ``dump()` will first show the ``asList()` output, followed by an indented structure listing parsed tokens that have been assigned results names.

1.2.9 1.2.9 Exception classes and Troubleshooting

- `ParseException` - exception returned when a grammar parse fails; ParseExceptions have attributes loc, msg, line, lineno, and column; to view the text line and location where the reported ParseException occurs, use:

```
except ParseException, err:
    print err.line
    print " " * (err.column - 1) + "^"
    print err
```

- `RecursiveGrammarException` - exception returned by `validate()` if the grammar contains a recursive infinite loop, such as:

```
badGrammar = Forward()
goodToken = Literal("A")
badGrammar <<= Optional(goodToken) + badGrammar
```

- `ParseFatalException` - exception that parse actions can raise to stop parsing immediately. Should be used when a semantic error is found in the input text, such as a mismatched XML tag.

- `ParseSyntaxException` - subclass of `ParseFatalException` raised when a syntax error is found, based on the use of the '-' operator when defining a sequence of expressions in an `And` expression.

You can also get some insights into the parsing logic using diagnostic parse actions, and setDebug(), or test the matching of expression fragments by testing them using scanString().

1.3 1.3 Miscellaneous attributes and methods

1.3.1 1.3.1 Helper methods

- `delimitedList(expr, delim=',')` - convenience function for matching one or more occurrences of expr, separated by delim. By default, the delimiters are suppressed, so the returned results contain only the separate list elements. Can optionally specify `combine=True`, indicating that the expressions and delimiters should be returned as one combined value (useful for scoped variables, such as `"a.b.c"`, or `"a::b::c"`, or paths such as `"a/b/c"`).
- `countedArray(expr)` - convenience function for a pattern where an list of instances of the given expression are preceded by an integer giving the count of elements in the list. Returns an expression that parses the leading integer, reads exactly that many expressions, and returns the array of expressions in the parse results - the leading integer is suppressed from the results (although it is easily reconstructed by using len on the returned array).
- `oneOf(string, caseless=False)` - convenience function for quickly declaring an alternative set of `Literal` tokens, by splitting the given string on whitespace boundaries. The tokens are sorted so that longer matches are attempted first; this ensures that a short token does not mask a longer one that starts with the same characters. If `caseless=True`, will create an alternative set of CaselessLiteral tokens.
- `dictOf(key, value)` - convenience function for quickly declaring a dictionary pattern of `Dict(ZeroOrMore(Group(key + value)))`.
- `makeHTMLTags(tagName)` and `makeXMLTags(tagName)` - convenience functions to create definitions of opening and closing tag expressions. Returns a pair of expressions, for the corresponding <tag> and </tag> strings. Includes support for attributes in the opening tag, such as <tag attr1="abc"> - attributes are returned as keyed tokens in the returned ParseResults. `makeHTMLTags` is less restrictive than `makeXMLTags`, especially with respect to case sensitivity.
- `infixNotation(baseOperand, operatorList)` - (formerly named `operatorPrecedence`) convenience function to define a grammar for parsing infix notation expressions with a hierarchical precedence of operators. To use the `infixNotation` helper:

  1. Define the base "atom" operand term of the grammar. For this simple grammar, the smallest operand is either and integer or a variable. This will be the first argument to the `infixNotation` method.
  2. Define a list of tuples for each level of operator precendence. Each tuple is of the form `(opExpr, numTerms, rightLeftAssoc, parseAction)`, where:
     - `opExpr` - the pyparsing expression for the operator; may also be a string, which will be converted to a Literal; if None, indicates an empty operator, such as the implied multiplication operation between 'm' and 'x' in "y = mx + b".
     - `numTerms` - the number of terms for this operator (must be 1, 2, or 3)
     - `rightLeftAssoc` is the indicator whether the operator is right or left associative, using the pyparsing-defined constants `opAssoc.RIGHT` and `opAssoc.LEFT`.
     - `parseAction` is the parse action to be associated with expressions matching this operator expression (the `parseAction` tuple member may be omitted)

3. Call `infixNotation` passing the operand expression and the operator precedence list, and save the returned value as the generated pyparsing expression. You can then use this expression to parse input strings, or incorporate it into a larger, more complex grammar.

- `matchPreviousLiteral` and `matchPreviousExpr` - function to define and expression that matches the same content as was parsed in a previous parse expression. For instance:

```
first = Word(nums)
matchExpr = first + ":" + matchPreviousLiteral(first)
```

will match "1:1", but not "1:2". Since this matches at the literal level, this will also match the leading "1:1" in "1:10".

In contrast:

```
first = Word(nums)
matchExpr = first + ":" + matchPreviousExpr(first)
```

will *not* match the leading "1:1" in "1:10"; the expressions are evaluated first, and then compared, so "1" is compared with "10".

- `nestedExpr(opener, closer, content=None, ignoreExpr=quotedString)` - method for defining nested lists enclosed in opening and closing delimiters.
  - `opener` - opening character for a nested list (default="("); can also be a pyparsing expression
  - `closer` - closing character for a nested list (default=")"); can also be a pyparsing expression
  - `content` - expression for items within the nested lists (default=None)
  - `ignoreExpr` - expression for ignoring opening and closing delimiters (default=quotedString)

  If an expression is not provided for the content argument, the nested expression will capture all whitespace-delimited content between delimiters

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv as a list of separate values.

  Use the ignoreExpr argument to define expressions that may contain opening or closing characters that should not be treated as opening or closing characters for nesting, such as quotedString or a comment expression. Specify multiple expressions using an Or or MatchFirst. The default is quotedString, but if no expressions are to be ignored, then pass None for this argument.

- `indentedBlock(statementExpr, indentationStackVar, indent=True)` - function to define an indented block of statements, similar to indentation-based blocking in Python source code:
  - `statementExpr` - the expression defining a statement that will be found in the indented block; a valid `indentedBlock` must contain at least 1 matching `statementExpr`
  - `indentationStackVar` - a Python list variable; this variable should be common to all `indentedBlock` expressions defined within the same grammar, and should be reinitialized to [1] each time the grammar is to be used
  - `indent` - a boolean flag indicating whether the expressions within the block must be indented from the current parse location; if using `indentedBlock` to define the left-most statements (all starting in column 1), set `indent` to False
- `originalTextFor(expr)` - helper function to preserve the originally parsed text, regardless of any token processing or conversion done by the contained expression. For instance, the following expression:

```
fullName = Word(alphas) + Word(alphas)
```

will return the parse of "John Smith" as ['John', 'Smith']. In some applications, the actual name as it was given in the input string is what is desired. To do this, use `originalTextFor`:

```
fullName = originalTextFor(Word(alphas) + Word(alphas))
```

- `ungroup(expr)` - function to "ungroup" returned tokens; useful to undo the default behavior of And to always group the returned tokens, even if there is only one in the list. (New in 1.5.6)
- `lineno(loc, string)` - function to give the line number of the location within the string; the first line is line 1, newlines start new rows
- `col(loc, string)` - function to give the column number of the location within the string; the first column is column 1, newlines reset the column number to 1
- `line(loc, string)` - function to retrieve the line of text representing `lineno(loc, string)`; useful when printing out diagnostic messages for exceptions
- `srange(rangeSpec)` - function to define a string of characters, given a string of the form used by regexp string ranges, such as `"[0-9]"` for all numeric digits, `"[A-Z_]"` for uppercase characters plus underscore, and so on (note that rangeSpec does not include support for generic regular expressions, just string range specs)
- `getTokensEndLoc()` - function to call from within a parse action to get the ending location for the matched tokens
- `traceParseAction(fn)` - decorator function to debug parse actions. Lists each call, called arguments, and return value or exception

1.3.2 1.3.2 Helper parse actions

- `removeQuotes` - removes the first and last characters of a quoted string; useful to remove the delimiting quotes from quoted strings
- `replaceWith(replString)` - returns a parse action that simply returns the replString; useful when using transformString, or converting HTML entities, as in:

```
nbsp = Literal(" ").setParseAction(replaceWith("<BLANK>"))
```

- `keepOriginalText`- (deprecated, use *originalTextFor* instead) restores any internal whitespace or suppressed text within the tokens for a matched parse expression. This is especially useful when defining expressions for scanString or transformString applications.
- `withAttribute(*args, **kwargs)` - helper to create a validating parse action to be used with start tags created with `makeXMLTags` or `makeHTMLTags`. Use `withAttribute` to qualify a starting tag with a required attribute value, to avoid false matches on common tags such as `<TD>` or `<DIV>`.

  `withAttribute` can be called with:

  - keyword arguments, as in `(class="Customer", align="right")`, or
  - a list of name-value tuples, as in `(("ns1:class", "Customer"), ("ns2:align", "right"))`

  An attribute can be specified to have the special value `withAttribute.ANY_VALUE`, which will match any value - use this to ensure that an attribute is present but any attribute value is acceptable.
- `downcaseTokens` - converts all matched tokens to lowercase
- `upcaseTokens` - converts all matched tokens to uppercase

- `matchOnlyAtCol(columnNumber)` - a parse action that verifies that an expression was matched at a particular column, raising a ParseException if matching at a different column number; useful when parsing tabular data

1.3.3 1.3.3 Common string and token constants

- `alphas` - same as `string.letters`
- `nums` - same as `string.digits`
- `alphanums` - a string containing `alphas + nums`
- `alphas8bit` - a string containing alphabetic 8-bit characters:

```
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ
```

- `printables` - same as `string.printable`, minus the space (`' '`) character
- `empty` - a global `Empty()`; will always match
- `sglQuotedString` - a string of characters enclosed in 's; may include whitespace, but not newlines
- `dblQuotedString` - a string of characters enclosed in "s; may include whitespace, but not newlines
- `quotedString` - `sglQuotedString | dblQuotedString`
- `cStyleComment` - a comment block delimited by `'/*'` and `'*/'` sequences; can span multiple lines, but does not support nesting of comments
- `htmlComment` - a comment block delimited by `'<!--'` and `'-->'` sequences; can span multiple lines, but does not support nesting of comments
- `commaSeparatedList` - similar to `delimitedList`, except that the list expressions can be any text value, or a quoted string; quoted strings can safely include commas without incorrectly breaking the string into two tokens
- `restOfLine` - all remaining printable characters up to but not including the next newline

CHAPTER 2

pyparsing

2.1 pyparsing module

2.1.1 pyparsing module - Classes and methods to define and execute parsing grammars

The pyparsing module is an alternative approach to creating and executing simple grammars, vs. the traditional lex/yacc approach, or the use of regular expressions. With pyparsing, you don't need to learn a new syntax for defining grammars or matching expressions - the parsing module provides a library of classes that you use to construct the grammar directly in Python.

Here is a program to parse "Hello, World!" (or any greeting of the form `"<salutation>, <addressee>!"`), built up using *Word*, *Literal*, and *And* elements (the `'+'` operators create *And* expressions, and the strings are auto-converted to *Literal* expressions):

```
from pyparsing import Word, alphas

# define grammar of a greeting
greet = Word(alphas) + "," + Word(alphas) + "!"

hello = "Hello, World!"
print (hello, "->", greet.parseString(hello))
```

The program outputs the following:

```
Hello, World! -> ['Hello', ',', 'World', '!']
```

The Python representation of the grammar is quite readable, owing to the self-explanatory class names, and the use of '+', '|' and '^' operators.

The *ParseResults* object returned from *ParserElement.parseString* can be accessed as a nested list, a dictionary, or an object with named attributes.

The pyparsing module handles some of the problems that are typically vexing when writing text parsers:

- extra or missing whitespace (the above program will also handle "Hello,World!", "Hello , World !", etc.)
- quoted strings
- embedded comments

Getting Started -

Visit the classes *ParserElement* and *ParseResults* to see the base classes that most other pyparsing classes inherit from. Use the docstrings for examples of how to:

- construct literal match expressions from *Literal* and *CaselessLiteral* classes
- construct character word-group expressions using the *Word* class
- see how to create repetitive expressions using *ZeroOrMore* and *OneOrMore* classes
- use *'+'*, *'|'*, *'^'*, and *'&'* operators to combine simple expressions into more complex ones
- associate names with your parsed results using *ParserElement.setResultsName*
- access the parsed data, which is returned as a *ParseResults* object
- find some helpful expression short-cuts like *delimitedList* and *oneOf*
- find more useful common expressions in the *pyparsing_common* namespace class

class pyparsing.**And**(*exprs*, *savelist=True*)
Bases: *pyparsing.ParseExpression*

Requires all given *ParseExpression* s to be found in the given order. Expressions may be separated by whitespace. May be constructed using the `'+'` operator. May also be constructed using the `'-'` operator, which will suppress backtracking.

Example:

```
integer = Word(nums)
name_expr = OneOrMore(Word(alphas))

expr = And([integer("id"), name_expr("name"), integer("age")])
# more easily written as:
expr = integer("id") + name_expr("name") + integer("age")
```

checkRecursion(*parseElementList*)

parseImpl(*instring*, *loc*, *doActions=True*)

streamline()

class pyparsing.**CaselessKeyword**(*matchString*, *identChars=None*)
Bases: *pyparsing.Keyword*

Caseless version of *Keyword*.

Example:

```
OneOrMore(CaselessKeyword("CMD")).parseString("cmd CMD Cmd10") # -> ['CMD', 'CMD']
```

(Contrast with example for *CaselessLiteral*.)

class pyparsing.**CaselessLiteral**(*matchString*)
Bases: *pyparsing.Literal*

Token to match a specified string, ignoring case of letters. Note: the matched results will always be in the case of the given match string, NOT the case of the input text.

Example:

```
OneOrMore(CaselessLiteral("CMD")).parseString("cmd CMD Cmd10") # -> ['CMD', 'CMD',
↪ 'CMD']
```

(Contrast with example for *CaselessKeyword*.)

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**CharsNotIn**(*notChars*, *min=1*, *max=0*, *exact=0*)
Bases: *pyparsing.Token*

Token for matching words composed of characters *not* in a given set (will include whitespace in matched characters if not listed in the provided exclusion set - see example). Defined with string containing all disallowed characters, and an optional minimum, maximum, and/or exact length. The default value for `min` is 1 (a minimum value < 1 is not valid); the default values for `max` and `exact` are 0, meaning no maximum or exact length restriction.

Example:

```
# define a comma-separated-value as anything that is not a ','
csv_value = CharsNotIn(',')
print(delimitedList(csv_value).parseString("dkls,lsdkjf,s12 34,@!#,213"))
```

prints:

```
['dkls', 'lsdkjf', 's12 34', '@!#', '213']
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Combine**(*expr*, *joinString=''*, *adjacent=True*)
Bases: *pyparsing.TokenConverter*

Converter to concatenate all matching tokens to a single string. By default, the matching patterns must also be contiguous in the input string; this can be disabled by specifying `'adjacent=False'` in the constructor.

Example:

```
real = Word(nums) + '.' + Word(nums)
print(real.parseString('3.1416')) # -> ['3', '.', '1416']
# will also erroneously match the following
print(real.parseString('3. 1416')) # -> ['3', '.', '1416']

real = Combine(Word(nums) + '.' + Word(nums))
print(real.parseString('3.1416')) # -> ['3.1416']
# no match when there are internal spaces
print(real.parseString('3. 1416')) # -> Exception: Expected W:(0123...)
```

ignore(*other*)
Define expression to be ignored (e.g., comments) while doing pattern matching; may be called repeatedly, to define multiple comment or other ignorable patterns.

Example:

```
patt = OneOrMore(Word(alphas))
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj']
```

(continues on next page)

(continued from previous page)

```
patt.ignore(cStyleComment)
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj', 'lskjd']
```

postParse(*instring*, *loc*, *tokenlist*)

class pyparsing.**Dict**(*expr*)

Bases: *pyparsing.TokenConverter*

Converter to return a repetitive expression as a list, but also as a dictionary. Each element can also be referenced using the first token in the expression as its key. Useful for tabular report scraping when the first column can be used as a item key.

Example:

```
data_word = Word(alphas)
label = data_word + FollowedBy(':')
attr_expr = Group(label + Suppress(':') + OneOrMore(data_word).setParseAction(' '.
↪join))

text = "shape: SQUARE posn: upper left color: light blue texture: burlap"
attr_expr = (label + Suppress(':') + OneOrMore(data_word, stopOn=label).
↪setParseAction(' '.join))

# print attributes as plain groups
print(OneOrMore(attr_expr).parseString(text).dump())

# instead of OneOrMore(expr), parse using Dict(OneOrMore(Group(expr))) - Dict␣
↪will auto-assign names
result = Dict(OneOrMore(Group(attr_expr))).parseString(text)
print(result.dump())

# access named fields as dict entries, or output as dict
print(result['shape'])
print(result.asDict())
```

prints:

```
['shape', 'SQUARE', 'posn', 'upper left', 'color', 'light blue', 'texture',
↪'burlap']
[['shape', 'SQUARE'], ['posn', 'upper left'], ['color', 'light blue'], ['texture',
↪ 'burlap']]
- color: light blue
- posn: upper left
- shape: SQUARE
- texture: burlap
SQUARE
{'color': 'light blue', 'posn': 'upper left', 'texture': 'burlap', 'shape':
↪'SQUARE'}
```

See more examples at *ParseResults* of accessing fields by results name.

postParse(*instring*, *loc*, *tokenlist*)

class pyparsing.**Each**(*exprs*, *savelist=True*)

Bases: *pyparsing.ParseExpression*

Requires all given *ParseExpression* s to be found, but in any order. Expressions may be separated by whitespace.

May be constructed using the '&' operator.

Example:

```
color = oneOf("RED ORANGE YELLOW GREEN BLUE PURPLE BLACK WHITE BROWN")
shape_type = oneOf("SQUARE CIRCLE TRIANGLE STAR HEXAGON OCTAGON")
integer = Word(nums)
shape_attr = "shape:" + shape_type("shape")
posn_attr = "posn:" + Group(integer("x") + ',' + integer("y"))("posn")
color_attr = "color:" + color("color")
size_attr = "size:" + integer("size")

# use Each (using operator '&') to accept attributes in any order
# (shape and posn are required, color and size are optional)
shape_spec = shape_attr & posn_attr & Optional(color_attr) & Optional(size_attr)

shape_spec.runTests('''
    shape: SQUARE color: BLACK posn: 100, 120
    shape: CIRCLE size: 50 color: BLUE posn: 50,80
    color:GREEN size:20 shape:TRIANGLE posn:20,40
    '''
    )
```

prints:

```
shape: SQUARE color: BLACK posn: 100, 120
['shape:', 'SQUARE', 'color:', 'BLACK', 'posn:', ['100', ',', '120']]
- color: BLACK
- posn: ['100', ',', '120']
  - x: 100
  - y: 120
- shape: SQUARE


shape: CIRCLE size: 50 color: BLUE posn: 50,80
['shape:', 'CIRCLE', 'size:', '50', 'color:', 'BLUE', 'posn:', ['50', ',', '80']]
- color: BLUE
- posn: ['50', ',', '80']
  - x: 50
  - y: 80
- shape: CIRCLE
- size: 50


color: GREEN size: 20 shape: TRIANGLE posn: 20,40
['color:', 'GREEN', 'size:', '20', 'shape:', 'TRIANGLE', 'posn:', ['20', ',', '40
↪']]
- color: GREEN
- posn: ['20', ',', '40']
  - x: 20
  - y: 40
- shape: TRIANGLE
- size: 20
```

checkRecursion(*parseElementList*)

parseImpl(*instring*, *loc*, *doActions=True*)

streamline()

class pyparsing.**Empty**
Bases: *pyparsing.Token*

An empty token, will always match.

class pyparsing.**FollowedBy**(*expr*)
Bases: *pyparsing.ParseElementEnhance*

Lookahead matching of the given parse expression. `FollowedBy` does *not* advance the parsing position within the input string, it only verifies that the specified parse expression matches at the current position. `FollowedBy` always returns a null token list. If any results names are defined in the lookahead expression, those *will* be returned for access by name.

Example:

```
# use FollowedBy to match a label only if it is followed by a ':'
data_word = Word(alphas)
label = data_word + FollowedBy(':')
attr_expr = Group(label + Suppress(':') + OneOrMore(data_word, stopOn=label).
↪setParseAction(' '.join))

OneOrMore(attr_expr).parseString("shape: SQUARE color: BLACK posn: upper left").
↪pprint()
```

prints:

```
[['shape', 'SQUARE'], ['color', 'BLACK'], ['posn', 'upper left']]
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Forward**(*other=None*)
Bases: *pyparsing.ParseElementEnhance*

Forward declaration of an expression to be defined later - used for recursive grammars, such as algebraic infix notation. When the expression is known, it is assigned to the `Forward` variable using the '<<' operator.

Note: take care when assigning to `Forward` not to overlook precedence of operators.

Specifically, '|' has a lower precedence than '<<', so that:

```
fwdExpr << a | b | c
```

will actually be evaluated as:

```
(fwdExpr << a) | b | c
```

thereby leaving b and c out as parseable alternatives. It is recommended that you explicitly group the values inserted into the `Forward`:

```
fwdExpr << (a | b | c)
```

Converting to use the '<<=' operator instead will avoid this problem.

See *ParseResults.pprint* for an example of a recursive parser created using `Forward`.

copy()
Make a copy of this *ParserElement*. Useful for defining different parse actions for the same parsing pattern, using copies of the original parse element.

Example:

```
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
integerK = integer.copy().addParseAction(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().addParseAction(lambda toks: toks[0] * 1024 * 1024)␣
↪+ Suppress("M")

print(OneOrMore(integerK | integerM | integer).parseString("5K 100 640K 256M
↪"))
```

prints:

```
[5120, 100, 655360, 268435456]
```

Equivalent form of `expr.copy()` is just `expr()`:

```
integerM = integer().addParseAction(lambda toks: toks[0] * 1024 * 1024) +␣
↪Suppress("M")
```

leaveWhitespace()
Disables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern. This is normally only used internally by the pyparsing module, but may be needed in some whitespace-sensitive grammars.

streamline()

validate(*validateTrace=None*)
Check defined expressions for valid structure, check for infinite recursive definitions.

class pyparsing.**GoToColumn**(*colno*)
Bases: pyparsing._PositionToken

Token to advance to a specific column of input text; useful for tabular report scraping.

parseImpl(*instring*, *loc*, *doActions=True*)

preParse(*instring*, *loc*)

class pyparsing.**Group**(*expr*)
Bases: *pyparsing.TokenConverter*

Converter to return the matched tokens as a list - useful for returning tokens of *ZeroOrMore* and *OneOrMore* expressions.

Example:

```
ident = Word(alphas)
num = Word(nums)
term = ident | num
func = ident + Optional(delimitedList(term))
print(func.parseString("fn a, b, 100"))  # -> ['fn', 'a', 'b', '100']

func = ident + Group(Optional(delimitedList(term)))
print(func.parseString("fn a, b, 100"))  # -> ['fn', ['a', 'b', '100']]
```

postParse(*instring*, *loc*, *tokenlist*)

class pyparsing.**Keyword**(*matchString*, *identChars=None*, *caseless=False*)
Bases: *pyparsing.Token*

Token to exactly match a specified string as a keyword, that is, it must be immediately followed by a non-keyword character. Compare with *Literal*:

- `Literal("if")` will match the leading `'if'` in `'ifAndOnlyIf'`.
- `Keyword("if")` will not; it will only match the leading `'if'` in `'if x=1'`, or `'if(y==2)'`

Accepts two optional constructor arguments in addition to the keyword string:

- `identChars` is a string of characters that would be valid identifier characters, defaulting to all alphanumerics + "_" and "$"
- `caseless` allows case-insensitive matching, default is `False`.

Example:

```
Keyword("start").parseString("start")  # -> ['start']
Keyword("start").parseString("starting")  # -> Exception
```

For case-insensitive matching, use *CaselessKeyword*.

DEFAULT_KEYWORD_CHARS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz012345678

copy()
Make a copy of this *ParserElement*. Useful for defining different parse actions for the same parsing pattern, using copies of the original parse element.

Example:

```
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
integerK = integer.copy().addParseAction(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().addParseAction(lambda toks: toks[0] * 1024 * 1024)␣
↪+ Suppress("M")

print(OneOrMore(integerK | integerM | integer).parseString("5K 100 640K 256M
↪"))
```

prints:

```
[5120, 100, 655360, 268435456]
```

Equivalent form of `expr.copy()` is just `expr()`:

```
integerM = integer().addParseAction(lambda toks: toks[0] * 1024 * 1024) +␣
↪Suppress("M")
```

parseImpl(*instring*, *loc*, *doActions=True*)

static setDefaultKeywordChars(*chars*)
Overrides the default Keyword chars

class pyparsing.**LineEnd**
Bases: pyparsing._PositionToken

Matches if current position is at the end of a line within the parse string

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**LineStart**
Bases: pyparsing._PositionToken

Matches if current position is at the beginning of a line within the parse string

Example:

```
test = '''\
AAA this line
AAA and this line
  AAA but not this one
B AAA and definitely not this one
'''

for t in (LineStart() + 'AAA' + restOfLine).searchString(test):
    print(t)
```

prints:

```
['AAA', ' this line']
['AAA', ' and this line']
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Literal**(*matchString*)
Bases: *pyparsing.Token*

Token to exactly match a specified string.

Example:

```
Literal('blah').parseString('blah')  # -> ['blah']
Literal('blah').parseString('blahfooblah')  # -> ['blah']
Literal('blah').parseString('bla')  # -> Exception: Expected "blah"
```

For case-insensitive matching, use *CaselessLiteral*.

For keyword matching (force word break before and after the matched string), use *Keyword* or *CaselessKeyword*.

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**PrecededBy**(*expr*, *retreat=None*)
Bases: *pyparsing.ParseElementEnhance*

Lookbehind matching of the given parse expression. `PrecededBy` does not advance the parsing position within the input string, it only verifies that the specified parse expression matches prior to the current position. `PrecededBy` always returns a null token list, but if a results name is defined on the given expression, it is returned.

Parameters:

- expr - expression that must match prior to the current parse location
- retreat - (default= `None`) - (int) maximum number of characters to lookbehind prior to the current parse location

If the lookbehind expression is a string, Literal, Keyword, or a Word or CharsNotIn with a specified exact or maximum length, then the retreat parameter is not required. Otherwise, retreat must be specified to give a maximum number of characters to look back from the current parse position for a lookbehind match.

Example:

```
# VB-style variable names with type prefixes
int_var = PrecededBy("#") + pyparsing_common.identifier
str_var = PrecededBy("$") + pyparsing_common.identifier
```

parseImpl(*instring*, *loc=0*, *doActions=True*)

class pyparsing.**MatchFirst**(*exprs*, *savelist=False*)
Bases: *pyparsing.ParseExpression*

Requires that at least one *ParseExpression* is found. If two expressions match, the first one listed is the one that will match. May be constructed using the '|' operator.

Example:

```
# construct MatchFirst using '|' operator

# watch the order of expressions to match
number = Word(nums) | Combine(Word(nums) + '.' + Word(nums))
print(number.searchString("123 3.1416 789")) #  Fail! -> [['123'], ['3'], ['1416
↪'], ['789']]

# put more selective expression first
number = Combine(Word(nums) + '.' + Word(nums)) | Word(nums)
print(number.searchString("123 3.1416 789")) #  Better -> [['123'], ['3.1416'], [
↪'789']]
```

checkRecursion(*parseElementList*)

parseImpl(*instring*, *loc*, *doActions=True*)

streamline()

class pyparsing.**NoMatch**
Bases: *pyparsing.Token*

A token that will never match.

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**NotAny**(*expr*)
Bases: *pyparsing.ParseElementEnhance*

Lookahead to disallow matching with the given parse expression. NotAny does *not* advance the parsing position within the input string, it only verifies that the specified parse expression does *not* match at the current position. Also, NotAny does *not* skip over leading whitespace. NotAny always returns a null token list. May be constructed using the '~' operator.

Example:

```
AND, OR, NOT = map(CaselessKeyword, "AND OR NOT".split())

# take care not to mistake keywords for identifiers
ident = ~(AND | OR | NOT) + Word(alphas)
boolean_term = Optional(NOT) + ident

# very crude boolean expression - to support parenthesis groups and
# operation hierarchy, use infixNotation
boolean_expr = boolean_term + ZeroOrMore((AND | OR) + boolean_term)

# integers that are followed by "." are actually floats
integer = Word(nums) + ~Char(".")
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**OneOrMore**(*expr*, *stopOn=None*)
Bases: pyparsing._MultipleMatch

Repetition of one or more of the given expression.

Parameters:

- expr - expression that must match one or more times
- **stopOn - (default= `None`) - expression for a terminating sentinel** (only required if the sentinel would ordinarily match the repetition expression)

Example:

```
data_word = Word(alphas)
label = data_word + FollowedBy(':')
attr_expr = Group(label + Suppress(':') + OneOrMore(data_word).setParseAction(' '.
↪join))

text = "shape: SQUARE posn: upper left color: BLACK"
OneOrMore(attr_expr).parseString(text).pprint()  # Fail! read 'color' as data␣
↪instead of next label -> [['shape', 'SQUARE color']]

# use stopOn attribute for OneOrMore to avoid reading label string as part of the␣
↪data
attr_expr = Group(label + Suppress(':') + OneOrMore(data_word, stopOn=label).
↪setParseAction(' '.join))
OneOrMore(attr_expr).parseString(text).pprint() # Better -> [['shape', 'SQUARE'],␣
↪['posn', 'upper left'], ['color', 'BLACK']]

# could also be written as
(attr_expr * (1,)).parseString(text).pprint()
```

class pyparsing.**OnlyOnce**(*methodCall*)

Bases: object

Wrapper for parse actions, to ensure they are only called once.

reset()

class pyparsing.**Optional**(*expr*, *default=<pyparsing._NullToken object>*)

Bases: *pyparsing.ParseElementEnhance*

Optional matching of the given expression.

Parameters:

- expr - expression that must match zero or more times
- default (optional) - value to be returned if the optional expression is not found.

Example:

```
# US postal code can be a 5-digit zip, plus optional 4-digit qualifier
zip = Combine(Word(nums, exact=5) + Optional('-' + Word(nums, exact=4)))
zip.runTests('''
    # traditional ZIP code
    12345

    # ZIP+4 form
    12101-0001

    # invalid ZIP
    98765-
    ''')
```

prints:

```
# traditional ZIP code
12345
['12345']

# ZIP+4 form
12101-0001
['12101-0001']

# invalid ZIP
98765-
     ^
FAIL: Expected end of text (at char 5), (line:1, col:6)
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Or**(*exprs*, *savelist=False*)
Bases: *pyparsing.ParseExpression*

Requires that at least one *ParseExpression* is found. If two expressions match, the expression that matches the longest string will be used. May be constructed using the `'^'` operator.

Example:

```
# construct Or using '^' operator

number = Word(nums) ^ Combine(Word(nums) + '.' + Word(nums))
print(number.searchString("123 3.1416 789"))
```

prints:

```
[['123'], ['3.1416'], ['789']]
```

checkRecursion(*parseElementList*)

parseImpl(*instring*, *loc*, *doActions=True*)

streamline()

exception pyparsing.**ParseBaseException**(*pstr*, *loc=0*, *msg=None*, *elem=None*)
Bases: exceptions.Exception

base exception class for all parsing runtime exceptions

markInputline(*markerString='>!<'*)
Extracts the exception line from the input string, and marks the location of the exception with a special symbol.

class pyparsing.**ParseElementEnhance**(*expr*, *savelist=False*)
Bases: *pyparsing.ParserElement*

Abstract subclass of *ParserElement*, for combining and post-processing parsed tokens.

checkRecursion(*parseElementList*)

ignore(*other*)
Define expression to be ignored (e.g., comments) while doing pattern matching; may be called repeatedly, to define multiple comment or other ignorable patterns.

Example:

```
patt = OneOrMore(Word(alphas))
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj']

patt.ignore(cStyleComment)
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj', 'lskjd']
```

leaveWhitespace()
Disables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern. This is normally only used internally by the pyparsing module, but may be needed in some whitespace-sensitive grammars.

parseImpl(*instring*, *loc*, *doActions=True*)

streamline()

validate(*validateTrace=None*)
Check defined expressions for valid structure, check for infinite recursive definitions.

exception pyparsing.**ParseException**(*pstr*, *loc=0*, *msg=None*, *elem=None*)
Bases: *pyparsing.ParseBaseException*

Exception thrown when parse expressions don't match class; supported attributes by name are: - lineno - returns the line number of the exception text - col - returns the column number of the exception text - line - returns the line containing the exception text

Example:

```
try:
    Word(nums).setName("integer").parseString("ABC")
except ParseException as pe:
    print(pe)
    print("column: {}".format(pe.col))
```

prints:

```
Expected integer (at char 0), (line:1, col:1)
 column: 1
```

static explain(*exc*, *depth=16*)
Method to take an exception and translate the Python internal traceback into a list of the pyparsing expressions that caused the exception to be raised.

Parameters:

- exc - exception raised during parsing (need not be a ParseException, in support of Python exceptions that might be raised in a parse action)
- depth (default=16) - number of levels back in the stack trace to list expression and function names; if None, the full stack trace names will be listed; if 0, only the failing input line, marker, and exception string will be shown

Returns a multi-line string listing the ParserElements and/or function names in the exception's stack trace.

Note: the diagnostic output will include string representations of the expressions that failed to parse. These representations will be more helpful if you use *setName* to give identifiable names to your expressions. Otherwise they will use the default string forms, which may be cryptic to read.

explain() is only supported under Python 3.

class pyparsing.**ParseExpression**(*exprs*, *savelist=False*)
Bases: *pyparsing.ParserElement*

Abstract subclass of ParserElement, for combining and post-processing parsed tokens.

append(*other*)

copy()
Make a copy of this *ParserElement*. Useful for defining different parse actions for the same parsing pattern, using copies of the original parse element.

Example:

```
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
integerK = integer.copy().addParseAction(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().addParseAction(lambda toks: toks[0] * 1024 * 1024)␣
↪+ Suppress("M")

print(OneOrMore(integerK | integerM | integer).parseString("5K 100 640K 256M
↪"))
```

prints:

```
[5120, 100, 655360, 268435456]
```

Equivalent form of `expr.copy()` is just `expr()`:

```
integerM = integer().addParseAction(lambda toks: toks[0] * 1024 * 1024) +␣
↪Suppress("M")
```

ignore(*other*)
Define expression to be ignored (e.g., comments) while doing pattern matching; may be called repeatedly, to define multiple comment or other ignorable patterns.

Example:

```
patt = OneOrMore(Word(alphas))
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj']

patt.ignore(cStyleComment)
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj', 'lskjd']
```

leaveWhitespace()
Extends `leaveWhitespace` defined in base class, and also invokes `leaveWhitespace` on all contained expressions.

streamline()

validate(*validateTrace=None*)
Check defined expressions for valid structure, check for infinite recursive definitions.

exception pyparsing.**ParseFatalException**(*pstr*, *loc=0*, *msg=None*, *elem=None*)
Bases: *pyparsing.ParseBaseException*

user-throwable exception thrown when inconsistent parse content is found; stops all parsing immediately

class pyparsing.**ParseResults**(*toklist=None*, *name=None*, *asList=True*, *modal=True*, *isinstance=<built-in function isinstance>*)
Bases: `object`

Structured parse results, to provide multiple means of access to the parsed data:

- as a list (`len(results)`)

- by list index (`results[0], results[1]`, etc.)
- by attribute (`results.<resultsName>` - see *ParserElement.setResultsName*)

Example:

```
integer = Word(nums)
date_str = (integer.setResultsName("year") + '/'
                + integer.setResultsName("month") + '/'
                + integer.setResultsName("day"))
# equivalent form:
# date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

# parseString returns a ParseResults object
result = date_str.parseString("1999/12/31")

def test(s, fn=repr):
    print("%s -> %s" % (s, fn(eval(s))))
test("list(result)")
test("result[0]")
test("result['month']")
test("result.day")
test("'month' in result")
test("'minutes' in result")
test("result.dump()", str)
```

prints:

```
list(result) -> ['1999', '/', '12', '/', '31']
result[0] -> '1999'
result['month'] -> '12'
result.day -> '31'
'month' in result -> True
'minutes' in result -> False
result.dump() -> ['1999', '/', '12', '/', '31']
- day: 31
- month: 12
- year: 1999
```

append(*item*)

Add single element to end of ParseResults list of elements.

Example:

```
print(OneOrMore(Word(nums)).parseString("0 123 321")) # -> ['0', '123', '321']

# use a parse action to compute the sum of the parsed integers, and add it to␣
↪the end
def append_sum(tokens):
    tokens.append(sum(map(int, tokens)))
print(OneOrMore(Word(nums)).addParseAction(append_sum).parseString("0 123 321
↪")) # -> ['0', '123', '321', 444]
```

asDict()

Returns the named parse results as a nested dictionary.

Example:

```
integer = Word(nums)
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

result = date_str.parseString('12/31/1999')
print(type(result), repr(result)) # -> <class 'pyparsing.ParseResults'> (['12
↪', '/', '31', '/', '1999'], {'day': [('1999', 4)], 'year': [('12', 0)],
↪'month': [('31', 2)]})

result_dict = result.asDict()
print(type(result_dict), repr(result_dict)) # -> <class 'dict'> {'day': '1999
↪', 'year': '12', 'month': '31'}

# even though a ParseResults supports dict-like access, sometime you just␣
↪need to have a dict
import json
print(json.dumps(result)) # -> Exception: TypeError: ... is not JSON␣
↪serializable
print(json.dumps(result.asDict())) # -> {"month": "31", "day": "1999", "year
↪": "12"}
```

asList()
Returns the parse results as a nested list of matching tokens, all converted to strings.

Example:

```
patt = OneOrMore(Word(alphas))
result = patt.parseString("sldkj lsdkj sldkj")
# even though the result prints in string-like form, it is actually a␣
↪pyparsing ParseResults
print(type(result), result) # -> <class 'pyparsing.ParseResults'> ['sldkj',
↪'lsdkj', 'sldkj']

# Use asList() to create an actual list
result_list = result.asList()
print(type(result_list), result_list) # -> <class 'list'> ['sldkj', 'lsdkj',
↪'sldkj']
```

asXML(*doctag=None*, *namedItemsOnly=False*, *indent=''*, *formatted=True*)
(Deprecated) Returns the parse results as XML. Tags are created for tokens and lists that have defined results names.

clear()
Clear all elements and results names.

copy()
Returns a new copy of a *ParseResults* object.

dump(*indent=''*, *full=True*, *include_list=True*, *_depth=0*)
Diagnostic method for listing out the contents of a *ParseResults*. Accepts an optional `indent` argument so that this string can be embedded in a nested display of other data.

Example:

```
integer = Word(nums)
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

result = date_str.parseString('12/31/1999')
print(result.dump())
```

prints:

```
['12', '/', '31', '/', '1999']
- day: 1999
- month: 31
- year: 12
```

extend(*itemseq*)

Add sequence of elements to end of ParseResults list of elements.

Example:

```
patt = OneOrMore(Word(alphas))

# use a parse action to append the reverse of the matched strings, to make a␣
↪palindrome
def make_palindrome(tokens):
    tokens.extend(reversed([t[::-1] for t in tokens]))
    return ''.join(tokens)
print(patt.addParseAction(make_palindrome).parseString("lskdj sdlkjf lksd"))
↪# -> 'lskdjsdlkjflksddsklfjkldsjdksl'
```

classmethod from_dict(*other*, *name=None*)

Helper classmethod to construct a ParseResults from a dict, preserving the name-value relations as results names. If an optional 'name' argument is given, a nested ParseResults will be returned

get(*key*, *defaultValue=None*)

Returns named result matching the given key, or if there is no such name, then returns the given `defaultValue` or `None` if no `defaultValue` is specified.

Similar to `dict.get()`.

Example:

```
integer = Word(nums)
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

result = date_str.parseString("1999/12/31")
print(result.get("year")) # -> '1999'
print(result.get("hour", "not specified")) # -> 'not specified'
print(result.get("hour")) # -> None
```

getName()

Returns the results name for this token expression. Useful when several different expressions might match at a particular location.

Example:

```
integer = Word(nums)
ssn_expr = Regex(r"\d\d\d-\d\d-\d\d\d\d")
house_number_expr = Suppress('#') + Word(nums, alphanums)
user_data = (Group(house_number_expr)("house_number")
            | Group(ssn_expr)("ssn")
            | Group(integer)("age"))
user_info = OneOrMore(user_data)

result = user_info.parseString("22 111-22-3333 #221B")
for item in result:
    print(item.getName(), ':', item[0])
```

prints:

```
age : 22
ssn : 111-22-3333
house_number : 221B
```

haskeys()
Since keys() returns an iterator, this method is helpful in bypassing code that looks for the existence of any defined results names.

insert(*index*, *insStr*)
Inserts new element at location index in the list of parsed tokens.

Similar to `list.insert()`.

Example:

```
print(OneOrMore(Word(nums)).parseString("0 123 321")) # -> ['0', '123', '321']

# use a parse action to insert the parse location in the front of the parsed␣
↪results
def insert_locn(locn, tokens):
    tokens.insert(0, locn)
print(OneOrMore(Word(nums)).addParseAction(insert_locn).parseString("0 123 321
↪")) # -> [0, '0', '123', '321']
```

items()
Returns an iterator of all named result key-value tuples.

iteritems()
Returns an iterator of all named result key-value tuples (Python 2.x only).

iterkeys()
Returns an iterator of all named result keys (Python 2.x only).

itervalues()
Returns an iterator of all named result values (Python 2.x only).

keys()
Returns an iterator of all named result keys.

pop(**args*, ***kwargs*)
Removes and returns item at specified index (default= `last`). Supports both `list` and `dict` semantics for `pop()`. If passed no argument or an integer argument, it will use `list` semantics and pop tokens from the list of parsed tokens. If passed a non-integer argument (most likely a string), it will use `dict` semantics and pop the corresponding value from any defined results names. A second default return value argument is supported, just as in `dict.pop()`.

Example:

```
def remove_first(tokens):
    tokens.pop(0)
print(OneOrMore(Word(nums)).parseString("0 123 321")) # -> ['0', '123', '321']
print(OneOrMore(Word(nums)).addParseAction(remove_first).parseString("0 123␣
↪321")) # -> ['123', '321']

label = Word(alphas)
patt = label("LABEL") + OneOrMore(Word(nums))
print(patt.parseString("AAB 123 321").dump())
```

(continues on next page)

(continued from previous page)

```
# Use pop() in a parse action to remove named result (note that corresponding␣
↪value is not
# removed from list form of results)
def remove_LABEL(tokens):
    tokens.pop("LABEL")
    return tokens
patt.addParseAction(remove_LABEL)
print(patt.parseString("AAB 123 321").dump())
```

prints:

```
['AAB', '123', '321']
- LABEL: AAB

['AAB', '123', '321']
```

pprint(**args*, ***kwargs*)

Pretty-printer for parsed results as a list, using the pprint module. Accepts additional positional or keyword args as defined for pprint.pprint .

Example:

```
ident = Word(alphas, alphanums)
num = Word(nums)
func = Forward()
term = ident | num | Group('(' + func + ')')
func <<= ident + Group(Optional(delimitedList(term)))
result = func.parseString("fna a,b,(fnb c,d,200),100")
result.pprint(width=40)
```

prints:

```
['fna',
 ['a',
  'b',
  ['(', 'fnb', ['c', 'd', '200'], ')'],
  '100']]
```

values()

Returns an iterator of all named result values.

exception pyparsing.**ParseSyntaxException**(*pstr*, *loc=0*, *msg=None*, *elem=None*)

Bases: *pyparsing.ParseFatalException*

just like *ParseFatalException*, but thrown internally when an ErrorStop ('-' operator) indicates that parsing is to stop immediately because an unbacktrackable syntax error has been found.

class pyparsing.**ParserElement**(*savelist=False*)

Bases: object

Abstract base level parser element class.

DEFAULT_WHITE_CHARS = ' \n\t\r'

addCondition(**fns*, ***kwargs*)

Add a boolean predicate function to expression's list of parse actions. See *setParseAction* for function call signatures. Unlike setParseAction, functions passed to addCondition need to return boolean success/fail of the condition.

Optional keyword arguments: - message = define a custom message to be used in the raised exception - fatal = if True, will raise ParseFatalException to stop parsing immediately; otherwise will raise ParseException

Example:

```
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
year_int = integer.copy()
year_int.addCondition(lambda toks: toks[0] >= 2000, message="Only support␣
↪years 2000 and later")
date_str = year_int + '/' + integer + '/' + integer

result = date_str.parseString("1999/12/31")  # -> Exception: Only support␣
↪years 2000 and later (at char 0), (line:1, col:1)
```

addParseAction (**fns*, ***kwargs*)
Add one or more parse actions to expression's list of parse actions. See *setParseAction*.

See examples in *copy*.

canParseNext (*instring*, *loc*)

checkRecursion (*parseElementList*)

copy ()
Make a copy of this *ParserElement*. Useful for defining different parse actions for the same parsing pattern, using copies of the original parse element.

Example:

```
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
integerK = integer.copy().addParseAction(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().addParseAction(lambda toks: toks[0] * 1024 * 1024)␣
↪+ Suppress("M")

print(OneOrMore(integerK | integerM | integer).parseString("5K 100 640K 256M
↪"))
```

prints:

```
[5120, 100, 655360, 268435456]
```

Equivalent form of `expr.copy()` is just `expr()`:

```
integerM = integer().addParseAction(lambda toks: toks[0] * 1024 * 1024) +␣
↪Suppress("M")
```

static enablePackrat (*cache_size_limit=128*)
Enables "packrat" parsing, which adds memoizing to the parsing logic. Repeated parse attempts at the same string location (which happens often in many complex grammars) can immediately return a cached value, instead of re-executing parsing/validating code. Memoizing is done of both valid results and parsing exceptions.

Parameters:

- cache_size_limit - (default= `128`) - if an integer value is provided will limit the size of the packrat cache; if None is passed, then the cache size will be unbounded; if 0 is passed, the cache will be effectively disabled.

This speedup may break existing programs that use parse actions that have side-effects. For this reason, packrat parsing is disabled when you first import pyparsing. To activate the packrat feature,

your program must call the class method *ParserElement.enablePackrat*. For best results, call `enablePackrat()` immediately after importing pyparsing.

Example:

```
import pyparsing
pyparsing.ParserElement.enablePackrat()
```

ignore(*other*)

Define expression to be ignored (e.g., comments) while doing pattern matching; may be called repeatedly, to define multiple comment or other ignorable patterns.

Example:

```
patt = OneOrMore(Word(alphas))
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj']

patt.ignore(cStyleComment)
patt.parseString('ablaj /* comment */ lskjd') # -> ['ablaj', 'lskjd']
```

static inlineLiteralsUsing(*cls*)

Set class to be used for inclusion of string literals into a parser.

Example:

```
# default literal class used is Literal
integer = Word(nums)
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

date_str.parseString("1999/12/31")  # -> ['1999', '/', '12', '/', '31']


# change to Suppress
ParserElement.inlineLiteralsUsing(Suppress)
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")

date_str.parseString("1999/12/31")  # -> ['1999', '12', '31']
```

leaveWhitespace()

Disables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern. This is normally only used internally by the pyparsing module, but may be needed in some whitespace-sensitive grammars.

matches(*testString*, *parseAll=True*)

Method for quick testing of a parser against a test string. Good for simple inline microtests of sub expressions while building up larger parser.

Parameters:

- testString - to test against this expression for a match
- parseAll - (default= `True`) - flag to pass to *parseString* when running tests

Example:

```
expr = Word(nums)
assert expr.matches("100")
```

packrat_cache = {}

packrat_cache_lock = <_RLock owner=None count=0>

packrat_cache_stats = [0, 0]

parseFile(*file_or_filename*, *parseAll=False*)
Execute the parse expression on the given file or filename. If a filename is specified (instead of a file object), the entire file is opened, read, and closed before parsing.

parseImpl(*instring*, *loc*, *doActions=True*)

parseString(*instring*, *parseAll=False*)
Execute the parse expression with the given string. This is the main interface to the client code, once the complete expression has been built.

Returns the parsed data as a *ParseResults* object, which may be accessed as a list, or as a dict or object with attributes if the given parser includes results names.

If you want the grammar to require that the entire input string be successfully parsed, then set `parseAll` to True (equivalent to ending the grammar with `StringEnd()`).

Note: `parseString` implicitly calls `expandtabs()` on the input string, in order to report proper column numbers in parse actions. If the input string contains tabs and the grammar uses parse actions that use the `loc` argument to index into the string being parsed, you can ensure you have a consistent view of the input string by:

- calling `parseWithTabs` on your grammar before calling `parseString` (see *parseWithTabs*)
- define your parse action using the full `(s, loc, toks)` signature, and reference the input string using the parse action's `s` argument
- explictly expand the tabs in your input string before calling `parseString`

Example:

```
Word('a').parseString('aaaaabaaa')  # -> ['aaaaa']
Word('a').parseString('aaaaabaaa', parseAll=True)  # -> Exception: Expected␣
↪end of text
```

parseWithTabs()
Overrides default behavior to expand `<TAB>``s to spaces before parsing the input string. Must be called before ``parseString` when the input grammar contains elements that match `<TAB>` characters.

postParse(*instring*, *loc*, *tokenlist*)

preParse(*instring*, *loc*)

static resetCache()

runTests(*tests*, *parseAll=True*, *comment='#'*, *fullDump=True*, *printResults=True*, *failureTests=False*, *postParse=None*, *file=None*)
Execute the parse expression on a series of test strings, showing each test, the parsed results or where the parse failed. Quick and easy way to run a parse expression against a list of sample strings.

Parameters:

- tests - a list of separate test strings, or a multiline string of test strings
- parseAll - (default= `True`) - flag to pass to *parseString* when running tests
- **comment - (default= `'#'`) - expression for indicating embedded comments in the test** string; pass None to disable comment filtering
- **fullDump - (default= `True`) - dump results as list followed by results names in nested outline;** if False, only dump nested list
- printResults - (default= `True`) prints test output to stdout

- failureTests - (default= `False`) indicates if these tests are expected to fail parsing
- **postParse - (default= `None`) optional callback for successful parse results; called as** *fn(test_string, parse_results)* and returns a string to be added to the test output
- **file - (default=‘‘None‘‘) optional file-like object to which test output will be written;** if None, will default to `sys.stdout`

Returns: a (success, results) tuple, where success indicates that all tests succeeded (or failed if `failureTests` is True), and the results contain a list of lines of each test's output

Example:

```
number_expr = pyparsing_common.number.copy()

result = number_expr.runTests('''
    # unsigned integer
    100
    # negative integer
    -100
    # float with scientific notation
    6.02e23
    # integer with scientific notation
    1e-12
    ''')
print("Success" if result[0] else "Failed!")

result = number_expr.runTests('''
    # stray character
    100Z
    # missing leading digit before '.'
    -.100
    # too many '.'
    3.14.159
    ''', failureTests=True)
print("Success" if result[0] else "Failed!")
```

prints:

```
# unsigned integer
100
[100]

# negative integer
-100
[-100]

# float with scientific notation
6.02e23
[6.02e+23]

# integer with scientific notation
1e-12
[1e-12]

Success

# stray character
100Z
```

(continues on next page)

(continued from previous page)

```
   ^
FAIL: Expected end of text (at char 3), (line:1, col:4)

# missing leading digit before '.'
-.100
^
FAIL: Expected {real number with scientific notation | real number | signed␣
↪integer} (at char 0), (line:1, col:1)

# too many '.'
3.14.159
    ^
FAIL: Expected end of text (at char 4), (line:1, col:5)

Success
```

Each test string must be on a single line. If you want to test a string that spans multiple lines, create a test like this:

```
expr.runTest(r"this is a test\n of strings that spans \n 3 lines")
```

(Note that this is a raw string literal, you must include the leading 'r'.)

scanString(*instring*, *maxMatches=9223372036854775807*, *overlap=False*)

Scan the input string for expression matches. Each match will return the matching tokens, start location, and end location. May be called with optional `maxMatches` argument, to clip scanning after 'n' matches are found. If `overlap` is specified, then overlapping matches will be reported.

Note that the start and end locations are reported relative to the string being parsed. See *parseString* for more information on parsing strings with embedded tabs.

Example:

```
source = "sldjf123lsdjjkf345sldkjf879lkjsfd987"
print(source)
for tokens, start, end in Word(alphas).scanString(source):
    print(' '*start + '^'*(end-start))
    print(' '*start + tokens[0])
```

prints:

```
sldjf123lsdjjkf345sldkjf879lkjsfd987
^^^^^
sldjf
        ^^^^^^^
        lsdjjkf
                 ^^^^^^
                 sldkjf
                          ^^^^^^
                          lkjsfd
```

searchString(*instring*, *maxMatches=9223372036854775807*)

Another extension to *scanString*, simplifying the access to the tokens found to match the given parse expression. May be called with optional `maxMatches` argument, to clip searching after 'n' matches are found.

Example:

```
# a capitalized word starts with an uppercase letter, followed by zero or␣
↪more lowercase letters
cap_word = Word(alphas.upper(), alphas.lower())

print(cap_word.searchString("More than Iron, more than Lead, more than Gold I␣
↪need Electricity"))

# the sum() builtin can be used to merge results into a single ParseResults␣
↪object
print(sum(cap_word.searchString("More than Iron, more than Lead, more than␣
↪Gold I need Electricity")))
```

prints:

```
[['More'], ['Iron'], ['Lead'], ['Gold'], ['I'], ['Electricity']]
['More', 'Iron', 'Lead', 'Gold', 'I', 'Electricity']
```

setBreak(*breakFlag=True*)
Method to invoke the Python pdb debugger when this element is about to be parsed. Set `breakFlag` to True to enable, False to disable.

setDebug(*flag=True*)
Enable display of debugging messages while doing pattern matching. Set `flag` to True to enable, False to disable.

Example:

```
wd = Word(alphas).setName("alphaword")
integer = Word(nums).setName("numword")
term = wd | integer

# turn on debugging for wd
wd.setDebug()

OneOrMore(term).parseString("abc 123 xyz 890")
```

prints:

```
Match alphaword at loc 0(1,1)
Matched alphaword -> ['abc']
Match alphaword at loc 3(1,4)
Exception raised:Expected alphaword (at char 4), (line:1, col:5)
Match alphaword at loc 7(1,8)
Matched alphaword -> ['xyz']
Match alphaword at loc 11(1,12)
Exception raised:Expected alphaword (at char 12), (line:1, col:13)
Match alphaword at loc 15(1,16)
Exception raised:Expected alphaword (at char 15), (line:1, col:16)
```

The output shown is that produced by the default debug actions - custom debug actions can be specified using *setDebugActions*. Prior to attempting to match the `wd` expression, the debugging message `"Match <exprname> at loc <n>(<line>,<col>)"` is shown. Then if the parse succeeds, a `"Matched"` message is shown, or an `"Exception raised"` message is shown. Also note the use of *setName* to assign a human-readable name to the expression, which makes debugging and exception messages easier to understand - for instance, the default name created for the *Word* expression without calling `setName` is `"W:(ABCD...)"`.

setDebugActions (*startAction*, *successAction*, *exceptionAction*)
Enable display of debugging messages while doing pattern matching.

static setDefaultWhitespaceChars (*chars*)
Overrides the default whitespace chars

Example:

```
# default whitespace chars are space, <TAB> and newline
OneOrMore(Word(alphas)).parseString("abc def\nghi jkl")  # -> ['abc', 'def',
↪'ghi', 'jkl']

# change to just treat newline as significant
ParserElement.setDefaultWhitespaceChars(" \t")
OneOrMore(Word(alphas)).parseString("abc def\nghi jkl")  # -> ['abc', 'def']
```

setFailAction (*fn*)
Define action to perform if parsing fails at this expression. Fail acton fn is a callable function that takes the arguments `fn(s, loc, expr, err)` where: - s = string being parsed - loc = location where expression match was attempted and failed - expr = the parse expression that failed - err = the exception thrown The function returns no value. It may throw *ParseFatalException* if it is desired to stop parsing immediately.

setName (*name*)
Define name for this expression, makes debugging and exception messages clearer.

Example:

```
Word(nums).parseString("ABC")  # -> Exception: Expected W:(0123...) (at char␣
↪0), (line:1, col:1)
Word(nums).setName("integer").parseString("ABC")  # -> Exception: Expected␣
↪integer (at char 0), (line:1, col:1)
```

setParseAction (**fns*, ***kwargs*)
Define one or more actions to perform when successfully matching parse element definition. Parse action fn is a callable method with 0-3 arguments, called as `fn(s, loc, toks)` , `fn(loc, toks)` , `fn(toks)` , or just `fn()` , where:

- s = the original string being parsed (see note below)
- loc = the location of the matching substring
- toks = a list of the matched tokens, packaged as a *ParseResults* object

If the functions in fns modify the tokens, they can return them as the return value from fn, and the modified list of tokens will replace the original. Otherwise, fn does not need to return any value.

If None is passed as the parse action, all previously added parse actions for this expression are cleared.

Optional keyword arguments: - callDuringTry = (default= `False`) indicate if parse action should be run during lookaheads and alternate testing

Note: the default parsing behavior is to expand tabs in the input string before starting the parsing process. See `parseString for more information on parsing strings containing ``<TAB>` s, and suggested methods to maintain a consistent view of the parsed string, the parse location, and line and column positions within the parsed string.

Example:

```
integer = Word(nums)
date_str = integer + '/' + integer + '/' + integer
```

(continues on next page)

(continued from previous page)

```
date_str.parseString("1999/12/31")  # -> ['1999', '/', '12', '/', '31']

# use parse action to convert to ints at parse time
integer = Word(nums).setParseAction(lambda toks: int(toks[0]))
date_str = integer + '/' + integer + '/' + integer

# note that integer fields are now ints, not strings
date_str.parseString("1999/12/31")  # -> [1999, '/', 12, '/', 31]
```

setResultsName(*name*, *listAllMatches=False*)

Define name for referencing matching tokens as a nested attribute of the returned parse results. NOTE: this returns a *copy* of the original *ParserElement* object; this is so that the client can define a basic element, such as an integer, and reference it in multiple places with different names.

You can also set results names using the abbreviated syntax, `expr("name")` in place of `expr.setResultsName("name")` - see `__call__`.

Example:

```
date_str = (integer.setResultsName("year") + '/'
            + integer.setResultsName("month") + '/'
            + integer.setResultsName("day"))

# equivalent form:
date_str = integer("year") + '/' + integer("month") + '/' + integer("day")
```

setWhitespaceChars(*chars*)

Overrides the default whitespace chars

split(*instring*, *maxsplit=9223372036854775807*, *includeSeparators=False*)

Generator method to split a string using the given expression as a separator. May be called with optional `maxsplit` argument, to limit the number of splits; and the optional `includeSeparators` argument (default= `False`), if the separating matching text should be included in the split results.

Example:

```
punc = oneOf(list(".,;:/-!?"))
print(list(punc.split("This, this?, this sentence, is badly punctuated!")))
```

prints:

```
['This', ' this', '', ' this sentence', ' is badly punctuated', '']
```

streamline()

suppress()

Suppresses the output of this *ParserElement*; useful to keep punctuation from cluttering up returned output.

transformString(*instring*)

Extension to *scanString*, to modify matching text with modified tokens that may be returned from a parse action. To use `transformString`, define a grammar and attach a parse action to it that modifies the returned token list. Invoking `transformString()` on a target string will then scan for matches, and replace the matched text patterns according to the logic in the parse action. `transformString()` returns the resulting transformed string.

Example:

```
wd = Word(alphas)
wd.setParseAction(lambda toks: toks[0].title())

print(wd.transformString("now is the winter of our discontent made glorious␣
↪summer by this sun of york."))
```

prints:

```
Now Is The Winter Of Our Discontent Made Glorious Summer By This Sun Of York.
```

tryParse(*instring*, *loc*)

validate(*validateTrace=None*)

Check defined expressions for valid structure, check for infinite recursive definitions.

verbose_stacktrace = False

class pyparsing.**QuotedString**(*quoteChar*, *escChar=None*, *escQuote=None*, *multiline=False*, *unquoteResults=True*, *endQuoteChar=None*, *convertWhitespaceEscapes=True*)

Bases: *pyparsing.Token*

Token for matching strings that are delimited by quoting characters.

Defined with the following parameters:

- quoteChar - string of one or more characters defining the quote delimiting string
- escChar - character to escape quotes, typically backslash (default= `None`)
- escQuote - special quote sequence to escape an embedded quote string (such as SQL's `""` to escape an embedded `"`) (default= `None`)
- multiline - boolean indicating whether quotes can span multiple lines (default= `False`)
- unquoteResults - boolean indicating whether the matched text should be unquoted (default= `True`)
- endQuoteChar - string of one or more characters defining the end of the quote delimited string (default= `None` => same as quoteChar)
- convertWhitespaceEscapes - convert escaped whitespace (`'\t'`, `'\n'`, etc.) to actual whitespace (default= `True`)

Example:

```
qs = QuotedString('"')
print(qs.searchString('lsjdf "This is the quote" sldjf'))
complex_qs = QuotedString('{{', endQuoteChar='}}')
print(complex_qs.searchString('lsjdf {{This is the "quote"}} sldjf'))
sql_qs = QuotedString('"', escQuote='""')
print(sql_qs.searchString('lsjdf "This is the quote with ""embedded"" quotes"␣
↪sldjf'))
```

prints:

```
[['This is the quote']]
[['This is the "quote"']]
[['This is the quote with "embedded" quotes']]
```

parseImpl(*instring*, *loc*, *doActions=True*)

exception pyparsing.**RecursiveGrammarException**(*parseElementList*)
Bases: exceptions.Exception

exception thrown by *ParserElement.validate* if the grammar could be improperly recursive

class pyparsing.**Regex**(*pattern*, *flags=0*, *asGroupList=False*, *asMatch=False*)
Bases: *pyparsing.Token*

Token for matching strings that match a given regular expression. Defined with string specifying the regular expression in a form recognized by the stdlib Python re module. If the given regex contains named groups (defined using `(?P<name>...)`), these will be preserved as named parse results.

Example:

```
realnum = Regex(r"[+-]?\d+\.\d*")
date = Regex(r'(?P<year>\d{4})-(?P<month>\d\d?)-(?P<day>\d\d?)')
# ref: https://stackoverflow.com/questions/267399/how-do-you-match-only-valid-
↪roman-numerals-with-a-regular-expression
roman = Regex(r"M{0,4}(CM|CD|D?{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})")
```

compiledREtype
alias of _sre.SRE_Pattern

parseImpl(*instring*, *loc*, *doActions=True*)

parseImplAsGroupList(*instring*, *loc*, *doActions=True*)

parseImplAsMatch(*instring*, *loc*, *doActions=True*)

sub(*repl*)
Return Regex with an attached parse action to transform the parsed result as if called using re.sub(expr, repl, string).

Example:

```
make_html = Regex(r"(\w+):(.*?):").sub(r"<\1>\2</\1>")
print(make_html.transformString("h1:main title:"))
# prints "<h1>main title</h1>"
```

class pyparsing.**SkipTo**(*other*, *include=False*, *ignore=None*, *failOn=None*)
Bases: *pyparsing.ParseElementEnhance*

Token for skipping over all undefined text until the matched expression is found.

Parameters:

- expr - target expression marking the end of the data to be skipped
- **include - (default= `False`) if True, the target expression is also parsed** (the skipped text and target expression are returned as a 2-element list).
- **ignore - (default= `None`) used to define grammars (typically quoted strings and** comments) that might contain false matches to the target expression
- **failOn - (default= `None`) define expressions that are not allowed to be** included in the skipped test; if found before the target expression is found, the SkipTo is not a match

Example:

```
report = '''
    Outstanding Issues Report - 1 Jan 2000

       # | Severity | Description                               |  Days Open
```

(continues on next page)

(continued from previous page)

```
    -----+----------+-----------------------------------------+-----------
     101 | Critical | Intermittent system crash               |           6
      94 | Cosmetic | Spelling error on Login ('log|n')       |          14
      79 | Minor    | System slow when running too many reports |          47
    '''
integer = Word(nums)
SEP = Suppress('|')
# use SkipTo to simply match everything up until the next SEP
# - ignore quoted strings, so that a '|' character inside a quoted string does␣
↪not match
# - parse action will call token.strip() for each matched token, i.e., the␣
↪description body
string_data = SkipTo(SEP, ignore=quotedString)
string_data.setParseAction(tokenMap(str.strip))
ticket_expr = (integer("issue_num") + SEP
               + string_data("sev") + SEP
               + string_data("desc") + SEP
               + integer("days_open"))

for tkt in ticket_expr.searchString(report):
    print tkt.dump()
```

prints:

```
['101', 'Critical', 'Intermittent system crash', '6']
- days_open: 6
- desc: Intermittent system crash
- issue_num: 101
- sev: Critical
['94', 'Cosmetic', "Spelling error on Login ('log|n')", '14']
- days_open: 14
- desc: Spelling error on Login ('log|n')
- issue_num: 94
- sev: Cosmetic
['79', 'Minor', 'System slow when running too many reports', '47']
- days_open: 47
- desc: System slow when running too many reports
- issue_num: 79
- sev: Minor
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**StringEnd**

Bases: pyparsing._PositionToken

Matches if current position is at the end of the parse string

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**StringStart**

Bases: pyparsing._PositionToken

Matches if current position is at the beginning of the parse string

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Suppress**(*expr*, *savelist=False*)

Bases: *pyparsing.TokenConverter*

Converter for ignoring the results of a parsed expression.

Example:

```
source = "a, b, c,d"
wd = Word(alphas)
wd_list1 = wd + ZeroOrMore(',' + wd)
print(wd_list1.parseString(source))

# often, delimiters that are useful during parsing are just in the
# way afterward - use Suppress to keep them out of the parsed output
wd_list2 = wd + ZeroOrMore(Suppress(',') + wd)
print(wd_list2.parseString(source))
```

prints:

```
['a', ',', 'b', ',', 'c', ',', 'd']
['a', 'b', 'c', 'd']
```

(See also *delimitedList*.)

postParse(*instring*, *loc*, *tokenlist*)

suppress()
Suppresses the output of this *ParserElement*; useful to keep punctuation from cluttering up returned output.

class pyparsing.**Token**
Bases: *pyparsing.ParserElement*

Abstract *ParserElement* subclass, for defining atomic matching patterns.

class pyparsing.**TokenConverter**(*expr*, *savelist=False*)
Bases: *pyparsing.ParseElementEnhance*

Abstract subclass of *ParseExpression*, for converting parsed results.

class pyparsing.**White**(*ws=' trn'*, *min=1*, *max=0*, *exact=0*)
Bases: *pyparsing.Token*

Special matching class for matching whitespace. Normally, whitespace is ignored by pyparsing grammars. This class is included when some whitespace structures are significant. Define with a string containing the whitespace characters to be matched; default is `" \t\r\n"`. Also takes optional `min`, `max`, and `exact` arguments, as defined for the *Word* class.

parseImpl(*instring*, *loc*, *doActions=True*)

whiteStrs = {'\t': '<TAB>', '\n': '<LF>', '\x0c': '<FF>', '\r': '<CR>', ' ': '<SP

class pyparsing.**Word**(*initChars*, *bodyChars=None*, *min=1*, *max=0*, *exact=0*, *asKeyword=False*, *excludeChars=None*)
Bases: *pyparsing.Token*

Token for matching words composed of allowed character sets. Defined with string containing all allowed initial characters, an optional string containing allowed body characters (if omitted, defaults to the initial character set), and an optional minimum, maximum, and/or exact length. The default value for `min` is 1 (a minimum value < 1 is not valid); the default values for `max` and `exact` are 0, meaning no maximum or exact length restriction. An optional `excludeChars` parameter can list characters that might be found in the input `bodyChars` string; useful to define a word of all printables except for one or two characters, for instance.

srange is useful for defining custom character set strings for defining `Word` expressions, using range notation from regular expression character sets.

A common mistake is to use *Word* to match a specific literal string, as in `Word("Address")`. Remember that *Word* uses the string argument to define *sets* of matchable characters. This expression would match "Add",

"AAA", "dAred", or any other word made up of the characters 'A', 'd', 'r', 'e', and 's'. To match an exact literal string, use *Literal* or *Keyword*.

pyparsing includes helper strings for building Words:

- alphas
- nums
- alphanums
- hexnums
- alphas8bit (alphabetic characters in ASCII range 128-255 - accented, tilded, umlauted, etc.)
- punc8bit (non-alphabetic characters in ASCII range 128-255 - currency, symbols, superscripts, diacriticals, etc.)
- printables (any non-whitespace character)

Example:

```
# a word composed of digits
integer = Word(nums) # equivalent to Word("0123456789") or Word(srange("0-9"))

# a word with a leading capital, and zero or more lowercase
capital_word = Word(alphas.upper(), alphas.lower())

# hostnames are alphanumeric, with leading alpha, and '-'
hostname = Word(alphas, alphanums + '-')

# roman numeral (not a strict parser, accepts invalid mix of characters)
roman = Word("IVXLCDM")

# any string of non-whitespace characters, except for ','
csv_value = Word(printables, excludeChars=",")
```

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**WordEnd**(*wordChars='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#*
-./:;<=>?@[\]^_'{|}~')

Bases: pyparsing._PositionToken

Matches if the current position is at the end of a Word, and is not followed by any character in a given set of wordChars (default= printables). To emulate the behavior of regular expressions, use WordEnd(alphanums). WordEnd will also match at the end of the string being parsed, or at the end of a line.

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**WordStart**(*wordChars='0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ*
-./:;<=>?@[\]^_'{|}~')

Bases: pyparsing._PositionToken

Matches if the current position is at the beginning of a Word, and is not preceded by any character in a given set of wordChars (default= printables). To emulate the behavior of regular expressions, use WordStart(alphanums). WordStart will also match at the beginning of the string being parsed, or at the beginning of a line.

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**ZeroOrMore**(*expr*, *stopOn=None*)

Bases: pyparsing._MultipleMatch

Optional repetition of zero or more of the given expression.

Parameters:

- expr - expression that must match zero or more times
- **stopOn - (default= None) - expression for a terminating sentinel** (only required if the sentinel would ordinarily match the repetition expression)

Example: similar to *OneOrMore*

parseImpl(*instring*, *loc*, *doActions=True*)

class pyparsing.**Char**(*charset*, *asKeyword=False*, *excludeChars=None*)
Bases: pyparsing._WordRegex

A short-cut class for defining `Word(characters, exact=1)`, when defining a match of any single character in a string of characters.

pyparsing.**cStyleComment = C style comment**
Comment of the form `/* ... */`

pyparsing.**col**(*loc*, *strg*)
Returns current column within a string, counting newlines as line separators. The first column is number 1.

Note: the default parsing behavior is to expand tabs in the input string before starting the parsing process. See *ParserElement.parseString* for more information on parsing strings containing `<TAB>` s, and suggested methods to maintain a consistent view of the parsed string, the parse location, and line and column positions within the parsed string.

pyparsing.**commaSeparatedList = commaSeparatedList**
(Deprecated) Predefined expression of 1 or more printable words or quoted strings, separated by commas.

This expression is deprecated in favor of *pyparsing_common.comma_separated_list*.

pyparsing.**countedArray**(*expr*, *intExpr=None*)
Helper to define a counted list of expressions.

This helper defines a pattern of the form:

```
integer expr expr expr...
```

where the leading integer tells how many expr expressions follow. The matched tokens returns the array of expr tokens as a list - the leading count token is suppressed.

If `intExpr` is specified, it should be a pyparsing expression that produces an integer value.

Example:

```
countedArray(Word(alphas)).parseString('2 ab cd ef')  # -> ['ab', 'cd']

# in this parser, the leading integer value is given in binary,
# '10' indicating that 2 values are in the array
binaryConstant = Word('01').setParseAction(lambda t: int(t[0], 2))
countedArray(Word(alphas), intExpr=binaryConstant).parseString('10 ab cd ef')  # -
↪> ['ab', 'cd']
```

pyparsing.**cppStyleComment = C++ style comment**
Comment of either form *cStyleComment* or *dblSlashComment*

pyparsing.**dblSlashComment = // comment**
Comment of the form `// ... (to end of line)`

pyparsing.**delimitedList**(*expr*, *delim=', '*, *combine=False*)

Helper to define a delimited list of expressions - the delimiter defaults to ','. By default, the list elements and delimiters can have intervening whitespace, and comments, but this can be overridden by passing `combine=True` in the constructor. If `combine` is set to `True`, the matching tokens are returned as a single token string, with the delimiters included; otherwise, the matching tokens are returned as a list of tokens, with the delimiters suppressed.

Example:

```
delimitedList(Word(alphas)).parseString("aa,bb,cc") # -> ['aa', 'bb', 'cc']
delimitedList(Word(hexnums), delim=':', combine=True).parseString("AA:BB:CC:DD:EE
↪") # -> ['AA:BB:CC:DD:EE']
```

pyparsing.**dictOf**(*key*, *value*)

Helper to easily and clearly define a dictionary by specifying the respective patterns for the key and value. Takes care of defining the *Dict*, *ZeroOrMore*, and *Group* tokens in the proper order. The key pattern can include delimiting markers or punctuation, as long as they are suppressed, thereby leaving the significant key text. The value pattern can include named results, so that the *Dict* results can include named token fields.

Example:

```
text = "shape: SQUARE posn: upper left color: light blue texture: burlap"
attr_expr = (label + Suppress(':') + OneOrMore(data_word, stopOn=label).
↪setParseAction(' '.join))
print(OneOrMore(attr_expr).parseString(text).dump())

attr_label = label
attr_value = Suppress(':') + OneOrMore(data_word, stopOn=label).setParseAction('
↪'.join)

# similar to Dict, but simpler call format
result = dictOf(attr_label, attr_value).parseString(text)
print(result.dump())
print(result['shape'])
print(result.shape)  # object attribute access works too
print(result.asDict())
```

prints:

```
[['shape', 'SQUARE'], ['posn', 'upper left'], ['color', 'light blue'], ['texture',
↪ 'burlap']]
- color: light blue
- posn: upper left
- shape: SQUARE
- texture: burlap
SQUARE
SQUARE
{'color': 'light blue', 'shape': 'SQUARE', 'posn': 'upper left', 'texture':
↪'burlap'}
```

pyparsing.**downcaseTokens**(*s*, *l*, *t*)

(Deprecated) Helper parse action to convert tokens to lower case. Deprecated in favor of *pyparsing_common.downcaseTokens*

pyparsing.**htmlComment = HTML comment**

Comment of the form `<!-- ... -->`

pyparsing.**javaStyleComment = C++ style comment**

Same as *cppStyleComment*

pyparsing.**line**(*loc*, *strg*)
Returns the line of text containing loc within a string, counting newlines as line separators.

pyparsing.**lineno**(*loc*, *strg*)
Returns current line number within a string, counting newlines as line separators. The first line is number 1.

Note - the default parsing behavior is to expand tabs in the input string before starting the parsing process. See *ParserElement.parseString* for more information on parsing strings containing <TAB> s, and suggested methods to maintain a consistent view of the parsed string, the parse location, and line and column positions within the parsed string.

pyparsing.**makeHTMLTags**(*tagStr*)
Helper to construct opening and closing tag expressions for HTML, given a tag name. Matches tags in either upper or lower case, attributes with namespaces and with quoted or unquoted values.

Example:

```
text = '<td>More info at the <a href="https://github.com/pyparsing/pyparsing/wiki
↪">pyparsing</a> wiki page</td>'
# makeHTMLTags returns pyparsing expressions for the opening and
# closing tags as a 2-tuple
a, a_end = makeHTMLTags("A")
link_expr = a + SkipTo(a_end)("link_text") + a_end

for link in link_expr.searchString(text):
    # attributes in the <A> tag (like "href" shown here) are
    # also accessible as named results
    print(link.link_text, '->', link.href)
```

prints:

```
pyparsing -> https://github.com/pyparsing/pyparsing/wiki
```

pyparsing.**makeXMLTags**(*tagStr*)
Helper to construct opening and closing tag expressions for XML, given a tag name. Matches tags only in the given upper/lower case.

Example: similar to *makeHTMLTags*

pyparsing.**matchOnlyAtCol**(*n*)
Helper method for defining parse actions that require matching at a specific column in the input text.

pyparsing.**matchPreviousExpr**(*expr*)
Helper to define an expression that is indirectly defined from the tokens matched in a previous expression, that is, it looks for a 'repeat' of a previous expression. For example:

```
first = Word(nums)
second = matchPreviousExpr(first)
matchExpr = first + ":" + second
```

will match "1:1", but not "1:2". Because this matches by expressions, will *not* match the leading "1:1" in "1:10"; the expressions are evaluated first, and then compared, so "1" is compared with "10". Do *not* use with packrat parsing enabled.

pyparsing.**matchPreviousLiteral**(*expr*)
Helper to define an expression that is indirectly defined from the tokens matched in a previous expression, that is, it looks for a 'repeat' of a previous expression. For example:

```
first = Word(nums)
second = matchPreviousLiteral(first)
matchExpr = first + ":" + second
```

will match `"1:1"`, but not `"1:2"`. Because this matches a previous literal, will also match the leading `"1:1"` in `"1:10"`. If this is not desired, use *matchPreviousExpr*. Do *not* use with packrat parsing enabled.

pyparsing.**nestedExpr**(*opener='('*, *closer=')'*, *content=None*, *ignoreExpr=quotedString using single or double quotes*)

Helper method for defining nested lists enclosed in opening and closing delimiters ("(" and ")" are the default).

Parameters:

- opener - opening character for a nested list (default= `"("`); can also be a pyparsing expression
- closer - closing character for a nested list (default= `")"`); can also be a pyparsing expression
- content - expression for items within the nested lists (default= `None`)
- ignoreExpr - expression for ignoring opening and closing delimiters (default= `quotedString`)

If an expression is not provided for the content argument, the nested expression will capture all whitespace-delimited content between delimiters as a list of separate values.

Use the `ignoreExpr` argument to define expressions that may contain opening or closing characters that should not be treated as opening or closing characters for nesting, such as quotedString or a comment expression. Specify multiple expressions using an *Or* or *MatchFirst*. The default is `quotedString`, but if no expressions are to be ignored, then pass `None` for this argument.

Example:

```
data_type = oneOf("void int short long char float double")
decl_data_type = Combine(data_type + Optional(Word('*')))
ident = Word(alphas+'_', alphanums+'_')
number = pyparsing_common.number
arg = Group(decl_data_type + ident)
LPAR, RPAR = map(Suppress, "()")

code_body = nestedExpr('{', '}', ignoreExpr=(quotedString | cStyleComment))

c_function = (decl_data_type("type")
              + ident("name")
              + LPAR + Optional(delimitedList(arg), [])("args") + RPAR
              + code_body("body"))
c_function.ignore(cStyleComment)

source_code = '''
    int is_odd(int x) {
        return (x%2);
    }

    int dec_to_hex(char hchar) {
        if (hchar >= '0' && hchar <= '9') {
            return (ord(hchar)-ord('0'));
        } else {
            return (10+ord(hchar)-ord('A'));
        }
    }
'''
```

(continues on next page)

(continued from previous page)

```
for func in c_function.searchString(source_code):
    print("%(name)s (%(type)s) args: %(args)s" % func)
```

prints:

```
is_odd (int) args: [['int', 'x']]
dec_to_hex (int) args: [['char', 'hchar']]
```

pyparsing.**nullDebugAction**(**args*)
'Do-nothing' debug action, to suppress debugging output during parsing.

pyparsing.**oneOf**(*strs*, *caseless=False*, *useRegex=True*, *asKeyword=False*)
Helper to quickly define a set of alternative Literals, and makes sure to do longest-first testing when there is a conflict, regardless of the input order, but returns a *MatchFirst* for best performance.

Parameters:

- strs - a string of space-delimited literals, or a collection of string literals
- caseless - (default= `False`) - treat all literals as caseless
- useRegex - (default= `True`) - as an optimization, will generate a Regex object; otherwise, will generate a *MatchFirst* object (if `caseless=True` or `asKeyword=True`, or if creating a *Regex* raises an exception)
- asKeyword - (default=''False'') - enforce Keyword-style matching on the generated expressions

Example:

```
comp_oper = oneOf("< = > <= >= !=")
var = Word(alphas)
number = Word(nums)
term = var | number
comparison_expr = term + comp_oper + term
print(comparison_expr.searchString("B = 12  AA=23 B<=AA AA>12"))
```

prints:

```
[['B', '=', '12'], ['AA', '=', '23'], ['B', '<=', 'AA'], ['AA', '>', '12']]
```

pyparsing.**operatorPrecedence**(*baseExpr*, *opList*, *lpar=Suppress:("(")*, *rpar=Suppress:(")")*)
(Deprecated) Former name of *infixNotation*, will be dropped in a future release.

pyparsing.**pythonStyleComment = Python style comment**
Comment of the form `# ... (to end of line)`

pyparsing.**removeQuotes**(*s*, *l*, *t*)
Helper parse action for removing quotation marks from parsed quoted strings.

Example:

```
# by default, quotation marks are included in parsed results
quotedString.parseString("'Now is the Winter of our Discontent'") # -> ["'Now is␣
↪the Winter of our Discontent'"]

# use removeQuotes to strip quotation marks from parsed results
quotedString.setParseAction(removeQuotes)
quotedString.parseString("'Now is the Winter of our Discontent'") # -> ["Now is␣
↪the Winter of our Discontent"]
```

pyparsing.**replaceHTMLEntity**(*t*)
Helper parser action to replace common HTML entities with their special characters

pyparsing.**replaceWith**(*replStr*)
Helper method for common parse actions that simply return a literal value. Especially useful when used with *transformString* ().

Example:

```
num = Word(nums).setParseAction(lambda toks: int(toks[0]))
na = oneOf("N/A NA").setParseAction(replaceWith(math.nan))
term = na | num

OneOrMore(term).parseString("324 234 N/A 234") # -> [324, 234, nan, 234]
```

pyparsing.**srange**(*s*)
Helper to easily define string ranges for use in Word construction. Borrows syntax from regexp '[]' string range definitions:

```
srange("[0-9]")   -> "0123456789"
srange("[a-z]")   -> "abcdefghijklmnopqrstuvwxyz"
srange("[a-z$_]") -> "abcdefghijklmnopqrstuvwxyz$_"
```

The input string must be enclosed in []'s, and the returned string is the expanded character set joined into a single string. The values enclosed in the []'s may be:

- a single character
- an escaped character with a leading backslash (such as `\-` or `\]`)
- an escaped hex character with a leading `'\x'` (`\x21`, which is a `'!'` character) (`\0x##` is also supported for backwards compatibility)
- an escaped octal character with a leading `'\0'` (`\041`, which is a `'!'` character)
- a range of any of the above, separated by a dash (`'a-z'`, etc.)
- any combination of the above (`'aeiouy'`, `'a-zA-Z0-9_$'`, etc.)

pyparsing.**traceParseAction**(*f*)
Decorator for debugging parse actions.

When the parse action is called, this decorator will print `">> entering method-name(line:<current_source_line>, <parse_location>, <matched_tokens>)"`. When the parse action completes, the decorator will print `"<<"` followed by the returned value, or any exception that the parse action raised.

Example:

```
wd = Word(alphas)

@traceParseAction
def remove_duplicate_chars(tokens):
    return ''.join(sorted(set(''.join(tokens))))

wds = OneOrMore(wd).setParseAction(remove_duplicate_chars)
print(wds.parseString("slkdjs sld sldd sdlf sdljf"))
```

prints:

```
>>entering remove_duplicate_chars(line: 'slkdjs sld sldd sdlf sdljf', 0, (['slkdjs
↪', 'sld', 'sldd', 'sdlf', 'sdljf'], {}))
<<leaving remove_duplicate_chars (ret: 'dfjkls')
['dfjkls']
```

pyparsing.**upcaseTokens**(*s*, *l*, *t*)

(Deprecated) Helper parse action to convert tokens to upper case. Deprecated in favor of *pyparsing_common.upcaseTokens*

pyparsing.**withAttribute**(**args*, ***attrDict*)

Helper to create a validating parse action to be used with start tags created with *makeXMLTags* or *makeHTMLTags*. Use `withAttribute` to qualify a starting tag with a required attribute value, to avoid false matches on common tags such as `<TD>` or `<DIV>`.

Call `withAttribute` with a series of attribute names and values. Specify the list of filter attributes names and values as:

- keyword arguments, as in `(align="right")`, or
- as an explicit dict with `**` operator, when an attribute name is also a Python reserved word, as in `**{"class":"Customer", "align":"right"}`
- a list of name-value tuples, as in `(("ns1:class", "Customer"), ("ns2:align", "right"))`

For attribute names with a namespace prefix, you must use the second form. Attribute names are matched insensitive to upper/lower case.

If just testing for `class` (with or without a namespace), use *withClass*.

To verify that the attribute exists, but without specifying a value, pass `withAttribute.ANY_VALUE` as the value.

Example:

```
html = '''
    <div>
    Some text
    <div type="grid">1 4 0 1 0</div>
    <div type="graph">1,3 2,3 1,1</div>
    <div>this has no type</div>
    </div>

'''
div,div_end = makeHTMLTags("div")

# only match div tag having a type attribute with value "grid"
div_grid = div().setParseAction(withAttribute(type="grid"))
grid_expr = div_grid + SkipTo(div | div_end)("body")
for grid_header in grid_expr.searchString(html):
    print(grid_header.body)

# construct a match with any div tag having a type attribute, regardless of the␣
↪value
div_any_type = div().setParseAction(withAttribute(type=withAttribute.ANY_VALUE))
div_expr = div_any_type + SkipTo(div | div_end)("body")
for div_header in div_expr.searchString(html):
    print(div_header.body)
```

prints:

```
1 4 0 1 0

1 4 0 1 0
1,3 2,3 1,1
```

pyparsing.**indentedBlock**(*blockStatementExpr*, *indentStack*, *indent=True*)

Helper method for defining space-delimited indentation blocks, such as those used to define block statements in Python source code.

Parameters:

- blockStatementExpr - expression defining syntax of statement that is repeated within the indented block
- indentStack - list created by caller to manage indentation stack (multiple statementWithIndentedBlock expressions within a single grammar should share a common indentStack)
- indent - boolean indicating whether block must be indented beyond the current level; set to False for block of left-most statements (default= `True`)

A valid block must contain at least one `blockStatement`.

Example:

```
data = '''
def A(z):
  A1
  B = 100
  G = A2
  A2
  A3
B
def BB(a,b,c):
  BB1
  def BBA():
    bba1
    bba2
    bba3
C
D
def spam(x,y):
     def eggs(z):
         pass
'''


indentStack = [1]
stmt = Forward()

identifier = Word(alphas, alphanums)
funcDecl = ("def" + identifier + Group("(" + Optional(delimitedList(identifier))␣
↪+ ")") + ":")
func_body = indentedBlock(stmt, indentStack)
funcDef = Group(funcDecl + func_body)

rvalue = Forward()
funcCall = Group(identifier + "(" + Optional(delimitedList(rvalue)) + ")")
rvalue << (funcCall | identifier | Word(nums))
assignment = Group(identifier + "=" + rvalue)
stmt << (funcDef | assignment | identifier)
```

(continues on next page)

(continued from previous page)

```
module_body = OneOrMore(stmt)

parseTree = module_body.parseString(data)
parseTree.pprint()
```

prints:

```
[['def',
  'A',
  ['(', 'z', ')'],
  ':',
  [['A1'], [['B', '=', '100']], [['G', '=', 'A2']], ['A2'], ['A3']]],
 'B',
 ['def',
  'BB',
  ['(', 'a', 'b', 'c', ')'],
  ':',
  [['BB1'], [['def', 'BBA', ['(', ')'], ':', [['bba1'], ['bba2'], ['bba3']]]]]],
 'C',
 'D',
 ['def',
  'spam',
  ['(', 'x', 'y', ')'],
  ':',
  [[['def', 'eggs', ['(', 'z', ')'], ':', [['pass']]]]]]]
```

pyparsing.**originalTextFor**(*expr*, *asString=True*)

Helper to return the original, untokenized text for a given expression. Useful to restore the parsed fields of an HTML start tag into the raw tag text itself, or to revert separate tokens with intervening whitespace back to the original matching input text. By default, returns astring containing the original parsed text.

If the optional `asString` argument is passed as `False`, then the return value is a *`ParseResults`* containing any results names that were originally matched, and a single token containing the original matched text from the input string. So if the expression passed to *`originalTextFor`* contains expressions with defined results names, you must set `asString` to `False` if you want to preserve those results name values.

Example:

```
src = "this is test <b> bold <i>text</i> </b> normal text "
for tag in ("b", "i"):
    opener, closer = makeHTMLTags(tag)
    patt = originalTextFor(opener + SkipTo(closer) + closer)
    print(patt.searchString(src)[0])
```

prints:

```
['<b> bold <i>text</i> </b>']
['<i>text</i>']
```

pyparsing.**ungroup**(*expr*)

Helper to undo pyparsing's default grouping of And expressions, even if all but one are non-empty.

pyparsing.**infixNotation**(*baseExpr*, *opList*, *lpar=Suppress:("(")*, *rpar=Suppress:(")")*)

Helper method for constructing grammars of expressions made up of operators working in a precedence hierarchy. Operators may be unary or binary, left- or right-associative. Parse actions can also be attached to operator expressions. The generated parser will also recognize the use of parentheses to override operator precedences

(see example below).

Note: if you define a deep operator list, you may see performance issues when using infixNotation. See *ParserElement.enablePackrat* for a mechanism to potentially improve your parser performance.

Parameters:

- baseExpr - expression representing the most basic element for the nested
- opList - list of tuples, one for each operator precedence level in the expression grammar; each tuple is of the form `(opExpr, numTerms, rightLeftAssoc, parseAction)`, where:
  - opExpr is the pyparsing expression for the operator; may also be a string, which will be converted to a Literal; if numTerms is 3, opExpr is a tuple of two expressions, for the two operators separating the 3 terms
  - numTerms is the number of terms for this operator (must be 1, 2, or 3)
  - rightLeftAssoc is the indicator whether the operator is right or left associative, using the pyparsing-defined constants `opAssoc.RIGHT` and `opAssoc.LEFT`.
  - parseAction is the parse action to be associated with expressions matching this operator expression (the parse action tuple member may be omitted); if the parse action is passed a tuple or list of functions, this is equivalent to calling `setParseAction(*fn)` (*ParserElement.setParseAction*)
- lpar - expression for matching left-parentheses (default= `Suppress('(')`)
- rpar - expression for matching right-parentheses (default= `Suppress(')')`)

Example:

```
# simple example of four-function arithmetic with ints and
# variable names
integer = pyparsing_common.signed_integer
varname = pyparsing_common.identifier

arith_expr = infixNotation(integer | varname,
    [
    ('-', 1, opAssoc.RIGHT),
    (oneOf('* /'), 2, opAssoc.LEFT),
    (oneOf('+ -'), 2, opAssoc.LEFT),
    ])

arith_expr.runTests('''
    5+3*6
    (5+3)*6
    -2--11
    ''', fullDump=False)
```

prints:

```
5+3*6
[[5, '+', [3, '*', 6]]]

(5+3)*6
[[[5, '+', 3], '*', 6]]

-2--11
[[['-', 2], '-', ['-', 11]]]
```

pyparsing.**locatedExpr**(*expr*)

Helper to decorate a returned token with its starting and ending locations in the input string.

This helper adds the following results names:

- locn_start = location where matched expression begins
- locn_end = location where matched expression ends
- value = the actual parsed results

Be careful if the input text contains <TAB> characters, you may want to call *ParserElement.parseWithTabs*

Example:

```
wd = Word(alphas)
for match in locatedExpr(wd).searchString("ljsdf123lksdjjf123lkkjj1222"):
    print(match)
```

prints:

```
[[0, 'ljsdf', 5]]
[[8, 'lksdjjf', 15]]
[[18, 'lkkjj', 23]]
```

pyparsing.**withClass**(*classname*, *namespace=''*)

Simplified version of *withAttribute* when matching on a div class - made difficult because `class` is a reserved word in Python.

Example:

```
html = '''
    <div>
    Some text
    <div class="grid">1 4 0 1 0</div>
    <div class="graph">1,3 2,3 1,1</div>
    <div>this &lt;div&gt; has no class</div>
    </div>

'''
div,div_end = makeHTMLTags("div")
div_grid = div().setParseAction(withClass("grid"))

grid_expr = div_grid + SkipTo(div | div_end)("body")
for grid_header in grid_expr.searchString(html):
    print(grid_header.body)

div_any_type = div().setParseAction(withClass(withAttribute.ANY_VALUE))
div_expr = div_any_type + SkipTo(div | div_end)("body")
for div_header in div_expr.searchString(html):
    print(div_header.body)
```

prints:

```
1 4 0 1 0

1 4 0 1 0
1,3 2,3 1,1
```

class pyparsing.**CloseMatch**(*match_string*, *maxMismatches=1*)

Bases: *pyparsing.Token*

A variation on *Literal* which matches "close" matches, that is, strings with at most 'n' mismatching characters. *CloseMatch* takes parameters:

- `match_string` - string to be matched
- `maxMismatches` - (`default=1`) maximum number of mismatches allowed to count as a match

The results from a successful parse will contain the matched text from the input string and the following named results:

- `mismatches` - a list of the positions within the match_string where mismatches were found
- `original` - the original match_string used to compare against the input string

If `mismatches` is an empty list, then the match was an exact match.

Example:

```
patt = CloseMatch("ATCATCGAATGGA")
patt.parseString("ATCATCGAAXGGA") # -> (['ATCATCGAAXGGA'], {'mismatches': [[9]],
↪'original': ['ATCATCGAATGGA']})
patt.parseString("ATCAXCGAAXGGA") # -> Exception: Expected 'ATCATCGAATGGA' (with␣
↪up to 1 mismatches) (at char 0), (line:1, col:1)

# exact match
patt.parseString("ATCATCGAATGGA") # -> (['ATCATCGAATGGA'], {'mismatches': [[]],
↪'original': ['ATCATCGAATGGA']})

# close match allowing up to 2 mismatches
patt = CloseMatch("ATCATCGAATGGA", maxMismatches=2)
patt.parseString("ATCAXCGAAXGGA") # -> (['ATCAXCGAAXGGA'], {'mismatches': [[4,␣
↪9]], 'original': ['ATCATCGAATGGA']})
```

parseImpl(*instring*, *loc*, *doActions=True*)

pyparsing.**tokenMap**(*func*, **args*)

Helper to define a parse action by mapping a function to all elements of a ParseResults list. If any additional args are passed, they are forwarded to the given function as additional arguments after the token, as in `hex_integer = Word(hexnums).setParseAction(tokenMap(int, 16))`, which will convert the parsed data to an integer using base 16.

Example (compare the last to example in *ParserElement.transformString*:

```
hex_ints = OneOrMore(Word(hexnums)).setParseAction(tokenMap(int, 16))
hex_ints.runTests('''
    00 11 22 aa FF 0a 0d 1a
    ''')

upperword = Word(alphas).setParseAction(tokenMap(str.upper))
OneOrMore(upperword).runTests('''
    my kingdom for a horse
    ''')

wd = Word(alphas).setParseAction(tokenMap(str.title))
OneOrMore(wd).setParseAction(' '.join).runTests('''
    now is the winter of our discontent made glorious summer by this sun of york
    ''')
```

prints:

```
00 11 22 aa FF 0a 0d 1a
[0, 17, 34, 170, 255, 10, 13, 26]

my kingdom for a horse
['MY', 'KINGDOM', 'FOR', 'A', 'HORSE']

now is the winter of our discontent made glorious summer by this sun of york
['Now Is The Winter Of Our Discontent Made Glorious Summer By This Sun Of York']
```

class pyparsing.**pyparsing_common**

Here are some common low-level expressions that may be useful in jump-starting parser development:

- numeric forms (*integers*, *reals*, *scientific notation*)
- common *programming identifiers*
- network addresses (*MAC*, *IPv4*, *IPv6*)
- ISO8601 *dates* and *datetime*
- *UUID*
- *comma-separated list*

Parse actions:

- *convertToInteger*
- *convertToFloat*
- *convertToDate*
- *convertToDatetime*
- *stripHTMLTags*
- *upcaseTokens*
- *downcaseTokens*

Example:

```
pyparsing_common.number.runTests('''
    # any int or real number, returned as the appropriate type
    100
    -100
    +100
    3.14159
    6.02e23
    1e-12
    ''')

pyparsing_common.fnumber.runTests('''
    # any int or real number, returned as float
    100
    -100
    +100
    3.14159
    6.02e23
    1e-12
    ''')

pyparsing_common.hex_integer.runTests('''
```

(continues on next page)

(continued from previous page)

```
    # hex numbers
    100
    FF
    ''')

pyparsing_common.fraction.runTests('''
    # fractions
    1/2
    -3/4
    ''')

pyparsing_common.mixed_integer.runTests('''
    # mixed fractions
    1
    1/2
    -3/4
    1-3/4
    ''')

import uuid
pyparsing_common.uuid.setParseAction(tokenMap(uuid.UUID))
pyparsing_common.uuid.runTests('''
    # uuid
    12345678-1234-5678-1234-567812345678
    ''')
```

prints:

```
# any int or real number, returned as the appropriate type
100
[100]

-100
[-100]

+100
[100]

3.14159
[3.14159]

6.02e23
[6.02e+23]

1e-12
[1e-12]

# any int or real number, returned as float
100
[100.0]

-100
[-100.0]

+100
[100.0]
```

(continues on next page)

(continued from previous page)

```
3.14159
[3.14159]

6.02e23
[6.02e+23]

1e-12
[1e-12]

# hex numbers
100
[256]

FF
[255]

# fractions
1/2
[0.5]

-3/4
[-0.75]

# mixed fractions
1
[1]

1/2
[0.5]

-3/4
[-0.75]

1-3/4
[1.75]

# uuid
12345678-1234-5678-1234-567812345678
[UUID('12345678-1234-5678-1234-567812345678')]
```

comma_separated_list = comma separated list
Predefined expression of 1 or more printable words or quoted strings, separated by commas.

static convertToDate (*fmt='%Y-%m-%d'*)
Helper to create a parse action for converting parsed date string to Python datetime.date

Params -

- fmt - format to be passed to datetime.strptime (default= `"%Y-%m-%d"`)

Example:

```
date_expr = pyparsing_common.iso8601_date.copy()
date_expr.setParseAction(pyparsing_common.convertToDate())
print(date_expr.parseString("1999-12-31"))
```

prints:

```
[datetime.date(1999, 12, 31)]
```

static convertToDatetime (*fmt='%Y-%m-%dT%H:%M:%S.%f'*)
Helper to create a parse action for converting parsed datetime string to Python datetime.datetime

Params -

- fmt - format to be passed to datetime.strptime (default= `"%Y-%m-%dT%H:%M:%S.%f"`)

Example:

```
dt_expr = pyparsing_common.iso8601_datetime.copy()
dt_expr.setParseAction(pyparsing_common.convertToDatetime())
print(dt_expr.parseString("1999-12-31T23:59:59.999"))
```

prints:

```
[datetime.datetime(1999, 12, 31, 23, 59, 59, 999000)]
```

convertToFloat (*l*, *t*)
Parse action for converting parsed numbers to Python float

convertToInteger (*l*, *t*)
Parse action for converting parsed integers to Python int

static downcaseTokens (*s*, *l*, *t*)
Parse action to convert tokens to lower case.

fnumber = fnumber
any int or real number, returned as float

fraction = fraction
fractional expression of an integer divided by an integer, returns a float

hex_integer = hex integer
expression that parses a hexadecimal integer, returns an int

identifier = identifier
typical code identifier (leading alpha or '_', followed by 0 or more alphas, nums, or '_')

integer = integer
expression that parses an unsigned integer, returns an int

ipv4_address = IPv4 address
IPv4 address (`0.0.0.0 - 255.255.255.255`)

ipv6_address = IPv6 address
IPv6 address (long, short, or mixed form)

iso8601_date = ISO8601 date
ISO8601 date (`yyyy-mm-dd`)

iso8601_datetime = ISO8601 datetime
ISO8601 datetime (`yyyy-mm-ddThh:mm:ss.s(Z|+-00:00)`) - trailing seconds, milliseconds, and timezone optional; accepts separating `'T'` or `' '`

mac_address = MAC address
MAC address xx:xx:xx:xx:xx (may also have '-' or '.' delimiters)

mixed_integer = fraction or mixed integer-fraction
mixed integer of the form 'integer - fraction', with optional leading integer, returns float

number = {real number with scientific notation | real number | signed integer}
any numeric expression, returns the corresponding Python type

real = real number
expression that parses a floating point number and returns a float

sci_real = real number with scientific notation
expression that parses a floating point number with optional scientific notation and returns a float

signed_integer = signed integer
expression that parses an integer with optional leading sign, returns an int

static stripHTMLTags(*s*, *l*, *tokens*)
Parse action to remove HTML tags from web page HTML source

Example:

```
# strip HTML links from normal text
text = '<td>More info at the <a href="https://github.com/pyparsing/pyparsing/
↪wiki">pyparsing</a> wiki page</td>'
td, td_end = makeHTMLTags("TD")
table_text = td + SkipTo(td_end).setParseAction(pyparsing_common.
↪stripHTMLTags)("body") + td_end
print(table_text.parseString(text).body)
```

Prints:

```
More info at the pyparsing wiki page
```

static upcaseTokens(*s*, *l*, *t*)
Parse action to convert tokens to upper case.

uuid = UUID
UUID (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)

class pyparsing.**pyparsing_unicode**
Bases: *pyparsing.unicode_set*

A namespace class for defining common language unicode_sets.

class Arabic
Bases: *pyparsing.unicode_set*

Unicode set for Arabic Unicode Character Range

class CJK
Bases: pyparsing.Chinese, pyparsing.Japanese, pyparsing.Korean

Unicode set for combined Chinese, Japanese, and Korean (CJK) Unicode Character Range

class Chinese
Bases: *pyparsing.unicode_set*

Unicode set for Chinese Unicode Character Range

class Cyrillic
Bases: *pyparsing.unicode_set*

Unicode set for Cyrillic Unicode Character Range

class Devanagari
Bases: *pyparsing.unicode_set*

Unicode set for Devanagari Unicode Character Range

class Greek
Bases: *pyparsing.unicode_set*

Unicode set for Greek Unicode Character Ranges

class Hebrew
Bases: *pyparsing.unicode_set*

Unicode set for Hebrew Unicode Character Range

class Japanese
Bases: *pyparsing.unicode_set*

Unicode set for Japanese Unicode Character Range, combining Kanji, Hiragana, and Katakana ranges

class Hiragana
Bases: *pyparsing.unicode_set*

Unicode set for Hiragana Unicode Character Range

class Kanji
Bases: *pyparsing.unicode_set*

Unicode set for Kanji Unicode Character Range

class Katakana
Bases: *pyparsing.unicode_set*

Unicode set for Katakana Unicode Character Range

class Korean
Bases: *pyparsing.unicode_set*

Unicode set for Korean Unicode Character Range

class Latin1
Bases: *pyparsing.unicode_set*

Unicode set for Latin-1 Unicode Character Range

class LatinA
Bases: *pyparsing.unicode_set*

Unicode set for Latin-A Unicode Character Range

class LatinB
Bases: *pyparsing.unicode_set*

Unicode set for Latin-B Unicode Character Range

class Thai
Bases: *pyparsing.unicode_set*

Unicode set for Thai Unicode Character Range

class pyparsing.**unicode_set**
Bases: object

A set of Unicode characters, for language-specific strings for `alphas`, `nums`, `alphanums`, and `printables`. A unicode_set is defined by a list of ranges in the Unicode character set, in a class attribute `_ranges`, such as:

```
_ranges = [(0x0020, 0x007e), (0x00a0, 0x00ff),]
```

A unicode set can also be defined using multiple inheritance of other unicode sets:

```
class CJK(Chinese, Japanese, Korean):
    pass
```

alphanums = u''

alphas = u''

nums = u''

printables = u''

pyparsing.**conditionAsParseAction**(*fn*, *message=None*, *fatal=False*)

CHAPTER 3

Contributor Covenant Code of Conduct

3.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

3.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

3.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

3.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at pyparsing@mail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

3.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Q

R

S

T

U

V

W

Z