PyParsing Documentation

Release 3.1.1

Paul T. McGuire

Sep 03, 2023

Contents:

Release v3.1.1

CHAPTER 1

1 What's New in Pyparsing 3.0.0

author Paul McGuire

date May, 2022

abstract This document summarizes the changes made in the 3.0.0 release of pyparsing. (Updated to reflect changes up to 3.0.10)

Contents

1.1 1.1 New Features

1.1.1 1.1.1 PEP-8 naming

This release of pyparsing will (finally!) include PEP-8 compatible names and arguments. Backward-compatibility is maintained by defining synonyms using the old camelCase names pointing to the new snake_case names.

This code written using non-PEP8 names:

```
wd = pp.Word(pp.printables, excludeChars="$")
wd_list = pp.delimitedList(wd, delim="$")
print(wd_list.parseString("dkls$134lkjk$lsd$$").asList())
```

can now be written as:

```
wd = pp.Word(pp.printables, exclude_chars="$")
wd_list = pp.delimited_list(wd, delim="$")
print(wd_list.parse_string("dkls$134lkjk$lsd$$").as_list())
```

Pyparsing 3.0 will run both versions of this example.

New code should be written using the PEP-8 compatible names. The compatibility synonyms will be removed in a future version of pyparsing.

1.1.2 1.1.2 Railroad diagramming

An excellent new enhancement is the new railroad diagram generator for documenting pyparsing parsers.:

```
import pyparsing as pp

# define a simple grammar for parsing street addresses such
# as "123 Main Street"
#     number word...
number = pp.Word(pp.nums).set_name("number")
name = pp.Word(pp.alphas).set_name("word")[1, ...]

parser = number("house_number") + name("street")
parser.set_name("street address")

# construct railroad track diagram for this parser and
```

(continues on next page)

(continued from previous page)

```
# save as HTML
parser.create_diagram('parser_rr_diag.html')
```

`create_diagram` accepts these named arguments:

- `vertical` (int) - threshold for formatting multiple alternatives vertically instead of horizontally (default=3)
- `show_results_names` - bool flag whether diagram should show annotations for defined results names
- `show_groups` - bool flag whether groups should be highlighted with an unlabeled surrounding box
- `embed` - bool flag whether generated HTML should omit `<HEAD>`, `<BODY>`, and `<DOCTYPE>` tags to embed the resulting HTML in an enclosing HTML source (new in 3.0.10)
- `head` - str containing additional HTML to insert into the `<HEAD>` section of the generated code; can be used to insert custom CSS styling
- `body` - str containing additional HTML to insert at the beginning of the `<BODY>` section of the generated code

To use this new feature, install the supporting diagramming packages using:

```
pip install pyparsing[diagrams]
```

See more in the examples directory: `make_diagram.py` and `railroad_diagram_demo.py`.

(Railroad diagram enhancement contributed by Michael Milton)

1.1.3 1.1.3 Support for left-recursive parsers

Another significant enhancement in 3.0 is support for left-recursive (LR) parsers. Previously, given a left-recursive parser, pyparsing would recurse repeatedly until hitting the Python recursion limit. Following the methods of the Python PEG parser, pyparsing uses a variation of packrat parsing to detect and handle left-recursion during parsing.:

```
import pyparsing as pp
pp.ParserElement.enable_left_recursion()

# a common left-recursion definition
# define a list of items as 'list + item | item'
# BNF:
#   item_list := item_list item | item
#   item := word of alphas
item_list = pp.Forward()
item = pp.Word(pp.alphas)
item_list <<= item_list + item | item

item_list.run_tests("""\
    To parse or not to parse that is the question
    """)
```

Prints:

```
['To', 'parse', 'or', 'not', 'to', 'parse', 'that', 'is', 'the', 'question']
```

See more examples in `left_recursion.py` in the pyparsing examples directory.

(LR parsing support contributed by Max Fischer)

1.1.4 1.1.4 Packrat/memoization enable and disable methods

As part of the implementation of left-recursion support, new methods have been added to enable and disable packrat parsing.

Name	Description
enable_packrat	Enable packrat parsing (with specified cache size)
enable_left_recursion	Enable left-recursion cache
disable_memoization	Disable all internal parsing caches

1.1.5 1.1.5 Type annotations on all public methods

Python 3.6 and upward compatible type annotations have been added to most of the public methods in pyparsing. This should facilitate developing pyparsing-based applications using IDEs for development-time type checking.

1.1.6 1.1.6 New string constants `identchars` and `identbodychars` to help in defining identifier Word expressions

Two new module-level strings have been added to help when defining identifiers, `identchars` and `identbodychars`.

Instead of writing:

```
import pyparsing as pp
identifier = pp.Word(pp.alphas + "_", pp.alphanums + "_")
```

you will be able to write:

```
identifier = pp.Word(pp.identchars, pp.identbodychars)
```

Those constants have also been added to all the Unicode string classes:

```
import pyparsing as pp
ppu = pp.pyparsing_unicode

cjk_identifier = pp.Word(ppu.CJK.identchars, ppu.CJK.identbodychars)
greek_identifier = pp.Word(ppu.Greek.identchars, ppu.Greek.identbodychars)
```

1.1.7 1.1.7 Refactored/added diagnostic flags

Expanded `__diag__` and `__compat__` to actual classes instead of just namespaces, to add some helpful behavior:

- `pyparsing.enable_diag()` and `pyparsing.disable_diag()` methods to give extra help when setting or clearing flags (detects invalid flag names, detects when trying to set a `__compat__` flag that is no longer settable). Use these methods now to set or clear flags, instead of directly setting to `True` or `False`:

  ```
  import pyparsing as pp
  pp.enable_diag(pp.Diagnostics.warn_multiple_tokens_in_named_alternation)
  ```

- `pyparsing.enable_all_warnings()` is another helper that sets all "warn*" diagnostics to `True`:

  ```
  pp.enable_all_warnings()
  ```

- added support for calling `enable_all_warnings()` if warnings are enabled using the Python `-W` switch, or setting a non-empty value to the environment variable `PYPARSINGENABLEALLWARNINGS`. (If using `-Wd` for testing, but wishing to disable pyparsing warnings, add `-Wi::pyparsing`.)
- added new warning, `warn_on_match_first_with_lshift_operator` to warn when using `'<<'` with a `'|'` `MatchFirst` operator, which will create an unintended expression due to precedence of operations.

  Example: This statement will erroneously define the `fwd` expression as just `expr_a`, even though `expr_a | expr_b` was intended, since `'<<'` operator has precedence over `'|'`:

```
fwd << expr_a | expr_b
```

  To correct this, use the `'<<='` operator (preferred) or parentheses to override operator precedence:

```
fwd <<= expr_a | expr_b
```

  or:

```
fwd << (expr_a | expr_b)
```

- `warn_on_parse_using_empty_Forward` - warns that a `Forward` has been included in a grammar, but no expression was attached to it using `'<<='` or `'<<'`
- `warn_on_assignment_to_Forward` - warns that a `Forward` has been created, but was probably later overwritten by erroneously using `'='` instead of `'<<='` (this is a common mistake when using Forwards) (**currently not working on PyPy**)

1.1.8 1.1.8 Support for yielding native Python `list` and `dict` types in place of `ParseResults`

To support parsers that are intended to generate native Python collection types such as lists and dicts, the `Group` and `Dict` classes now accept an additional boolean keyword argument `aslist` and `asdict` respectively. See the `jsonParser.py` example in the `pyparsing/examples` source directory for how to return types as `ParseResults` and as Python collection types, and the distinctions in working with the different types.

In addition parse actions that must return a value of list type (which would normally be converted internally to a `ParseResults`) can override this default behavior by returning their list wrapped in the new `ParseResults.List` class:

```
# this parse action tries to return a list, but pyparsing
# will convert to a ParseResults
def return_as_list_but_still_get_parse_results(tokens):
    return tokens.asList()

# this parse action returns the tokens as a list, and pyparsing will
# maintain its list type in the final parsing results
def return_as_list(tokens):
    return ParseResults.List(tokens.asList())
```

This is the mechanism used internally by the `Group` class when defined using `aslist=True`.

1.1.9 1.1.9 New Located class to replace `locatedExpr` helper method

The new `Located` class will replace the current `locatedExpr` method for marking parsed results with the start and end locations of the parsed data in the input string. `locatedExpr` had several bugs, and returned its results in

a hard-to-use format (location data and results names were mixed in with the located expression's parsed results, and wrapped in an unnecessary extra nesting level).

For this code:

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

the docs for `locatedExpr` show this output:

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

The parsed values and the start and end locations are merged into a single nested `ParseResults` (and any results names in the parsed values are also merged in with the start and end location names).

Using `Located`, the output is:

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

With `Located`, the parsed expression values and results names are kept separate in the second parsed value, and there is no extra grouping level on the whole result.

The existing `locatedExpr` is retained for backward-compatibility, but will be deprecated in a future release.

1.1.10 1.1.10 New `AtLineStart` and `AtStringStart` classes

As part of fixing some matching behavior in `LineStart` and `StringStart`, two new classes have been added: `AtLineStart` and `AtStringStart`.

`LineStart` and `StringStart` can be treated as separate elements, including whitespace skipping. `AtLineStart` and `AtStringStart` enforce that an expression starts exactly at column 1, with no leading whitespace.:

```
(LineStart() + Word(alphas)).parseString("ABC")     # passes
(LineStart() + Word(alphas)).parseString("  ABC")   # passes
AtLineStart(Word(alphas)).parseString("  ABC")      # fails
```

[This is a fix to behavior that was added in 3.0.0, but was actually a regression from 2.4.x.]

1.1.11 1.1.11 New `IndentedBlock` class to replace `indentedBlock` helper method

The new `IndentedBlock` class will replace the current `indentedBlock` method for defining indented blocks of text, similar to Python source code. Using `IndentedBlock`, the expression instance itself keeps track of the indent stack, so a separate external `indentStack` variable is no longer required.

Here is a simple example of an expression containing an alphabetic key, followed by an indented list of integers:

```
integer = pp.Word(pp.nums)
group = pp.Group(pp.Char(pp.alphas) + pp.IndentedBlock(integer))
```

parses:

```
A
    100
    101
B
    200
    201
```

as:

```
[['A', [100, 101]], ['B', [200, 201]]]
```

By default, the results returned from the `IndentedBlock` are grouped.

`IndentedBlock` may also be used to define a recursive indented block (containing nested indented blocks).

The existing `indentedBlock` is retained for backward-compatibility, but will be deprecated in a future release.

1.1.12 1.1.12 Shortened tracebacks

Cleaned up default tracebacks when getting a `ParseException` when calling `parse_string`. Exception traces should now stop at the call in `parse_string`, and not include the internal pyparsing traceback frames. (If the full traceback is desired, then set `ParserElement.verbose_traceback` to `True`.)

1.1.13 1.1.13 Improved debug logging

Debug logging has been improved by:

- Including `try/match/fail` logging when getting results from the packrat cache (previously cache hits did not show debug logging). Values returned from the packrat cache are marked with an '*'.
- Improved fail logging, showing the failed expression, text line, and marker where the failure occurred.
- Adding `with_line_numbers` to `pyparsing_testing`. Use `with_line_numbers` to visualize the data being parsed, with line and column numbers corresponding to the values output when enabling `set_debug()` on an expression:

```
data = """\
   A
      100"""
expr = pp.Word(pp.alphanums).set_name("word").set_debug()
print(ppt.with_line_numbers(data))
expr[...].parseString(data)
```

prints:

```
.          1
  1234567890
1:   A
2:      100
Match word at loc 3(1,4)
    A
    ^
Matched word -> ['A']
Match word at loc 11(2,7)
       100
       ^
Matched word -> ['100']
```

1.1.14 1.1.14 New / improved examples

- `number_words.py` includes a parser/evaluator to parse `"forty-two"` and return `42`. Also includes example code to generate a railroad diagram for this parser.
- `BigQueryViewParser.py` added to examples directory, submitted by Michael Smedberg.
- `booleansearchparser.py` added to examples directory, submitted by xecgr. Builds on searchparser.py, adding support for '*' wildcards and non-Western alphabets.
- Improvements in `select_parser.py`, to include new SQL syntax from SQLite, submitted by Robert Coup.
- Off-by-one bug found in the `roman_numerals.py` example, a bug that has been there for about 14 years! Submitted by Jay Pedersen.
- A simplified Lua parser has been added to the examples (`lua_parser.py`).
- Demonstration of defining a custom Unicode set for cuneiform symbols, as well as simple Cuneiform->Python conversion is included in `cuneiform_python.py`.
- Fixed bug in `delta_time.py` example, when using a quantity of seconds/minutes/hours/days > 999.

1.1.15 1.1.15 Other new features

- `url` expression added to `pyparsing_common`, with named fields for common fields in URLs. See the updated `urlExtractorNew.py` file in the `examples` directory. Submitted by Wolfgang Fahl.
- `DelimitedList` now supports an additional flag `allow_trailing_delim`, to optionally parse an additional delimiter at the end of the list. Submitted by Kazantcev Andrey.
- Added global method `autoname_elements()` to call `set_name()` on all locally defined `ParserElements` that haven't been explicitly named using `set_name()`, using their local variable name. Useful for setting names on multiple elements when creating a railroad diagram:

```
a = pp.Literal("a")
b = pp.Literal("b").set_name("bbb")
pp.autoname_elements()
```

  `a` will get named "a", while `b` will keep its name "bbb".

- Enhanced default strings created for `Word` expressions, now showing string ranges if possible. `Word(alphas)` would formerly print as `W:(ABCD...)`, now prints as `W:(A-Za-z)`.
- Better exception messages to show full word where an exception occurred.:

```
Word(alphas)[...].parse_string("abc 123", parse_all=True)
```

  Was:

```
pyparsing.ParseException: Expected end of text, found '1'  (at char 4), (line:1,␣
↪col:5)
```

  Now:

```
pyparsing.exceptions.ParseException: Expected end of text, found '123'  (at char␣
↪4), (line:1, col:5)
```

- Using `...` for `SkipTo` can now be wrapped in `Suppress` to suppress the skipped text from the returned parse results.:

```
source = "lead in START relevant text END trailing text"
start_marker = Keyword("START")
end_marker = Keyword("END")
find_body = Suppress(...) + start_marker + ... + end_marker
print(find_body.parse_string(source).dump())
```

Prints:

```
['START', 'relevant text ', 'END']
- _skipped: ['relevant text ']
```

- Added `ignore_whitespace(recurse:bool = True)` and added a `recurse` argument to `leave_whitespace`, both added to provide finer control over pyparsing's whitespace skipping. Contributed by Michael Milton.
- Added `ParserElement.recurse()` method to make it simpler for grammar utilities to navigate through the tree of expressions in a pyparsing grammar.
- The `repr()` string for `ParseResults` is now of the form:

```
ParseResults([tokens], {named_results})
```

  The previous form omitted the leading `ParseResults` class name, and was easily misinterpreted as a `tuple` containing a `list` and a `dict`.

- Minor reformatting of output from `run_tests` to make embedded comments more visible.
- New `pyparsing_test` namespace, assert methods and classes added to support writing unit tests.
  - `assertParseResultsEquals`
  - `assertParseAndCheckList`
  - `assertParseAndCheckDict`
  - `assertRunTestResults`
  - `assertRaisesParseException`
  - `reset_pyparsing_context` context manager, to restore pyparsing config settings
- Enhanced error messages and error locations when parsing fails on the `Keyword` or `CaselessKeyword` classes due to the presence of a preceding or trailing keyword character.
- Enhanced the `Regex` class to be compatible with re's compiled with the re-equivalent `regex` module. Individual expressions can be built with regex compiled expressions using:

```
import pyparsing as pp
import regex

# would use regex for this expression
integer_parser = pp.Regex(regex.compile(r'\d+'))
```

- Fixed handling of `ParseSyntaxExceptions` raised as part of `Each` expressions, when sub-expressions contain `'-'` backtrack suppression.
- Potential performance enhancement when parsing `Word` expressions built from `pyparsing_unicode` character sets. `Word` now internally converts ranges of consecutive characters to regex character ranges (converting `"0123456789"` to `"0-9"` for instance).
- Added a caseless parameter to the `CloseMatch` class to allow for casing to be ignored when checking for close matches. Contributed by Adrian Edwards.

1.2 1.2 API Changes

- [Note added in pyparsing 3.0.7, reflecting a change in 3.0.0] Fixed a bug in the `ParseResults` class implementation of `__bool__`, which would formerly return `False` if the `ParseResults` item list was empty, even if it contained named results. Now `ParseResults` will return `True` if either the item list is not empty *or* if the named results list is not empty:

```
# generate an empty ParseResults by parsing a blank string with a ZeroOrMore
result = Word(alphas)[...].parse_string("")
print(result.as_list())
print(result.as_dict())
print(bool(result))

# add a results name to the result
result["name"] = "empty result"
print(result.as_list())
print(result.as_dict())
print(bool(result))
```

Prints:

```
[]
{}
False

[]
{'name': 'empty result'}
True
```

In previous versions, the second call to `bool()` would return `False`.

- [Note added in pyparsing 3.0.4, reflecting a change in 3.0.0] The `ParseResults` class now uses `__slots__` to pre-define instance attributes. This means that code written like this (which was allowed in pyparsing 2.4.7):

```
result = Word(alphas).parseString("abc")
result.xyz = 100
```

now raises this Python exception:

```
AttributeError: 'ParseResults' object has no attribute 'xyz'
```

To add new attribute values to ParseResults object in 3.0.0 and later, you must assign them using indexed notation:

```
result["xyz"] = 100
```

You will still be able to access this new value as an attribute or as an indexed item.

- `enable_diag()` and `disable_diag()` methods to enable specific diagnostic values (instead of setting them to `True` or `False`). `enable_all_warnings()` has also been added.
- `counted_array` formerly returned its list of items nested within another list, so that accessing the items required indexing the 0'th element to get the actual list. This extra nesting has been removed. In addition, if there are other metadata fields parsed between the count and the list items, they can be preserved in the resulting list if given results names.
- `ParseException.explain()` is now an instance method of `ParseException`:

```
expr = pp.Word(pp.nums) * 3
try:
    expr.parse_string("123 456 A789")
except pp.ParseException as pe:
    print(pe.explain(depth=0))
```

prints:

```
123 456 A789
        ^
ParseException: Expected W:(0-9), found 'A789'  (at char 8), (line:1, col:9)
```

To run explain against other exceptions, use `ParseException.explain_exception()`.

- Debug actions now take an added keyword argument `cache_hit`. Now that debug actions are called for expressions matched in the packrat parsing cache, debug actions are now called with this extra flag, set to `True`. For custom debug actions, it is necessary to add support for this new argument.
- `ZeroOrMore` expressions that have results names will now include empty lists for their name if no matches are found. Previously, no named result would be present. Code that tested for the presence of any expressions using `"if name in results:"` will now always return `True`. This code will need to change to `"if name in results and results[name]:"` or just `"if results[name]:"`. Also, any parser unit tests that check the `as_dict()` contents will now see additional entries for parsers having named `ZeroOrMore` expressions, whose values will be `[]`.
- `ParserElement.set_default_whitespace_chars` will now update whitespace characters on all built-in expressions defined in the pyparsing module.
- `camelCase` names have been converted to PEP-8 `snake_case` names.

  Method names and arguments that were camel case (such as `parseString`) have been replaced with PEP-8 snake case versions (`parse_string`).

  Backward-compatibility synonyms for all names and arguments have been included, to allow parsers written using the old names to run without change. The synonyms will be removed in a future release. New parser code should be written using the new PEP-8 snake case names.

Name	Previous name
ParserElement	
• parse_string	parseString
• scan_string	scanString
• search_string	searchString
• transform_string	transformString
• add_condition	addCondition

Continued on next page

Table 1 – continued from previous page

• add_parse_action	addParseAction
• can_parse_next	canParseNext
• default_name	defaultName
• enable_left_recursion	enableLeftRecursion
• enable_packrat	enablePackrat
• ignore_whitespace	ignoreWhitespace
• inline_literals_using	inlineLiteralsUsing
• parse_file	parseFile
• leave_whitespace	leaveWhitespace
• parse_string	parseString
• parse_with_tabs	parseWithTabs
• reset_cache	resetCache
• run_tests	runTests
• scan_string	scanString
• search_string	searchString
• set_break	setBreak
• set_debug	setDebug

Continued on next page

Table 1 – continued from previous page

• set_debug_actions	setDebugActions
• set_default_whitespace_chars	setDefaultWhitespaceChars
• set_fail_action	setFailAction
• set_name	setName
• set_parse_action	setParseAction
• set_results_name	setResultsName
• set_whitespace_chars	setWhitespaceChars
• transform_string	transformString
• try_parse	tryParse
ParseResults	
• as_list	asList
• as_dict	asDict
• get_name	getName
ParseBaseException	
• parser_element	parserElement
any_open_tag	anyOpenTag
any_close_tag	anyCloseTag
c_style_comment	cStyleComment
common_html_entity	commonHTMLEntity
condition_as_parse_action	conditionAsParseAction
counted_array	countedArray
cpp_style_comment	cppStyleComment
dbl_quoted_string	dblQuotedString
dbl_slash_comment	dblSlashComment
DelimitedList	delimitedList

Continued on next page

Table 1 – continued from previous page

DelimitedList	delimited_list
dict_of	dictOf
html_comment	htmlComment
infix_notation	infixNotation
java_style_comment	javaStyleComment
line_end	lineEnd
line_start	lineStart
make_html_tags	makeHTMLTags
make_xml_tags	makeXMLTags
match_only_at_col	matchOnlyAtCol
match_previous_expr	matchPreviousExpr
match_previous_literal	matchPreviousLiteral
nested_expr	nestedExpr
null_debug_action	nullDebugAction
one_of	oneOf
OpAssoc	opAssoc
original_text_for	originalTextFor
python_style_comment	pythonStyleComment
quoted_string	quotedString
remove_quotes	removeQuotes
replace_html_entity	replaceHTMLEntity
replace_with	replaceWith
rest_of_line	restOfLine
sgl_quoted_string	sglQuotedString
string_end	stringEnd
string_start	stringStart
token_map	tokenMap
trace_parse_action	traceParseAction
unicode_string	unicodeString
with_attribute	withAttribute
with_class	withClass

1.3 1.3 Discontinued Features

1.3.1 1.3.1 Python 2.x no longer supported

Removed Py2.x support and other deprecated features. Pyparsing now requires Python 3.6.8 or later. If you are using an earlier version of Python, you must use a Pyparsing 2.4.x version.

1.3.2 1.3.2 Other discontinued features

- `ParseResults.asXML()` - if used for debugging, switch to using `ParseResults.dump()`; if used for data transfer, use `ParseResults.as_dict()` to convert to a nested Python dict, which can then be converted to XML or JSON or other transfer format
- `operatorPrecedence` synonym for `infixNotation` - convert to calling `infix_notation`
- `commaSeparatedList` - convert to using `pyparsing_common.comma_separated_list`

- `upcaseTokens` and `downcaseTokens` - convert to using `pyparsing_common.upcase_tokens` and `downcase_tokens`
- `__compat__.collect_all_And_tokens` will not be settable to `False` to revert to pre-2.3.1 results name behavior - review use of names for `MatchFirst` and Or expressions containing `And` expressions, as they will return the complete list of parsed tokens, not just the first one. Use `pyparsing.enable_diag(pyparsing.Diagnostics.warn_multiple_tokens_in_named_alternation)` to help identify those expressions in your parsers that will have changed as a result.
- Removed support for running `python setup.py test`. The setuptools maintainers consider the `test` command deprecated (see <https://github.com/pypa/setuptools/issues/1684>). To run the Pyparsing tests, use the command `tox`.

1.4 1.4 Fixed Bugs

- [Reverted in 3.0.2]Fixed issue when `LineStart()` expressions would match input text that was not necessarily at the beginning of a line.

  [The previous behavior was the correct behavior, since it represents the `LineStart` as its own matching expression. `ParserElements` that must start in column 1 can be wrapped in the new `AtLineStart` class.]
- Fixed bug in regex definitions for `real` and `sci_real` expressions in `pyparsing_common`.
- Fixed `FutureWarning` raised beginning in Python 3.7 for `Regex` expressions containing '[' within a regex set.
- Fixed bug in `PrecededBy` which caused infinite recursion.
- Fixed bug in `CloseMatch` where end location was incorrectly computed; and updated `partial_gene_match.py` example.
- Fixed bug in `indentedBlock` with a parser using two different types of nested indented blocks with different indent values, but sharing the same indent stack.
- Fixed bug in `Each` when using `Regex`, when `Regex` expression would get parsed twice.
- Fixed bugs in `Each` when passed `OneOrMore` or `ZeroOrMore` expressions: . first expression match could be enclosed in an extra nesting level . out-of-order expressions now handled correctly if mixed with required expressions . results names are maintained correctly for these expression
- Fixed `FutureWarning` that sometimes is raised when `'['` passed as a character to `Word`.
- Fixed debug logging to show failure location after whitespace skipping.
- Fixed `ParseFatalExceptions` failing to override normal exceptions or expression matches in `MatchFirst` expressions.
- Fixed bug in which `ParseResults` replaces a collection type value with an invalid type annotation (as a result of changed behavior in Python 3.9).
- Fixed bug in `ParseResults` when calling `__getattr__` for special double-underscored methods. Now raises `AttributeError` for non-existent results when accessing a name starting with '__'.
- Fixed bug in `Located` class when used with a results name.
- Fixed bug in `QuotedString` class when the escaped quote string is not a repeated character.

1.5 1.5 Acknowledgments

And finally, many thanks to those who helped in the restructuring of the pyparsing code base as part of this release. Pyparsing now has more standard package structure, more standard unit tests, and more standard code formatting (using black). Special thanks to jdufresne, klahnakoski, mattcarmody, ckeygusuz, tmiguelt, and toonarmycaptain to name just a few.

Thanks also to Michael Milton and Max Fischer, who added some significant new features to pyparsing.

CHAPTER 2

1 Using the pyparsing module

author Paul McGuire

address ptmcg.pm+pyparsing@gmail.com

revision 3.1.1

date July, 2023

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 extensive examples at the online doc server, and in the online help for the various pyparsing classes and methods (viewable using the Python interpreter's built-in `help()` function). You will also find many example scripts in the examples directory of the pyparsing GitHub repo.

Note: *In pyparsing 3.0, many method and function names which were originally written using camelCase have been converted to PEP8-compatible snake_case. So ''parseString()'' is being renamed to ''parse_string()'', ''delimitedList'' to DelimitedList_, and so on. You may see the old names in legacy parsers, and they will be supported for a time with synonyms, but the synonyms will be removed in a future release.*

If you are using this documentation, but working with a 2.4.x version of pyparsing, you'll need to convert methods and arguments from the documented snake_case names to the legacy camelCase names. In pyparsing 3.0.x and 3.1.x, both forms are supported, but the legacy forms are deprecated; they will be dropped in a future release.

2.1 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 parse actions can also be defined at this time.
2. Call `parse_string()`, `scan_string()`, or `search_string()` 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 *ParseResults* object. The *ParseResults* object can be accessed as if it were 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 `set_results_name()`.

2.1.1 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>!":

```
import pyparsing as pp

greet = pp.Word(pp.alphas) + "," + pp.Word(pp.alphas) + "!"
for greeting_str in [
            "Hello, World!",
            "Bonjour, Monde!",
            "Hola, Mundo!",
            "Hallo, Welt!",
        ]:
    greeting = greet.parse_string(greeting_str)
    print(greeting)
```

The parsed tokens are returned in the following form:

```
['Hello', ',', 'World', '!']
['Bonjour', ',', 'Monde', '!']
['Hola', ',', 'Mundo', '!']
['Hallo', ',', 'Welt', '!']
```

2.1.2 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 = Char(alphas)          # single letter variable, such as x, z, m, etc.
  arith_op = one_of("+ - * /")      # arithmetic operators
  equation = variable + "=" + integer + arith_op + 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, and simpleArith.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.set_default_whitespace_chars` to override the normal set of whitespace chars (`' \t\n'`). For instance when defining a grammar in which newlines are significant, you should call `ParserElement.set_default_whitespace_chars(' \t')` to remove newline from the set of skippable whitespace characters. Calling this method will affect all pyparsing expressions defined afterward.
  - call `leave_whitespace()` 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 + Opt(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[...]`, `expr[0, ...]` and `expr * ...` 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`.
- `[]` notation will also accept a stop expression using ':' slice notation:
  - `expr[...:end_expr]` is equivalent to `ZeroOrMore(expr, stop_on=end_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 `parse_file` method using:

```
expr.parse_file(source_file)
```

- `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 `set_results_name` call:

```
stats = ("AVE:" + real_num.set_results_name("average")
         + "MIN:" + real_num.set_results_name("min")
         + "MAX:" + real_num.set_results_name("max"))
```

  can more simply and cleanly be written as this:

```
stats = ("AVE:" + real_num("average")
         + "MIN:" + real_num("min")
         + "MAX:" + real_num("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.

2.2 1.2 Classes

All the pyparsing classes can be found in this UML class diagram.

2.2.1 1.2.1 Classes in the pyparsing module

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

- `parse_string(source_string, parse_all=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 `parse_all` is set to True, then `parse_string` will raise a *ParseException* if the grammar does not process the complete input string.

- `parse_file(source_file)` - a convenience function, that accepts an input file object or filename. The file contents are passed as a string to `parse_string()`. `parse_file` also supports the `parse_all` argument.

- `scan_string(source_string)` - 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

  `scan_string` 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 `parse_string`).
- `transform_string(source_string)` - convenience wrapper function for `scan_string`, to process the input source string, and replace matching text with the tokens returned from parse actions defined in the grammar (see *set_parse_action*).
- `search_string(source_string)` - another convenience wrapper function for `scan_string`, returns a list of the matching tokens returned from each call to `scan_string`.
- `set_name(name)` - associate a short descriptive name for this element, useful in displaying exceptions and trace information
- `run_tests(tests_string)` - useful development and testing method on expressions, to pass a multiline string of sample strings to test against the expression. Comment lines (beginning with `#`) can be inserted and they will be included in the test output:

```
digits = Word(nums).set_name("numeric digits")
real_num = Combine(digits + '.' + digits)
real_num.run_tests("""\
    # valid number
    3.14159

    # no integer part
    .00001

    # no decimal
    101

    # no decimal value
    101.
    """)
```

will print:

```
# valid number
3.14159
['3.14159']

# no integer part
.00001
^
FAIL: Expected numeric digits, found '.'  (at char 0), (line:1, col:1)

# no decimal
101
   ^
FAIL: Expected ".", found end of text  (at char 3), (line:1, col:4)

# no decimal value
101.
```

(continues on next page)

(continued from previous page)

```
    ^
FAIL: Expected numeric digits, found end of text  (at char 4), (line:1, col:5)
```

- `set_results_name(string, list_all_matches=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 `list_all_matches` is set to True, then a list of all the matching tokens is returned.

  `expr.set_results_name("key")` can also be written `expr("key")` (a results name with a trailing '*' character will be interpreted as setting `list_all_matches` to `True`).

  Note: `set_results_name` 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.

- `set_parse_action(*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

  Parse actions can have any of the following signatures:

```
fn(s: str, loc: int, tokens: ParseResults)
fn(loc: int, tokens: ParseResults)
fn(tokens: ParseResults)
fn()
```

  Multiple functions can be attached to a `ParserElement` by specifying multiple arguments to `set_parse_action`, or by calling `add_parse_action`. Calls to `set_parse_action` will replace any previously defined parse actions. `set_parse_action(None)` will clear all previously defined parse actions.

  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:

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

  If `fn` modifies the `toks` list in-place, it does not need to return and pyparsing will use the modified `toks` list.

  If `set_parse_action` is called with an argument of `None`, then this clears all parse actions attached to that expression.

  A nice short-cut for calling `set_parse_action` is to use it as a decorator:

```
identifier = Word(alphas, alphanums+"_")

@identifier.set_parse_action
def resolve_identifier(results: ParseResults):
    return variable_values.get(results[0])
```

  (Posted by @MisterMiyagi in this SO answer: https://stackoverflow.com/a/63031959/165216)

- `add_parse_action` - similar to `set_parse_action`, but instead of replacing any previously defined parse actions, will append the given action or actions to the existing defined parse actions.

- `add_condition` - a simplified form of `add_parse_action` if the purpose of the parse action is to simply do some validation, and raise an exception if the validation fails. Takes a method that takes the same arguments, but simply returns `True` or `False`. If `False` is returned, an exception will be raised.
- `set_break(break_flag=True)` - if `break_flag` 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; a short-form `expr()` is equivalent to `expr.copy()`
- `leave_whitespace()` - change default behavior of skipping whitespace before starting matching (mostly used internally to the pyparsing module, rarely used by client code)
- `set_whitespace_chars(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)
- `set_default_whitespace_chars(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
- `set_debug(flag=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 (`validate()` is deprecated, and will be removed in a future pyparsing release. Pyparsing now supports left-recursive parsers, which this function attempted to catch.)
- `parse_with_tabs()` - 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.
- `enable_packrat()` - 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.enable_packrat()`. For best results, call `enable_packrat()` immediately after importing pyparsing.
- `enable_left_recursion()` - a class-level static method to enable pyparsing with left-recursive (LR) parsers. Similar to `ParserElement.enable_packrat()`, your program must call the class method `ParserElement.enable_left_recursion()` to enable this feature. `enable_left_recursion()` uses a separate packrat cache, and so is incompatible with `enable_packrat()`.

2.2.2 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 as described in *CaselessLiteral*.

- `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_]"))
```

  Pyparsing also provides pre-defined strings `identchars` and `identbodychars` so that you can also write:

```
Word(identchars, identbodychars)
```

  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 one of:

```
``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`
  - `as_keyword` - indicating that preceding and following characters must be whitespace or non-keyword characters
  - `exclude_chars` - a string of characters that should be excluded from init_chars and body_chars

    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 `exclude_chars` 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, exclude_chars='.')`.

- `Char` - a convenience form of `Word` that will match just a single character from a string of matching characters:

```
single_digit = Char(nums)
```

- `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 `dbl_quoted_string` and `sgl_quoted_string`. `QuotedString` allows you to specify the following parameters:
  - `quote_char` - string of one or more characters defining the quote delimiting string
  - `esc_char` - character to escape quotes, typically backslash (default=None)

  - `esc_quote` - 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)
  - `unquote_results` - boolean indicating whether the matched text should be unquoted (default=True)
  - `end_quote_char` - string of one or more characters defining the end of the quote delimited string (default=None => same as `quote_char`)

- `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
  - `fail_on` - 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

  `SkipTo` can also be written using `...`:

```
LBRACE, RBRACE = map(Literal, "{}")

brace_expr = LBRACE + SkipTo(RBRACE) + RBRACE
# can also be written as
brace_expr = LBRACE + ... + RBRACE
```

- `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 *parse_with_tabs* 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

2.2.3 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:

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

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

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

  A special form of `And` is created if the '-' operator is used instead of the '+' operator. In the `ip_address` 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 `ip_address`. However, if `ip_address` is defined as:

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

  then no backing up is done. If the first `Word(nums)` of `strict_ip_address` 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
- `Opt` - 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.

  An optional element `expr` can also be expressed using `expr | ""`.

  (`Opt` was formerly named `Optional`, but since the standard Python library module `typing` now defines `Optional`, the pyparsing class has been renamed to `Opt`. A compatibility synonym `Optional` is defined, but will be removed in a future release.)
- `ZeroOrMore` - similar to `Opt`, but can be repeated; `ZeroOrMore(expr)` can also be written as `expr[...]`.
- `OneOrMore` - similar to *ZeroOrMore*, but at least one match must be present; `OneOrMore(expr)` can also be written as `expr[1, ...]`.
- `DelimitedList` - used 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"`). Can also optionally specify `min` and `max` restrictions on the length of the list, and `allow_trailing_delim` to accept a trailing delimiter at the end of the list.
- `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

2.2.4 1.2.4 Expression operators

- `+` - 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`, interpret as `(0,max)`; if `max` is `None`, interpret as `expr*min + ZeroOrMore(expr)`
- `-` - like `+` but with no backup and retry of alternatives
- `~` - creates *NotAny* using the expression after the operator
- `==` - 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 (`<<` can also be used, but `<<=` is preferred to avoid operator precedence misinterpretation of the pyparsing expression)
- `...` - inserts a *SkipTo* expression leading to the next expression, as in `Keyword("start") + ... + Keyword("end")`.
- `[min, max]` - specifies repetition similar to `*` with `min` and `max` specified as the minimum and maximum number of repetitions. `...` can be used in place of `None`. For example `expr[...]` is equivalent to `ZeroOrMore(expr)`, `expr[1, ...]` is equivalent to `OneOrMore(expr)`, and `expr[..., 3]` is equivalent to "up to 3 instances of `expr`".

2.2.5 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

2.2.6 1.2.6 Converter subclasses

- `Combine` - joins all matched tokens into a single string, using specified `join_string` (default `join_string=""`); 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)

2.2.7 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
- `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).

2.2.8 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., or retrieved using slices

    - elements can be deleted using `del`
    - the last element can be extracted and removed in a single operation using `pop()`, or any element can be extracted and removed using `pop(n)`
    - a nested *ParseResults* can be created by using the pyparsing `Group` class around elements in an expression:

```
Word(alphas) + Group(Word(nums)[...]) + Word(alphas)
```

      will parse the string "abc 100 200 300 end" as:

```
['abc', ['100', '200', '300'], 'end']
```

      If the `Group` is constructed using `aslist=True`, the resulting tokens will be a Python list instead of a *ParseResults*. In this case, the returned value will no longer support the extended features or methods of a *ParseResults*.

- as a dictionary
    - if `set_results_name()` 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; when constructed with `asdict=True`, will return an actual Python `dict` instead of a *ParseResults*. In this case, the returned value will no longer support the extended features or methods of a *ParseResults*.
    - 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 (`parse_results["X"] = "new item"`); named elements can be removed using `del parse_results["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
- as an object
    - named elements can be accessed as if they were attributes of an object: if an element is referenced that does not exist, it will return `""`.

ParseResults can also be converted to an ordinary list of strings by calling `as_list()`. 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 `as_list()`.)

If a *ParseResults* is built with expressions that use results names (see _set_results_name) or using the `Dict` class, then those names and values can be extracted as a Python dict using `as_dict()`.

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

Here is sample code illustrating some of these methods:

```
>>> number = Word(nums)
>>> name = Combine(Word(alphas)[...], adjacent=False, join_string=" ")
>>> parser = number("house_number") + name("street_name")
>>> result = parser.parse_string("123 Main St")
>>> print(result)
['123', 'Main St']
>>> print(type(result))
<class 'pyparsing.ParseResults'>
>>> print(repr(result))
([['123', 'Main St'], {'house_number': ['123'], 'street_name': ['Main St']}])
>>> result.house_number
'123'
>>> result["street_name"]
'Main St'
>>> result.as_list()
['123', 'Main St']
>>> result.as_dict()
{'house_number': '123', 'street_name': 'Main St'}
>>> print(result.dump())
['123', 'Main St']
- house_number: '123'
- street_name: 'Main St'
```

2.2.9 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 as err:
    print(err.line)
    print(" " * (err.column - 1) + "^")
    print(err)
```

  `ParseExceptions` also have an `explain()` method that gives this same information:

```
except ParseException as err:
    print(err.explain())
```

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

```
bad_grammar = Forward()
good_token = Literal("A")
bad_grammar <<= Opt(good_token) + bad_grammar
```

- `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 `set_debug()`, or test the matching of expression fragments by testing them using `search_string()` or `scan_string()`.
- Use `with_line_numbers` from `pyparsing_testing` to display the input string being parsed, with line and column numbers that correspond to the values reported in set_debug() output:

```
import pyparsing as pp
ppt = pp.testing

data = """\
   A
      100"""

expr = pp.Word(pp.alphanums).set_name("word").set_debug()
print(ppt.with_line_numbers(data))
expr[...].parseString(data)
```

prints:

```
.          1
  1234567890
1:   A|
2:      100|

Match word at loc 3(1,4)
    A
    ^
Matched word -> ['A']
Match word at loc 11(2,7)
       100
       ^
Matched word -> ['100']
```

with_line_numbers has several options for displaying control characters, end-of-line and space markers, Unicode symbols for control characters - these are documented in the function's docstring.

- Diagnostics can be enabled using `pyparsing.enable_diag` and passing one of the following enum values defined in `pyparsing.Diagnostics`
  - `warn_multiple_tokens_in_named_alternation` - flag to enable warnings when a results name is defined on a *MatchFirst* or *Or* expression with one or more *And* subexpressions
  - `warn_ungrouped_named_tokens_in_collection` - flag to enable warnings when a results name is defined on a containing expression with ungrouped subexpressions that also have results names
  - `warn_name_set_on_empty_Forward` - flag to enable warnings when a `Forward` is defined with a results name, but has no contents defined
  - `warn_on_parse_using_empty_Forward` - flag to enable warnings when a `Forward` is defined in a grammar but has never had an expression attached to it
  - `warn_on_assignment_to_Forward` - flag to enable warnings when a `Forward` is defined but is overwritten by assigning using `'='` instead of `'<<='` or `'<<'`
  - `warn_on_multiple_string_args_to_oneof` - flag to enable warnings when `one_of` is incorrectly called with multiple str arguments
  - `enable_debug_on_named_expressions` - flag to auto-enable debug on all subsequent calls to `ParserElement.set_name`

  All warnings can be enabled by calling `pyparsing.enable_all_warnings()`. Sample:

```
import pyparsing as pp
pp.enable_all_warnings()

fwd = pp.Forward().set_results_name("recursive_expr")
```

(continues on next page)

(continued from previous page)

```
>>> UserWarning: warn_name_set_on_empty_Forward: setting results name 'recursive_
↪expr'
                on Forward expression that has no contained expression
```

Warnings can also be enabled using the Python `-W` switch (using `-Wd` or `-Wd::pyparsing`) or setting a non-empty value to the environment variable `PYPARSINGENABLEALLWARNINGS`. (If using `-Wd` for testing, but wishing to disable pyparsing warnings, add `-Wi::pyparsing`.)

2.3 1.3 Miscellaneous attributes and methods

2.3.1 1.3.1 Helper methods

- `counted_array(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).
- `one_of(choices, caseless=False, as_keyword=False)` - convenience function for quickly declaring an alternative set of *Literal* expressions. `choices` can be passed as a list of strings or as a single string of values separated by spaces. The values are sorted so that longer matches are attempted first; this ensures that a short value does not mask a longer one that starts with the same characters. If `caseless=True`, will create an alternative set of *CaselessLiteral* tokens. If `as_keyword=True`, `one_of` will declare *Keyword* expressions instead of *Literal* expressions.
- `dict_of(key, value)` - convenience function for quickly declaring a dictionary pattern of `Dict(ZeroOrMore(Group(key + value)))`.
- `make_html_tags(tag_str)` and `make_xml_tags(tag_str)` - 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 named results in the returned *ParseResults*. `make_html_tags` is less restrictive than `make_xml_tags`, especially with respect to case sensitivity.
- `infix_notation(base_operand, operator_list)` - convenience function to define a grammar for parsing infix notation expressions with a hierarchical precedence of operators. To use the `infix_notation` helper:

  1. Define the base "atom" operand term of the grammar. For this simple grammar, the smallest operand is either an integer or a variable. This will be the first argument to the `infix_notation` method.
  2. Define a list of tuples for each level of operator precedence. Each tuple is of the form `(operand_expr, num_operands, right_left_assoc, parse_action)`, where:
     - `operand_expr` - 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".
     - `num_operands` - the number of terms for this operator (must be 1, 2, or 3)
     - `right_left_assoc` is the indicator whether the operator is right or left associative, using the pyparsing-defined constants `OpAssoc.RIGHT` and `OpAssoc.LEFT`.
     - `parse_action` is the parse action to be associated with expressions matching this operator expression (the `parse_action` tuple member may be omitted)

3. Call `infix_notation` 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.

`infix_notation` also supports optional arguments `lpar` and `rpar`, to parse groups with symbols other than "(" and ")". They may be passed as strings (in which case they will be converted to `Suppress` objects, and suppressed from the parsed results), or passed as pyparsing expressions, in which case they will be kept as-is, and grouped with their contents.

For instance, to use "<" and ">" for grouping symbols, you could write:

```
expr = infix_notation(int_expr,
    [
        (one_of("+ -"), 2, opAssoc.LEFT),
    ],
    lpar="<",
    rpar=">"
    )
expr.parse_string("3 - <2 + 11>")
```

returning:

```
[3, '-', [2, '+', 11]]
```

If the grouping symbols are to be retained, then pass them as pyparsing `Literals`:

```
expr = infix_notation(int_expr,
    [
        (one_of("+ -"), 2, opAssoc.LEFT),
    ],
    lpar=Literal("<"),
    rpar=Literal(">")
    )
expr.parse_string("3 - <2 + 11>")
```

returning:

```
[3, '-', ['<', [2, '+', 11], '>']]
```

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

```
first = Word(nums)
match_expr = first + ":" + match_previous_literal(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)
match_expr = first + ":" + match_previous_expr(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".

- `nested_expr(opener, closer, content=None, ignore_expr=quoted_string)` - 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)
  - `ignore_expr` - expression for ignoring opening and closing delimiters (default=‘‘quoted_string‘‘)

  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 `ignore_expr` 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 `quoted_string` or a comment expression. Specify multiple expressions using an *Or* or *MatchFirst*. The default is `quoted_string`, but if no expressions are to be ignored, then pass `None` for this argument.

- `IndentedBlock(statement_expr, recursive=False, grouped=True)` - function to define an indented block of statements, similar to indentation-based blocking in Python source code:
  - `statement_expr` - the expression defining a statement that will be found in the indented block; a valid `IndentedBlock` must contain at least 1 matching `statement_expr`
  - `recursive` - flag indicating whether the IndentedBlock can itself contain nested sub-blocks of the same type of expression (default=False)
  - `grouped` - flag indicating whether the tokens returned from parsing the IndentedBlock should be grouped (default=True)
- `original_text_for(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:

```
full_name = 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 `original_text_for`:

```
full_name = original_text_for(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.
- `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(range_spec)` - 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 `range_spec` does not include support for generic regular expressions, just string range specs)
- `trace_parse_action(fn)` - decorator function to debug parse actions. Lists each call, called arguments, and return value or exception

2.3.2 1.3.2 Helper parse actions

- `remove_quotes` - removes the first and last characters of a quoted string; useful to remove the delimiting quotes from quoted strings

- `replace_with(repl_string)` - returns a parse action that simply returns the `repl_string`; useful when using `transform_string`, or converting HTML entities, as in:

```
nbsp = Literal(" ").set_parse_action(replace_with("<BLANK>"))
```

- `original_text_for`- restores any internal whitespace or suppressed text within the tokens for a matched parse expression. This is especially useful when defining expressions for `scan_string` or `transform_string` applications.
- `with_attribute(*args, **kwargs)` - helper to create a validating parse action to be used with start tags created with `make_xml_tags` or `make_html_tags`. Use `with_attribute` to qualify a starting tag with a required attribute value, to avoid false matches on common tags such as `<TD>` or `<DIV>`.

  `with_attribute` 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 `with_attribute.ANY_VALUE`, which will match any value - use this to ensure that an attribute is present but any attribute value is acceptable.
- `match_only_at_col(column_number)` - 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
- `common.convert_to_integer()` - converts all matched tokens to int
- `common.convert_to_float()` - converts all matched tokens to float
- `common.convert_to_date()` - converts matched token to a datetime.date
- `common.convert_to_datetime()` - converts matched token to a datetime.datetime
- `common.strip_html_tags()` - removes HTML tags from matched token
- `common.downcase_tokens()` - converts all matched tokens to lowercase
- `common.upcase_tokens()` - converts all matched tokens to uppercase

2.3.3 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
```

- `identchars` - a string containing characters that are valid as initial identifier characters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyzª
µºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

- `identbodychars` - a string containing characters that are valid as identifier body characters (those following a valid leading identifier character as given in *identchars*):

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyzª
µ·ºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

- `printables` - same as `string.printable`, minus the space (`' '`) character
- `empty` - a global `Empty()`; will always match
- `sgl_quoted_string` - a string of characters enclosed in 's; may include whitespace, but not newlines
- `dbl_quoted_string` - a string of characters enclosed in "s; may include whitespace, but not newlines
- `quoted_string` - `sgl_quoted_string | dbl_quoted_string`
- `python_quoted_string` - `quoted_string | multiline quoted string`
- `c_style_comment` - a comment block delimited by `'/*'` and `'*/'` sequences; can span multiple lines, but does not support nesting of comments
- `html_comment` - a comment block delimited by `'<!--'` and `'-->'` sequences; can span multiple lines, but does not support nesting of comments
- `comma_separated_list` - 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
- `rest_of_line` - all remaining printable characters up to but not including the next newline
- `common.integer` - an integer with no leading sign; parsed token is converted to int
- `common.hex_integer` - a hexadecimal integer; parsed token is converted to int
- `common.signed_integer` - an integer with optional leading sign; parsed token is converted to int
- `common.fraction` - signed_integer '/' signed_integer; parsed tokens are converted to float
- `common.mixed_integer` - signed_integer '-' fraction; parsed tokens are converted to float
- `common.real` - real number; parsed tokens are converted to float
- `common.sci_real` - real number with optional scientific notation; parsed tokens are convert to float
- `common.number` - any numeric expression; parsed tokens are returned as converted by the matched expression
- `common.fnumber` - any numeric expression; parsed tokens are converted to float
- `common.identifier` - a programming identifier (follows Python's syntax convention of leading alpha or "_", followed by 0 or more alpha, num, or "_")
- `common.ipv4_address` - IPv4 address
- `common.ipv6_address` - IPv6 address
- `common.mac_address` - MAC address (with ":", "-", or "." delimiters)
- `common.iso8601_date` - date in `YYYY-MM-DD` format
- `common.iso8601_datetime` - datetime in `YYYY-MM-DDThh:mm:ss.s(Z|+-00:00)` format; trailing seconds, milliseconds, and timezone optional; accepts separating `'T'` or `' '`
- `common.url` - matches URL strings and returns a ParseResults with named fields like those returned by `urllib.parse.urlparse()`

2.3.4 1.3.4 Unicode character sets for international parsing

Pyparsing includes the `unicode` namespace that contains definitions for `alphas`, `nums`, `alphanums`, `identchars`, `identbodychars`, and `printables` for character ranges besides 7- or 8-bit ASCII. You can access them using code like the following:

```
import pyparsing as pp
ppu = pp.unicode

greek_word = pp.Word(ppu.Greek.alphas)
greek_word[...].parse_string("Καλημρα κσμε")
```

The following language ranges are defined.

Unicode set	Alternate names	Description
Arabic		
Chinese		
CJK		Union of Chinese, Japanese, and Korean sets
Cyrillic		
Devanagari		
Greek	Ελληνικ	
Hangul	Korean,	
Hebrew		
Japanese		Union of Kanji, Katakana, and Hiragana sets
Japanese.Hiragana		
Japanese.Kanji		
Japanese.Katakana		
Latin1		All Unicode characters up to code point 255
LatinA		
LatinB		
Thai		
BasicMultilingualPlane	BMP	All Unicode characters up to code point 65535

The base `unicode` class also includes definitions based on all Unicode code points up to `sys.maxunicode`. This set will include emojis, wingdings, and many other specialized and typographical variant characters.

2.4 1.4 Generating Railroad Diagrams

Grammars are conventionally represented in what are called "railroad diagrams", which allow you to visually follow the sequence of tokens in a grammar along lines which are a bit like train tracks. You might want to generate a railroad diagram for your grammar in order to better understand it yourself, or maybe to communicate it to others.

2.4.1 1.4.1 Usage

To generate a railroad diagram in pyparsing, you first have to install pyparsing with the `diagrams` extra. To do this, just run `pip install pyparsing[diagrams]`, and make sure you add `pyparsing[diagrams]` to any `setup.py` or `requirements.txt` that specifies pyparsing as a dependency.

Create your parser as you normally would. Then call `create_diagram()`, passing the name of an output HTML file.:

```
street_address = Word(nums).set_name("house_number") + Word(alphas)[1, ...].set_name(
↪"street_name")
street_address.set_name("street_address")
street_address.create_diagram("street_address_diagram.html")
```

This will result in the railroad diagram being written to `street_address_diagram.html`.

create_diagrams takes the following arguments:

- `output_html` (str or file-like object) - output target for generated diagram HTML
- `vertical` (int) - threshold for formatting multiple alternatives vertically instead of horizontally (default=3)
- `show_results_names` - bool flag whether diagram should show annotations for defined results names
- `show_groups` - bool flag whether groups should be highlighted with an unlabeled surrounding box
- `embed` - bool flag whether generated HTML should omit <HEAD>, <BODY>, and <DOCTYPE> tags to embed the resulting HTML in an enclosing HTML source (such as PyScript HTML)
- `head` - str containing additional HTML to insert into the <HEAD> section of the generated code; can be used to insert custom CSS styling
- `body` - str containing additional HTML to insert at the beginning of the <BODY> section of the generated code

2.4.2 1.4.2 Example

You can view an example railroad diagram generated from a pyparsing grammar for SQL SELECT statements (generated from examples/select_parser.py).

2.4.3 1.4.3 Naming tip

Parser elements that are separately named will be broken out as their own sub-diagrams. As a short-cut alternative to going through and adding `.set_name()` calls on all your sub-expressions, you can use `autoname_elements()` after defining your complete grammar. For example:

```
a = pp.Literal("a")
b = pp.Literal("b").set_name("bbb")
pp.autoname_elements()
```

a will get named "a", while *b* will keep its name "bbb".

2.4.4 1.4.4 Customization

You can customize the resulting diagram in a few ways. To do so, run `pyparsing.diagrams.to_railroad` to convert your grammar into a form understood by the railroad-diagrams module, and then `pyparsing.diagrams.railroad_to_html` to convert that into an HTML document. For example:

```
from pyparsing.diagram import to_railroad, railroad_to_html

with open('output.html', 'w') as fp:
    railroad = to_railroad(my_grammar)
    fp.write(railroad_to_html(railroad))
```

This will result in the railroad diagram being written to `output.html`

You can then pass in additional keyword arguments to `pyparsing.diagrams.to_railroad`, which will be passed into the `Diagram()` constructor of the underlying library, as explained here.

In addition, you can edit global options in the underlying library, by editing constants:

```
from pyparsing.diagram import to_railroad, railroad_to_html
import railroad

railroad.DIAGRAM_CLASS = "my-custom-class"
my_railroad = to_railroad(my_grammar)
```

These options are documented here.

Finally, you can edit the HTML produced by `pyparsing.diagrams.railroad_to_html` by passing in certain keyword arguments that will be used in the HTML template. Currently, these are:

- `head`: A string containing HTML to use in the `<head>` tag. This might be a stylesheet or other metadata
- `body`: A string containing HTML to use in the `<body>` tag, above the actual diagram. This might consist of a heading, description, or JavaScript.

If you want to provide a custom stylesheet using the `head` keyword, you can make use of the following CSS classes:

- `railroad-group`: A group containing everything relating to a given element group (ie something with a heading)
- `railroad-heading`: The title for each group
- `railroad-svg`: A div containing only the diagram SVG for each group
- `railroad-description`: A div containing the group description (unused)

CHAPTER 3

pyparsing

3.1 pyparsing module

3.1.1 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.parse_string(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.parse_string* 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.set_results_name*
- access the parsed data, which is returned as a *ParseResults* object
- find some helpful expression short-cuts like *DelimitedList* and *one_of*
- find more useful common expressions in the *pyparsing_common* namespace class

class pyparsing.**__compat__**
Bases: pyparsing.util.__config_flags

A cross-version compatibility configuration for pyparsing features that will be released in a future version. By setting values in this configuration to True, those features can be enabled in prior versions for compatibility development and testing.

- collect_all_And_tokens - flag to enable fix for Issue #63 that fixes erroneous grouping of results names when an *And* expression is nested within an *Or* or *MatchFirst*; maintained for compatibility, but setting to False no longer restores pre-2.3.1 behavior

class pyparsing.**__diag__**
Bases: pyparsing.util.__config_flags

class pyparsing.**And**(*exprs_arg: Iterable[pyparsing.core.ParserElement], savelist: bool = True*)
Bases: pyparsing.core.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 = Word(alphas)[1, ...]

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

__init__(*exprs_arg: Iterable[pyparsing.core.ParserElement], savelist: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**AtLineStart**(*expr: Union[pyparsing.core.ParserElement, str]*)
Bases: pyparsing.core.ParseElementEnhance

Matches if an expression matches 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 (AtLineStart('AAA') + rest_of_line).search_string(test):
    print(t)
```

prints:

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

__init__(*expr: Union[pyparsing.core.ParserElement, str]*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**AtStringStart**(*expr: Union[pyparsing.core.ParserElement, str]*)
Bases: pyparsing.core.ParseElementEnhance

Matches if expression matches at the beginning of the parse string:

```
AtStringStart(Word(nums)).parse_string("123")
# prints ["123"]

AtStringStart(Word(nums)).parse_string("    123")
# raises ParseException
```

__init__(*expr: Union[pyparsing.core.ParserElement, str]*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**CaselessKeyword**(*match_string: str = ''*, *ident_chars: Optional[str] = None*, ***, *matchString: str = ''*, *identChars: Optional[str] = None*)
Bases: pyparsing.core.Keyword

Caseless version of *Keyword*.

Example:

```
CaselessKeyword("CMD")[1, ...].parse_string("cmd CMD Cmd10")
# -> ['CMD', 'CMD']
```

(Contrast with example for *CaselessLiteral*.)

__init__(*match_string: str = ''*, *ident_chars: Optional[str] = None*, ***, *matchString: str = ''*, *identChars: Optional[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**CaselessLiteral**(*match_string: str = ''*, ***, *matchString: str = ''*)
Bases: pyparsing.core.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:

```
CaselessLiteral("CMD")[1, ...].parse_string("cmd CMD Cmd10")
# -> ['CMD', 'CMD', 'CMD']
```

(Contrast with example for *CaselessKeyword*.)

__init__(*match_string: str = ''*, *, matchString: str = ''*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**CharsNotIn**(*not_chars: str = ''*, *min: int = 1*, *max: int = 0*, *exact: int = 0*, *, *notChars: str = ''*)
Bases: pyparsing.core.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).parse_string("dkls,lsdkjf,s12 34,@!#,213"))
```

prints:

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

__init__(*not_chars: str = ''*, *min: int = 1*, *max: int = 0*, *exact: int = 0*, *, *notChars: str = ''*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**CloseMatch**(*match_string: str*, *max_mismatches: Optional[int] = None*, *, *maxMismatches: int = 1*, *caseless=False*)
Bases: pyparsing.core.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
- `caseless` - a boolean indicating whether to ignore casing when comparing characters
- `max_mismatches` - (`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.parse_string("ATCATCGAAXGGA") # -> (['ATCATCGAAXGGA'], {'mismatches': [[9]],
↪'original': ['ATCATCGAATGGA']})
patt.parse_string("ATCAXCGAAXGGA") # -> Exception: Expected 'ATCATCGAATGGA' (with␣
↪up to 1 mismatches) (at char 0), (line:1, col:1)

# exact match
```

(continues on next page)

(continued from previous page)

```
patt.parse_string("ATCATCGAATGGA") # -> (['ATCATCGAATGGA'], {'mismatches': [[]],
↪'original': ['ATCATCGAATGGA']})

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

__init__(*match_string: str*, *max_mismatches: Optional[int] = None*, ***, *maxMismatches: int = 1*, *caseless=False*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Combine**(*expr: pyparsing.core.ParserElement*, *join_string: str = ''*, *adjacent: bool = True*, ***, *joinString: Optional[str] = None*)
Bases: pyparsing.core.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.parse_string('3.1416')) # -> ['3', '.', '1416']
# will also erroneously match the following
print(real.parse_string('3. 1416')) # -> ['3', '.', '1416']

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

__init__(*expr: pyparsing.core.ParserElement*, *join_string: str = ''*, *adjacent: bool = True*, ***, *joinString: Optional[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

ignore(*other*) → pyparsing.core.ParserElement
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 = Word(alphas)[1, ...]
patt.parse_string('ablaj /* comment */ lskjd')
# -> ['ablaj']

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

class pyparsing.**DelimitedList**(*expr: Union[str, pyparsing.core.ParserElement]*, *delim: Union[str, pyparsing.core.ParserElement] = ','*, *combine: bool = False*, *min: Optional[int] = None*, *max: Optional[int] = None*, ***, *allow_trailing_delim: bool = False*)
Bases: pyparsing.core.ParseElementEnhance

__init__(*expr: Union[str, pyparsing.core.ParserElement]*, *delim: Union[str, pyparsing.core.ParserElement] = ','*, *combine: bool = False*, *min: Optional[int] = None*, *max: Optional[int] = None*, ***, *allow_trailing_delim: bool = 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.

If `allow_trailing_delim` is set to True, then the list may end with a delimiter.

Example:

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

class pyparsing.**Dict**(*expr: pyparsing.core.ParserElement*, *asdict: bool = False*)

Bases: pyparsing.core.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.

The optional `asdict` argument when set to True will return the parsed tokens as a Python dict instead of a pyparsing ParseResults.

Example:

```
data_word = Word(alphas)
label = data_word + FollowedBy(':')

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

# print attributes as plain groups
print(attr_expr[1, ...].parse_string(text).dump())

# instead of OneOrMore(expr), parse using Dict(Group(expr)[1, ...]) - Dict will␣
↪auto-assign names
result = Dict(Group(attr_expr)[1, ...]).parse_string(text)
print(result.dump())

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

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.

__init__(*expr: pyparsing.core.ParserElement*, *asdict: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Each**(*exprs: Iterable[pyparsing.core.ParserElement]*, *savelist: bool = True*)
Bases: pyparsing.core.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 = one_of("RED ORANGE YELLOW GREEN BLUE PURPLE BLACK WHITE BROWN")
shape_type = one_of("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 & Opt(color_attr) & Opt(size_attr)

shape_spec.run_tests('''
    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
```

(continues on next page)

(continued from previous page)

```
  - y: 40
- shape: TRIANGLE
- size: 20
```

__init__(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Empty**(*match_string=''*, *, *matchString=''*)
Bases: pyparsing.core.Literal

An empty token, will always match.

__init__(*match_string=''*, *, *matchString=''*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**FollowedBy**(*expr: Union[pyparsing.core.ParserElement, str]*)
Bases: pyparsing.core.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, stop_on=label).set_
↪parse_action(' '.join))

attr_expr[1, ...].parse_string("shape: SQUARE color: BLACK posn: upper left").
↪pprint()
```

prints:

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

__init__(*expr: Union[pyparsing.core.ParserElement, str]*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Forward**(*other: Union[pyparsing.core.ParserElement, str, None] = None*)
Bases: pyparsing.core.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:

```
fwd_expr << a | b | c
```

will actually be evaluated as:

```
(fwd_expr << 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:

```
fwd_expr << (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.

__init__(*other: Union[pyparsing.core.ParserElement, str, None] = None*)
Initialize self. See help(type(self)) for accurate signature.

__or__(*other*) → pyparsing.core.ParserElement
Implementation of | operator - returns *MatchFirst*

copy() → pyparsing.core.ParserElement
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).set_parse_action(lambda toks: int(toks[0]))
integerK = integer.copy().add_parse_action(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().add_parse_action(lambda toks: toks[0] * 1024 *␣
↪1024) + Suppress("M")

print((integerK | integerM | integer)[1, ...].parse_string("5K 100 640K 256M
↪"))
```

prints:

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

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

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

ignoreWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *ignore_whitespace*

ignore_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Enables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern.

Parameters **recursive** – If True (the default), also enable whitespace skipping in child elements (if any)

leaveWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *leave_whitespace*

leave_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
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.

Parameters **recursive** – If true (the default), also disable whitespace skipping in child elements (if any)

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

class pyparsing.**GoToColumn**(*colno: int*)
Bases: pyparsing.core.PositionToken

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

__init__(*colno: int*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Group**(*expr: pyparsing.core.ParserElement*, *aslist: bool = False*)
Bases: pyparsing.core.TokenConverter

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

The optional `aslist` argument when set to True will return the parsed tokens as a Python list instead of a pyparsing ParseResults.

Example:

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

func = ident + Group(Opt(DelimitedList(term)))
print(func.parse_string("fn a, b, 100"))
# -> ['fn', ['a', 'b', '100']]
```

__init__(*expr: pyparsing.core.ParserElement*, *aslist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**IndentedBlock**(*expr: pyparsing.core.ParserElement*, ***, *recursive: bool = False*, *grouped: bool = True*)
Bases: pyparsing.core.ParseElementEnhance

Expression to match one or more expressions at a given indentation level. Useful for parsing text where structure is implied by indentation (like Python source code).

__init__(*expr: pyparsing.core.ParserElement*, ***, *recursive: bool = False*, *grouped: bool = True*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Keyword**(*match_string: str = ''*, *ident_chars: Optional[str] = None*, *caseless: bool = False*, ***, *matchString: str = ''*, *identChars: Optional[str] = None*)
Bases: pyparsing.core.Token

Token to exactly match a specified string as a keyword, that is, it must be immediately preceded and followed by whitespace or non-keyword characters. 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:

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

For case-insensitive matching, use *CaselessKeyword*.

__init__(*match_string: str = ''*, *ident_chars: Optional[str] = None*, *caseless: bool = False*, ***, *matchString: str = ''*, *identChars: Optional[str] = None*)
Initialize self. See help(type(self)) for accurate signature.

static setDefaultKeywordChars(*chars*) → None
Overrides the default characters used by *Keyword* expressions.

static set_default_keyword_chars(*chars*) → None
Overrides the default characters used by *Keyword* expressions.

class pyparsing.**LineEnd**
Bases: pyparsing.core.PositionToken

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

__init__()
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**LineStart**
Bases: pyparsing.core.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' + rest_of_line).search_string(test):
    print(t)
```

prints:

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

__init__()
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Literal**(*match_string: str = ''*, ***, *matchString: str = ''*)
Bases: pyparsing.core.Token

Token to exactly match a specified string.

Example:

```
Literal('blah').parse_string('blah')  # -> ['blah']
Literal('blah').parse_string('blahfooblah')  # -> ['blah']
Literal('blah').parse_string('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*.

__init__(*match_string: str = ''*, *, *matchString: str = ''*)
Initialize self. See help(type(self)) for accurate signature.

static __new__(*cls*, *match_string: str = ''*, *, *matchString: str = ''*)
Create and return a new object. See help(type) for accurate signature.

class pyparsing.**Located**(*expr: Union[pyparsing.core.ParserElement, str]*, *savelist: bool = False*)
Bases: pyparsing.core.ParseElementEnhance

Decorates 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.parse_with_tabs*

Example:

```
wd = Word(alphas)
for match in Located(wd).search_string("ljsdf123lksdjjf123lkkjj1222"):
    print(match)
```

prints:

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

class pyparsing.**PrecededBy**(*expr: Union[pyparsing.core.ParserElement, str]*, *retreat: Optional[int] = None*)
Bases: pyparsing.core.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
```

__init__(*expr: Union[pyparsing.core.ParserElement, str], retreat: Optional[int] = None*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**MatchFirst**(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Bases: pyparsing.core.ParseExpression

Requires that at least one *ParseExpression* is found. If more than one expression matches, 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.search_string("123 3.1416 789")) #  Fail! -> [['123'], ['3'], ['1416
↪'], ['789']]

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

__init__(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

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

A token that will never match.

__init__()
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**NotAny**(*expr: Union[pyparsing.core.ParserElement, str]*)
Bases: pyparsing.core.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 = Opt(NOT) + ident

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

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

__init__(*expr: Union[pyparsing.core.ParserElement, str]*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**OneOrMore**(*expr: Union[str, pyparsing.core.ParserElement], stop_on: Union[pyparsing.core.ParserElement, str, None] = None, *, stopOn: Union[pyparsing.core.ParserElement, str, None] = None*)

Bases: pyparsing.core._MultipleMatch

Repetition of one or more of the given expression.

Parameters:

- expr - expression that must match one or more times
- stop_on - (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).set_parse_action('
↪'.join))

text = "shape: SQUARE posn: upper left color: BLACK"
attr_expr[1, ...].parse_string(text).pprint()  # Fail! read 'color' as data␣
↪instead of next label -> [['shape', 'SQUARE color']]

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

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

class pyparsing.**OnlyOnce**(*method_call*)

Bases: object

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

__call__(*s*, *l*, *t*)
Call self as a function.

__init__(*method_call*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

reset()
Allow the associated parse action to be called once more.

class pyparsing.**OpAssoc**

Bases: enum.Enum

Enumeration of operator associativity - used in constructing InfixNotationOperatorSpec for *infix_notation*

class pyparsing.**Opt**(*expr: Union[pyparsing.core.ParserElement, str], default: Any = <pyparsing.core._NullToken object>*)

Bases: pyparsing.core.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) + Opt('-' + Word(nums, exact=4)))
zip.run_tests('''
    # 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)
```

__init__(*expr: Union[pyparsing.core.ParserElement, str], default: Any = <pyparsing.core._NullToken object>*)
Initialize self. See help(type(self)) for accurate signature.

pyparsing.**Optional**
alias of `pyparsing.core.Opt`

class pyparsing.**Or**(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Bases: `pyparsing.core.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.search_string("123 3.1416 789"))
```

prints:

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

__init__(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

exception pyparsing.**ParseBaseException**(*pstr: str, loc: int = 0, msg: Optional[str] = None, elem=None*)
Bases: Exception

base exception class for all parsing runtime exceptions

__init__(*pstr: str, loc: int = 0, msg: Optional[str] = None, elem=None*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__() → str
Return str(self).

col
Return the 1-based column on the line of text where the exception occurred.

column
Return the 1-based column on the line of text where the exception occurred.

explain(*depth=16*) → str
Method to translate the Python internal traceback into a list of the pyparsing expressions that caused the exception to be raised.

Parameters:

- 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.

Example:

```
expr = pp.Word(pp.nums) * 3
try:
    expr.parse_string("123 456 A789")
except pp.ParseException as pe:
    print(pe.explain(depth=0))
```

prints:

```
123 456 A789
        ^
ParseException: Expected W:(0-9), found 'A'  (at char 8), (line:1, col:9)
```

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

Note: pyparsing's default truncation of exception tracebacks may also truncate the stack of expressions that are displayed in the `explain` output. To get the full listing of parser expressions, you may have to set `ParserElement.verbose_stacktrace = True`

static explain_exception(*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.

line
Return the line of text where the exception occurred.

lineno
Return the 1-based line number of text where the exception occurred.

markInputline(*marker_string: Optional[str] = None*, *, *markerString: str = '>!<'*) → str
Deprecated - use *mark_input_line*

mark_input_line(*marker_string: Optional[str] = None*, *, *markerString: str = '>!<'*) → str
Extracts the exception line from the input string, and marks the location of the exception with a special symbol.

class pyparsing.**ParseElementEnhance**(*expr: Union[pyparsing.core.ParserElement, str], savelist: bool = False*)
Bases: pyparsing.core.ParserElement

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

__init__(*expr: Union[pyparsing.core.ParserElement, str], savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

ignore(*other*) → pyparsing.core.ParserElement
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 = Word(alphas)[1, ...]
patt.parse_string('ablaj /* comment */ lskjd')
# -> ['ablaj']

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

ignoreWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *ignore_whitespace*

ignore_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Enables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern.

Parameters **recursive** – If True (the default), also enable whitespace skipping in child elements (if any)

leaveWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *leave_whitespace*

leave_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
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.

Parameters **recursive** – If true (the default), also disable whitespace skipping in child elements (if any)

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

exception pyparsing.**ParseException**(*pstr: str, loc: int = 0, msg: Optional[str] = None, elem=None*)
Bases: pyparsing.exceptions.ParseBaseException

Exception thrown when a parse expression doesn't match the input string

Example:

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

prints:

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

__weakref__
list of weak references to the object (if defined)

class pyparsing.**ParseExpression**(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Bases: pyparsing.core.ParserElement

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

__init__(*exprs: Iterable[pyparsing.core.ParserElement], savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

copy() → pyparsing.core.ParserElement
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).set_parse_action(lambda toks: int(toks[0]))
integerK = integer.copy().add_parse_action(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
integerM = integer.copy().add_parse_action(lambda toks: toks[0] * 1024 *␣
↪1024) + Suppress("M")

print((integerK | integerM | integer)[1, ...].parse_string("5K 100 640K 256M
↪"))
```

prints:

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

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

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

ignore(*other*) → pyparsing.core.ParserElement
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 = Word(alphas)[1, ...]
patt.parse_string('ablaj /* comment */ lskjd')
# -> ['ablaj']

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

ignoreWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *ignore_whitespace*

ignore_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement

Extends ignore_whitespace defined in base class, and also invokes leave_whitespace on all contained expressions.

leaveWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *leave_whitespace*

leave_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement

Extends leave_whitespace defined in base class, and also invokes leave_whitespace on all contained expressions.

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

exception pyparsing.**ParseFatalException**(*pstr: str*, *loc: int = 0*, *msg: Optional[str] = None*, *elem=None*)
Bases: pyparsing.exceptions.ParseBaseException

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

__weakref__
list of weak references to the object (if defined)

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.<results_name> - see *ParserElement.set_results_name*)

Example:

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

(continues on next page)

(continued from previous page)

```
#             + integer("day"))

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

def test(s, fn=repr):
    print(f"{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'
```

class List

Bases: `list`

Simple wrapper class to distinguish parsed list results that should be preserved as actual Python lists, instead of being converted to *ParseResults*:

```
LBRACK, RBRACK = map(pp.Suppress, "[]")
element = pp.Forward()
item = ppc.integer
element_list = LBRACK + pp.DelimitedList(element) + RBRACK

# add parse actions to convert from ParseResults to actual Python collection␣
↪types
def as_python_list(t):
    return pp.ParseResults.List(t.as_list())
element_list.add_parse_action(as_python_list)

element <<= item | element_list

element.run_tests('''
    100
    [2,3,4]
    [[2, 1],3,4]
    [(2, 1),3,4]
    (2,3,4)
    ''', post_parse=lambda s, r: (r[0], type(r[0])))
```

prints:

```
100
(100, <class 'int'>)

[2,3,4]
([2, 3, 4], <class 'list'>)

[[2, 1],3,4]
([[2, 1], 3, 4], <class 'list'>)
```

(Used internally by *Group* when *aslist=True*.)

static __new__(*cls*, *contained=None*)
Create and return a new object. See help(type) for accurate signature.

__weakref__
list of weak references to the object (if defined)

__dir__()
Default dir() implementation.

__init__(*toklist=None*, *name=None*, *asList=True*, *modal=True*, *isinstance=<built-in function isinstance>*)
Initialize self. See help(type(self)) for accurate signature.

static __new__(*cls*, *toklist=None*, *name=None*, ***kwargs*)
Create and return a new object. See help(type) for accurate signature.

__repr__() → str
Return repr(self).

__str__() → str
Return str(self).

append(*item*)
Add single element to end of `ParseResults` list of elements.

Example:

```
numlist = Word(nums)[...]
print(numlist.parse_string("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)))
numlist.add_parse_action(append_sum)
print(numlist.parse_string("0 123 321")) # -> ['0', '123', '321', 444]
```

asDict() → dict
Deprecated - use *as_dict*

asList() → list
Deprecated - use *as_list*

as_dict() → dict
Returns the named parse results as a nested dictionary.

Example:

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

(continues on next page)

(continued from previous page)

```
result = date_str.parse_string('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.as_dict()
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.as_dict())) # -> {"month": "31", "day": "1999", "year
↪": "12"}
```

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

Example:

```
patt = Word(alphas)[1, ...]
result = patt.parse_string("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 as_list() to create an actual list
result_list = result.as_list()
print(type(result_list), result_list) # -> <class 'list'> ['sldkj', 'lsdkj',
↪'sldkj']
```

clear()
Clear all elements and results names.

copy() → pyparsing.results.ParseResults
Returns a new shallow copy of a *ParseResults* object. *ParseResults* items contained within the source are shared with the copy. Use *ParseResults.deepcopy()* to create a copy with its own separate content values.

deepcopy() → pyparsing.results.ParseResults
Returns a new deep copy of a *ParseResults* object.

dump(*indent=''*, *full=True*, *include_list=True*, *_depth=0*) → str
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.parse_string('1999/12/31')
print(result.dump())
```

prints:

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

extend(*itemseq*)
Add sequence of elements to end of `ParseResults` list of elements.

Example:

```
patt = Word(alphas)[1, ...]

# 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)
patt.add_parse_action(make_palindrome)
print(patt.parse_string("lskdj sdlkjf lksd")) # ->
↪'lskdjsdlkjflksddsklfjkldsjdksl'
```

classmethod from_dict(*other*, *name=None*) → pyparsing.results.ParseResults
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*, *default_value=None*)
Returns named result matching the given key, or if there is no such name, then returns the given `default_value` or `None` if no `default_value` is specified.

Similar to `dict.get()`.

Example:

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

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

getName()
Deprecated - use *get_name*

get_name()
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 = user_data[1, ...]
```

(continues on next page)

(continued from previous page)

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

prints:

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

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

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

Similar to `list.insert()`.

Example:

```
numlist = Word(nums)[...]
print(numlist.parse_string("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)
numlist.add_parse_action(insert_locn)
print(numlist.parse_string("0 123 321")) # -> [0, '0', '123', '321']
```

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:

```
numlist = Word(nums)[...]
print(numlist.parse_string("0 123 321")) # -> ['0', '123', '321']

def remove_first(tokens):
    tokens.pop(0)
numlist.add_parse_action(remove_first)
print(numlist.parse_string("0 123 321")) # -> ['123', '321']

label = Word(alphas)
patt = label("LABEL") + Word(nums)[1, ...]
print(patt.parse_string("AAB 123 321").dump())

# 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):
```

(continues on next page)

(continued from previous page)

```
    tokens.pop("LABEL")
    return tokens
patt.add_parse_action(remove_LABEL)
print(patt.parse_string("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.parse_string("fna a,b,(fnb c,d,200),100")
result.pprint(width=40)
```

prints:

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

exception pyparsing.**ParseSyntaxException**(*pstr: str*, *loc: int = 0*, *msg: Optional[str] = None*, *elem=None*)

Bases: pyparsing.exceptions.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: bool = False*)

Bases: abc.ABC

Abstract base level parser element class.

class DebugActions(*debug_try*, *debug_match*, *debug_fail*)

Bases: tuple

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *debug_try: Optional[Callable[[str, int, ParserElement, bool], None]]*, *debug_match: Optional[Callable[[str, int, int, ParserElement, pyparsing.results.ParseResults, bool], None]]*, *debug_fail: Optional[Callable[[str, int, ParserElement, Exception, bool], None]]*)

Create new instance of DebugActions(debug_try, debug_match, debug_fail)

__repr__()

Return a nicely formatted representation string

debug_fail
Alias for field number 2

debug_match
Alias for field number 1

debug_try
Alias for field number 0

__add__(*other*) → pyparsing.core.ParserElement
Implementation of + operator - returns *And*. Adding strings to a *ParserElement* converts them to *Literal*s by default.

Example:

```
greet = Word(alphas) + "," + Word(alphas) + "!"
hello = "Hello, World!"
print(hello, "->", greet.parse_string(hello))
```

prints:

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

`...` may be used as a parse expression as a short form of *SkipTo*:

```
Literal('start') + ... + Literal('end')
```

is equivalent to:

```
Literal('start') + SkipTo('end')("_skipped*") + Literal('end')
```

Note that the skipped text is returned with '_skipped' as a results name, and to support having multiple skips in the same parser, the value returned is a list of all skipped text.

__and__(*other*) → pyparsing.core.ParserElement
Implementation of `&` operator - returns *Each*

__call__(*name: Optional[str] = None*) → pyparsing.core.ParserElement
Shortcut for *set_results_name*, with `list_all_matches=False`.

If `name` is given with a trailing `'*'` character, then `list_all_matches` will be passed as `True`.

If `name` is omitted, same as calling *copy*.

Example:

```
# these are equivalent
userdata = Word(alphas).set_results_name("name") + Word(nums + "-").set_
↪results_name("socsecno")
userdata = Word(alphas)("name") + Word(nums + "-")("socsecno")
```

__eq__(*other*)
Return self==value.

__getitem__(*key*)
use `[]` indexing notation as a short form for expression repetition:

- `expr[n]` is equivalent to `expr*n`
- `expr[m, n]` is equivalent to `expr*(m, n)`

- **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)`

`None` may be used in place of `...`.

Note that `expr[..., n]` and `expr[m, n]` do not raise an exception if more than `n` `expr`s exist in the input stream. If this behavior is desired, then write `expr[..., n] + ~expr`.

For repetition with a stop_on expression, use slice notation:

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

__hash__()
Return hash(self).

__init__(*savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

__invert__() → pyparsing.core.ParserElement
Implementation of ~ operator - returns *NotAny*

__mul__(*other*) → pyparsing.core.ParserElement
Implementation of * operator, allows use of `expr * 3` in place of `expr + expr + expr`. Expressions may also be multiplied by a 2-integer tuple, similar to `{min, max}` multipliers in regular expressions. Tuples may also include `None` as in:

- `expr*(n, None)` or `expr*(n, )` is equivalent to `expr*n + ZeroOrMore(expr)` (read as "at least n instances of `expr`")
- `expr*(None, n)` is equivalent to `expr*(0, n)` (read as "0 to n instances of `expr`")
- `expr*(None, None)` is equivalent to `ZeroOrMore(expr)`
- `expr*(1, None)` is equivalent to `OneOrMore(expr)`

Note that `expr*(None, n)` does not raise an exception if more than n exprs exist in the input stream; that is, `expr*(None, n)` does not enforce a maximum number of expr occurrences. If this behavior is desired, then write `expr*(None, n) + ~expr`

__or__(*other*) → pyparsing.core.ParserElement
Implementation of | operator - returns *MatchFirst*

__radd__(*other*) → pyparsing.core.ParserElement
Implementation of + operator when left operand is not a *ParserElement*

__rand__(*other*) → pyparsing.core.ParserElement
Implementation of & operator when left operand is not a *ParserElement*

__repr__() → str
Return repr(self).

__ror__(*other*) → pyparsing.core.ParserElement
Implementation of | operator when left operand is not a *ParserElement*

__rsub__(*other*) → pyparsing.core.ParserElement
Implementation of − operator when left operand is not a *ParserElement*

__rxor__(*other*) → pyparsing.core.ParserElement
Implementation of ^ operator when left operand is not a *ParserElement*

__str__() → str
Return str(self).

__sub__(*other*) → pyparsing.core.ParserElement
Implementation of − operator, returns *And* with error stop

__weakref__
list of weak references to the object (if defined)

__xor__(*other*) → pyparsing.core.ParserElement
Implementation of ^ operator - returns *Or*

addCondition(**fns*, ***kwargs*) → pyparsing.core.ParserElement
Deprecated - use *add_condition*

addParseAction(**fns*, ***kwargs*) → pyparsing.core.ParserElement
Deprecated - use *add_parse_action*

add_condition(**fns*, ***kwargs*) → pyparsing.core.ParserElement
Add a boolean predicate function to expression's list of parse actions. See *set_parse_action* for function call signatures. Unlike `set_parse_action`, functions passed to `add_condition` 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
- `call_during_try` = boolean to indicate if this method should be called during internal tryParse calls, default=False

Example:

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

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

add_parse_action(**fns*, ***kwargs*) → pyparsing.core.ParserElement
Add one or more parse actions to expression's list of parse actions. See *set_parse_action*.

See examples in *copy*.

copy() → pyparsing.core.ParserElement
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).set_parse_action(lambda toks: int(toks[0]))
integerK = integer.copy().add_parse_action(lambda toks: toks[0] * 1024) +␣
↪Suppress("K")
```

(continues on next page)

(continued from previous page)

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

print((integerK | integerM | integer)[1, ...].parse_string("5K 100 640K 256M
↪"))
```

prints:

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

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

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

create_diagram(*output_html: Union[TextIO, pathlib.Path, str]*, *vertical: int = 3*, *show_results_names: bool = False*, *show_groups: bool = False*, *embed: bool = False*, ***kwargs*) → None

Create a railroad diagram for the parser.

Parameters:

- `output_html` (str or file-like object) - output target for generated diagram HTML
- `vertical` (int) - threshold for formatting multiple alternatives vertically instead of horizontally (default=3)
- `show_results_names` - bool flag whether diagram should show annotations for defined results names
- `show_groups` - bool flag whether groups should be highlighted with an unlabeled surrounding box
- `embed` - bool flag whether generated HTML should omit <HEAD>, <BODY>, and <DOCTYPE> tags to embed the resulting HTML in an enclosing HTML source
- `head` - str containing additional HTML to insert into the <HEAD> section of the generated code; can be used to insert custom CSS styling
- `body` - str containing additional HTML to insert at the beginning of the <BODY> section of the generated code

Additional diagram-formatting keyword arguments can also be included; see railroad.Diagram class.

static disable_memoization() → None

Disables active Packrat or Left Recursion parsing and their memoization

This method also works if neither Packrat nor Left Recursion are enabled. This makes it safe to call before activating Packrat nor Left Recursion to clear any previous settings.

static enableLeftRecursion(*cache_size_limit: Optional[int] = None*, ***, *force=False*) → None

Deprecated - use *enable_left_recursion*

static enablePackrat(*cache_size_limit: Optional[int] = 128*, ***, *force: bool = False*) → None

Deprecated - use *enable_packrat*

static enable_left_recursion(*cache_size_limit: Optional[int] = None*, ***, *force=False*) → None

Enables "bounded recursion" parsing, which allows for both direct and indirect left-recursion. During parsing, left-recursive *Forward* elements are repeatedly matched with a fixed recursion depth that is gradually increased until finding the longest match.

Example:

```
import pyparsing as pp
pp.ParserElement.enable_left_recursion()

E = pp.Forward("E")
num = pp.Word(pp.nums)
# match `num`, or `num '+' num`, or `num '+' num '+' num`, ...
E <<= E + '+' - num | num

print(E.parse_string("1+2+3"))
```

Recursion search naturally memoizes matches of `Forward` elements and may thus skip reevaluation of parse actions during backtracking. This may break programs with parse actions which rely on strict ordering of side-effects.

Parameters:

- `cache_size_limit` - (default=‘‘None‘‘) - memoize at most this many `Forward` elements during matching; if `None` (the default), memoize all `Forward` elements.

Bounded Recursion parsing works similar but not identical to Packrat parsing, thus the two cannot be used together. Use `force=True` to disable any previous, conflicting settings.

static enable_packrat(*cache_size_limit: Optional[int] = 128*, *, *force: bool = False*) → None

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.enable_packrat*. For best results, call `enable_packrat()` immediately after importing pyparsing.

Example:

```
import pyparsing
pyparsing.ParserElement.enable_packrat()
```

Packrat parsing works similar but not identical to Bounded Recursion parsing, thus the two cannot be used together. Use `force=True` to disable any previous, conflicting settings.

ignore(*other: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

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 = Word(alphas)[1, ...]
patt.parse_string('ablaj /* comment */ lskjd')
# -> ['ablaj']

patt.ignore(c_style_comment)
```

(continues on next page)

(continued from previous page)

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

ignoreWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *ignore_whitespace*

ignore_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Enables the skipping of whitespace before matching the characters in the *ParserElement*'s defined pattern.

Parameters **recursive** – If `True` (the default), also enable whitespace skipping in child elements (if any)

static inlineLiteralsUsing(*cls: type*) → None
Deprecated - use *inline_literals_using*

static inline_literals_using(*cls: type*) → None
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.parse_string("1999/12/31")  # -> ['1999', '/', '12', '/', '31']


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

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

leaveWhitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *leave_whitespace*

leave_whitespace(*recursive: bool = True*) → pyparsing.core.ParserElement
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.

Parameters **recursive** – If true (the default), also disable whitespace skipping in child elements (if any)

matches(*test_string: str*, *parse_all: bool = True*, ***, *parseAll: bool = True*) → bool
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:

- `test_string` - to test against this expression for a match
- `parse_all` - (default= `True`) - flag to pass to *parse_string* when running tests

Example:

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

parseFile(*file_or_filename: Union[str, pathlib.Path, TextIO], encoding: str = 'utf-8', parse_all: bool = False, *, parseAll: bool = False*) → pyparsing.results.ParseResults
Deprecated - use *parse_file*

parseString(*instring: str, parse_all: bool = False, *, parseAll: bool = False*) → pyparsing.results.ParseResults
Deprecated - use *parse_string*

parseWithTabs() → pyparsing.core.ParserElement
Deprecated - use *parse_with_tabs*

parse_file(*file_or_filename: Union[str, pathlib.Path, TextIO], encoding: str = 'utf-8', parse_all: bool = False, *, parseAll: bool = False*) → pyparsing.results.ParseResults
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.

parse_string(*instring: str, parse_all: bool = False, *, parseAll: bool = False*) → pyparsing.results.ParseResults
Parse a string with respect to the parser definition. This function is intended as the primary interface to the client code.

Parameters

- **instring** – The input string to be parsed.
- **parse_all** – If set, the entire input string must match the grammar.
- **parseAll** – retained for pre-PEP8 compatibility, will be removed in a future release.

Raises ***ParseException*** – Raised if `parse_all` is set and the input string does not match the whole grammar.

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

If the input string is required to match the entire grammar, `parse_all` flag must be set to `True`. This is also equivalent to ending the grammar with *StringEnd*().

To report proper column numbers, `parse_string` operates on a copy of the input string where all tabs are converted to spaces (8 spaces per tab, as per the default in `string.expandtabs`). If the input string contains tabs and the grammar uses parse actions that use the `loc` argument to index into the string being parsed, one can ensure a consistent view of the input string by doing one of the following:

- calling `parse_with_tabs` on your grammar before calling `parse_string` (see *parse_with_tabs*),
- define your parse action using the full `(s,loc,toks)` signature, and reference the input string using the parse action's `s` argument, or
- explicitly expand the tabs in your input string before calling `parse_string`.

Examples:

By default, partial matches are OK.

```
>>> res = Word('a').parse_string('aaaaabaaa')
>>> print(res)
['aaaaa']
```

The parsing behavior varies by the inheriting class of this abstract class. Please refer to the children directly to see more examples.

It raises an exception if parse_all flag is set and instring does not match the whole grammar.

```
>>> res = Word('a').parse_string('aaaaabaaa', parse_all=True)
Traceback (most recent call last):
...
pyparsing.ParseException: Expected end of text, found 'b'  (at char 5),␣
↪(line:1, col:6)
```

parse_with_tabs() → pyparsing.core.ParserElement
Overrides default behavior to expand <TAB> s to spaces before parsing the input string. Must be called before parse_string when the input grammar contains elements that match <TAB> characters.

runTests(*tests: Union[str, List[str]], parse_all: bool = True, comment: Union[ParserElement, str, None] = '#', full_dump: bool = True, print_results: bool = True, failure_tests: bool = False, post_parse: Optional[Callable[[str, pyparsing.results.ParseResults], str]] = None, file: Optional[TextIO] = None, with_line_numbers: bool = False, *, parseAll: bool = True, fullDump: bool = True, printResults: bool = True, failureTests: bool = False, postParse: Optional[Callable[[str, pyparsing.results.ParseResults], str]] = None*) → Tuple[bool, List[Tuple[str, Union[pyparsing.results.ParseResults, Exception]]]]
Deprecated - use *run_tests*

run_tests(*tests: Union[str, List[str]], parse_all: bool = True, comment: Union[ParserElement, str, None] = '#', full_dump: bool = True, print_results: bool = True, failure_tests: bool = False, post_parse: Optional[Callable[[str, pyparsing.results.ParseResults], str]] = None, file: Optional[TextIO] = None, with_line_numbers: bool = False, *, parseAll: bool = True, fullDump: bool = True, printResults: bool = True, failureTests: bool = False, postParse: Optional[Callable[[str, pyparsing.results.ParseResults], str]] = None*) → Tuple[bool, List[Tuple[str, Union[pyparsing.results.ParseResults, Exception]]]]
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
- parse_all - (default= True) - flag to pass to *parse_string* when running tests
- comment - (default= '#') - expression for indicating embedded comments in the test string; pass None to disable comment filtering
- full_dump - (default= True) - dump results as list followed by results names in nested outline; if False, only dump nested list
- print_results - (default= True) prints test output to stdout
- failure_tests - (default= False) indicates if these tests are expected to fail parsing
- post_parse - (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
- with_line_numbers - default= False) show test strings with line and column numbers

Returns: a (success, results) tuple, where success indicates that all tests succeeded (or failed if failure_tests 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.run_tests('''
```

(continues on next page)

(continued from previous page)

```
    # 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.run_tests('''
    # stray character
    100Z
    # missing leading digit before '.'
    -.100
    # too many '.'
    3.14.159
    ''', failure_tests=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
   ^
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)
```

(continues on next page)

(continued from previous page)

```
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.run_tests(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: str*, *max_matches: int = 9223372036854775807*, *overlap: bool = False*, ***, *debug: bool = False*, *maxMatches: int = 9223372036854775807*) → Generator[Tuple[pyparsing.results.ParseResults, int, int], None, None]

Deprecated - use *scan_string*

scan_string(*instring: str*, *max_matches: int = 9223372036854775807*, *overlap: bool = False*, ***, *debug: bool = False*, *maxMatches: int = 9223372036854775807*) → Generator[Tuple[pyparsing.results.ParseResults, int, int], None, None]

Scan the input string for expression matches. Each match will return the matching tokens, start location, and end location. May be called with optional `max_matches` 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 *parse_string* for more information on parsing strings with embedded tabs.

Example:

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

prints:

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

searchString(*instring: str*, *max_matches: int = 9223372036854775807*, ***, *debug: bool = False*, *maxMatches: int = 9223372036854775807*) → pyparsing.results.ParseResults

Deprecated - use *search_string*

search_string(*instring: str*, *max_matches: int = 9223372036854775807*, ***, *debug: bool = False*, *maxMatches: int = 9223372036854775807*) → pyparsing.results.ParseResults

Another extension to *scan_string*, simplifying the access to the tokens found to match the given parse expression. May be called with optional `max_matches` 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.search_string("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.search_string("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(*break_flag: bool = True*) → pyparsing.core.ParserElement
Deprecated - use *set_break*

setDebug(*flag: bool = True*, *recurse: bool = False*) → pyparsing.core.ParserElement
Deprecated - use *set_debug*

setDebugActions(*start_action: Callable[[str, int, ParserElement, bool], None]*, *success_action: Callable[[str, int, int, ParserElement, pyparsing.results.ParseResults, bool], None]*, *exception_action: Callable[[str, int, ParserElement, Exception, bool], None]*) → pyparsing.core.ParserElement
Deprecated - use *set_debug_actions*

static setDefaultWhitespaceChars(*chars: str*) → None
Deprecated - use *set_default_whitespace_chars*

setFailAction(*fn: Callable[[str, int, ParserElement, Exception], None]*) → pyparsing.core.ParserElement
Deprecated - use *set_fail_action*

setName(*name: str*) → pyparsing.core.ParserElement
Deprecated - use *set_name*

setParseAction(**fns*, ***kwargs*) → pyparsing.core.ParserElement
Deprecated - use *set_parse_action*

setResultsName(*name: str*, *list_all_matches: bool = False*, ***, *listAllMatches: bool = False*) → pyparsing.core.ParserElement
Deprecated - use *set_results_name*

setWhitespaceChars(*chars: Union[Set[str], str]*, *copy_defaults: bool = False*) → pyparsing.core.ParserElement
Deprecated - use *set_whitespace_chars*

set_break(*break_flag: bool = True*) → pyparsing.core.ParserElement
Method to invoke the Python pdb debugger when this element is about to be parsed. Set `break_flag` to `True` to enable, `False` to disable.

set_debug(*flag: bool = True*, *recurse: bool = False*) → pyparsing.core.ParserElement
Enable display of debugging messages while doing pattern matching. Set `flag` to `True` to enable, `False` to disable. Set `recurse` to `True` to set the debug flag on this expression and all sub-expressions.

Example:

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

# turn on debugging for wd
wd.set_debug()

term[1, ...].parse_string("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 *set_debug_actions*. 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 *set_name* 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 `set_name` is `"W:(A-Za-z)"`.

set_debug_actions(*start_action: Callable[[str, int, ParserElement, bool], None]*, *success_action: Callable[[str, int, int, ParserElement, pyparsing.results.ParseResults, bool], None]*, *exception_action: Callable[[str, int, ParserElement, Exception, bool], None]*) → pyparsing.core.ParserElement

Customize display of debugging messages while doing pattern matching:

- `start_action` - method to be called when an expression is about to be parsed; should have the signature `fn(input_string: str, location: int, expression: ParserElement, cache_hit: bool)`
- `success_action` - method to be called when an expression has successfully parsed; should have the signature `fn(input_string: str, start_location: int, end_location: int, expression: ParserELement, parsed_tokens: ParseResults, cache_hit: bool)`
- `exception_action` - method to be called when expression fails to parse; should have the signature `fn(input_string: str, location: int, expression: ParserElement, exception: Exception, cache_hit: bool)`

static set_default_whitespace_chars(*chars: str*) → None

Overrides the default whitespace chars

Example:

```
# default whitespace chars are space, <TAB> and newline
Word(alphas)[1, ...].parse_string("abc def\nghi jkl")  # -> ['abc', 'def',
↪'ghi', 'jkl']
```

(continues on next page)

(continued from previous page)

```
# change to just treat newline as significant
ParserElement.set_default_whitespace_chars(" \t")
Word(alphas)[1, ...].parse_string("abc def\nghi jkl")  # -> ['abc', 'def']
```

set_fail_action (*fn: Callable[[str, int, ParserElement, Exception], None]*) → pyparsing.core.ParserElement

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.

set_name (*name: str*) → pyparsing.core.ParserElement

Define name for this expression, makes debugging and exception messages clearer.

Example:

```
Word(nums).parse_string("ABC")  # -> Exception: Expected W:(0-9) (at char 0),␣
↪(line:1, col:1)
Word(nums).set_name("integer").parse_string("ABC")  # -> Exception: Expected␣
↪integer (at char 0), (line:1, col:1)
```

set_parse_action (**fns*, ***kwargs*) → pyparsing.core.ParserElement

Define one or more actions to perform when successfully matching parse element definition.

Parse actions can be called to perform data conversions, do extra validation, update external data structures, or enhance or replace the parsed tokens. Each 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

The parsed tokens are passed to the parse action as ParseResults. They can be modified in place using list-style append, extend, and pop operations to update the parsed list elements; and with dictionary-style item set and del operations to add, update, or remove any named results. If the tokens are modified in place, it is not necessary to return them with a return statement.

Parse actions can also completely replace the given tokens, with another `ParseResults` object, or with some entirely different object (common for parse actions that perform data conversions). A convenient way to build a new parse result is to define the values using a dict, and then create the return value using *ParseResults.from_dict*.

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

Optional keyword arguments:

- `call_during_try` = (default= `False`) indicate if parse action should be run during lookaheads and alternate testing. For parse actions that have side effects, it is important to only call the parse action once it is determined that it is being called as part of a successful parse. For parse actions that perform additional validation, then call_during_try should be passed as True, so that the validation code is included in the preliminary "try" parses.

Note: the default parsing behavior is to expand tabs in the input string before starting the parsing process. See *parse_string* 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:

```
# parse dates in the form YYYY/MM/DD

# use parse action to convert toks from str to int at parse time
def convert_to_int(toks):
    return int(toks[0])

# use a parse action to verify that the date is a valid date
def is_valid_date(instring, loc, toks):
    from datetime import date
    year, month, day = toks[::2]
    try:
        date(year, month, day)
    except ValueError:
        raise ParseException(instring, loc, "invalid date given")

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

# add parse actions
integer.set_parse_action(convert_to_int)
date_str.set_parse_action(is_valid_date)

# note that integer fields are now ints, not strings
date_str.run_tests('''
    # successful parse - note that integer fields were converted to ints
    1999/12/31

    # fail - invalid date
    1999/13/31
    ''')
```

set_results_name(*name: str*, *list_all_matches: bool = False*, *, *listAllMatches: bool = False*) → pyparsing.core.ParserElement

Define name for referencing matching tokens as a nested attribute of the returned parse results.

Normally, results names are assigned as you would assign keys in a dict: any existing value is overwritten by later values. If it is necessary to keep all values captured for a particular results name, call `set_results_name` with `list_all_matches` = True.

NOTE: `set_results_name` 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.set_results_name("name")` - see *__call__*. If `list_all_matches` is required, use `expr("name*")`.

Example:

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

(continues on next page)

(continued from previous page)

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

set_whitespace_chars(*chars: Union[Set[str], str]*, *copy_defaults: bool = False*) → pyparsing.core.ParserElement
Overrides the default whitespace chars

split(*instring: str*, *maxsplit: int = 9223372036854775807*, *include_separators: bool = False*, *, includeSeparators=False*) → Generator[str, None, None]
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 `include_separators` argument (default= `False`), if the separating matching text should be included in the split results.

Example:

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

prints:

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

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

suppress_warning(*warning_type: pyparsing.core.Diagnostics*) → pyparsing.core.ParserElement
Suppress warnings emitted for a particular diagnostic on this expression.

Example:

```
base = pp.Forward()
base.suppress_warning(Diagnostics.warn_on_parse_using_empty_Forward)

# statement would normally raise a warning, but is now suppressed
print(base.parse_string("x"))
```

transformString(*instring: str*, *, *debug: bool = False*) → str
Deprecated - use *transform_string*

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

Example:

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

print(wd.transform_string("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: str*, *loc: int*, *, *raise_fatal: bool = False*, *do_actions: bool = False*) → int
Deprecated - use `try_parse`

classmethod using_each(*seq*, ***class_kwargs*)
Yields a sequence of class(obj, **class_kwargs) for obj in seq.

Example:

```
LPAR, RPAR, LBRACE, RBRACE, SEMI = Suppress.using_each("(){};")
```

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

visit_all()
General-purpose method to yield all expressions and sub-expressions in a grammar. Typically just for internal use.

class pyparsing.**PositionToken**
Bases: `pyparsing.core.Token`

__init__()
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**QuotedString**(*quote_char: str = ''*, *esc_char: Optional[str] = None*, *esc_quote: Optional[str] = None*, *multiline: bool = False*, *unquote_results: bool = True*, *end_quote_char: Optional[str] = None*, *convert_whitespace_escapes: bool = True*, *, *quoteChar: str = ''*, *escChar: Optional[str] = None*, *escQuote: Optional[str] = None*, *unquoteResults: bool = True*, *endQuoteChar: Optional[str] = None*, *convertWhitespaceEscapes: bool = True*)
Bases: `pyparsing.core.Token`

Token for matching strings that are delimited by quoting characters.

Defined with the following parameters:

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

Example:

```
qs = QuotedString('"')
print(qs.search_string('lsjdf "This is the quote" sldjf'))
complex_qs = QuotedString('{{', end_quote_char='}}')
print(complex_qs.search_string('lsjdf {{This is the "quote"}} sldjf'))
```

(continues on next page)

(continued from previous page)

```
sql_qs = QuotedString('"', esc_quote='""')
print(sql_qs.search_string('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']]
```

__init__(*quote_char: str = ”, esc_char: Optional[str] = None, esc_quote: Optional[str] = None, multiline: bool = False, unquote_results: bool = True, end_quote_char: Optional[str] = None, convert_whitespace_escapes: bool = True, *, quoteChar: str = ”, escChar: Optional[str] = None, escQuote: Optional[str] = None, unquoteResults: bool = True, endQuoteChar: Optional[str] = None, convertWhitespaceEscapes: bool = True*)

Initialize self. See help(type(self)) for accurate signature.

exception pyparsing.**RecursiveGrammarException**(*parseElementList*)

Bases: Exception

Exception thrown by *ParserElement.validate* if the grammar could be left-recursive; parser may need to enable left recursion using *ParserElement.enable_left_recursion*

__init__(*parseElementList*)

Initialize self. See help(type(self)) for accurate signature.

__str__() → str

Return str(self).

__weakref__

list of weak references to the object (if defined)

class pyparsing.**Regex**(*pattern: Any, flags: Union[re.RegexFlag, int] = 0, as_group_list: bool = False, as_match: bool = False, *, asGroupList: bool = False, asMatch: bool = False*)

Bases: pyparsing.core.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 *ParseResults*.

If instead of the Python stdlib `re` module you wish to use a different RE module (such as the `regex` module), you can do so by building your `Regex` object with a compiled RE that was compiled using `regex`.

Example:

```
realnum = Regex(r"[+-]?\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})")

# named fields in a regex will be returned as named results
date = Regex(r'(?P<year>\d{4})-(?P<month>\d\d?)-(?P<day>\d\d?)')

# the Regex class will accept re's compiled using the regex module
import regex
parser = pp.Regex(regex.compile(r'[0-9]'))
```

__init__(*pattern: Any*, *flags: Union[re.RegexFlag, int] = 0*, *as_group_list: bool = False*, *as_match: bool = False*, *, *asGroupList: bool = False*, *asMatch: bool = False*)
The parameters `pattern` and `flags` are passed to the `re.compile()` function as-is. See the Python re module module for an explanation of the acceptable patterns and flags.

sub(*repl: str*) → pyparsing.core.ParserElement
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.transform_string("h1:main title:"))
# prints "<h1>main title</h1>"
```

class pyparsing.**SkipTo**(*other: Union[pyparsing.core.ParserElement, str]*, *include: bool = False*, *ignore: Union[pyparsing.core.ParserElement, str, None] = None*, *fail_on: Union[pyparsing.core.ParserElement, str, None] = None*, *, *failOn: Union[pyparsing.core.ParserElement, str, None] = None*)
Bases: `pyparsing.core.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` - if `True`, the target expression is also parsed (the skipped text and target expression are returned as a 2-element list) (default= `False`).
- `ignore` - (default= `None`) used to define grammars (typically quoted strings and comments) that might contain false matches to the target expression
- `fail_on` - (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
    -----+----------+-------------------------------------------+-----------
     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=quoted_string)
string_data.set_parse_action(token_map(str.strip))
ticket_expr = (integer("issue_num") + SEP
              + string_data("sev") + SEP
              + string_data("desc") + SEP
              + integer("days_open"))
```

(continues on next page)

(continued from previous page)

```
for tkt in ticket_expr.search_string(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'
```

__init__(*other: Union[pyparsing.core.ParserElement, str], include: bool = False, ignore: Union[pyparsing.core.ParserElement, str, None] = None, fail_on: Union[pyparsing.core.ParserElement, str, None] = None, *, failOn: Union[pyparsing.core.ParserElement, str, None] = None*)

Initialize self. See help(type(self)) for accurate signature.

ignore(*expr*)

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 = Word(alphas)[1, ...]
patt.parse_string('ablaj /* comment */ lskjd')
# -> ['ablaj']

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

class pyparsing.**StringEnd**

Bases: pyparsing.core.PositionToken

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

__init__()

Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**StringStart**

Bases: pyparsing.core.PositionToken

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

__init__()

Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Suppress**(*expr: Union[pyparsing.core.ParserElement, str], savelist: bool = False*)

Bases: pyparsing.core.TokenConverter

Converter for ignoring the results of a parsed expression.

Example:

```
source = "a, b, c,d"
wd = Word(alphas)
wd_list1 = wd + (',' + wd)[...]
print(wd_list1.parse_string(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 + (Suppress(',') + wd)[...]
print(wd_list2.parse_string(source))

# Skipped text (using '...') can be suppressed as well
source = "lead in START relevant text END trailing text"
start_marker = Keyword("START")
end_marker = Keyword("END")
find_body = Suppress(...) + start_marker + ... + end_marker
print(find_body.parse_string(source)
```

prints:

```
['a', ',', 'b', ',', 'c', ',', 'd']
['a', 'b', 'c', 'd']
['START', 'relevant text ', 'END']
```

(See also *DelimitedList*.)

__add__(*other*) → pyparsing.core.ParserElement
Implementation of + operator - returns *And*. Adding strings to a *ParserElement* converts them to *Literal*s by default.

Example:

```
greet = Word(alphas) + "," + Word(alphas) + "!"
hello = "Hello, World!"
print(hello, "->", greet.parse_string(hello))
```

prints:

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

`...` may be used as a parse expression as a short form of *SkipTo*:

```
Literal('start') + ... + Literal('end')
```

is equivalent to:

```
Literal('start') + SkipTo('end')("_skipped*") + Literal('end')
```

Note that the skipped text is returned with '_skipped' as a results name, and to support having multiple skips in the same parser, the value returned is a list of all skipped text.

__init__(*expr: Union[pyparsing.core.ParserElement, str], savelist: bool = False*)
Initialize self. See help(type(self)) for accurate signature.

__sub__(*other*) → pyparsing.core.ParserElement
Implementation of − operator, returns *And* with error stop

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

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

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

__init__()
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**TokenConverter**(*expr: Union[pyparsing.core.ParserElement, str], savelist=False*)
Bases: pyparsing.core.ParseElementEnhance

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

__init__(*expr: Union[pyparsing.core.ParserElement, str], savelist=False*)
Initialize self. See help(type(self)) for accurate signature.

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

__init__(*ws: str = ' \t\r\n'*, *min: int = 1*, *max: int = 0*, *exact: int = 0*)
Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Word**(*init_chars: str = ''*, *body_chars: Optional[str] = None*, *min: int = 1*, *max: int = 0*, *exact: int = 0*, *as_keyword: bool = False*, *exclude_chars: Optional[str] = None*, ***, *initChars: Optional[str] = None*, *bodyChars: Optional[str] = None*, *asKeyword: bool = False*, *excludeChars: Optional[str] = None*)
Bases: pyparsing.core.Token

Token for matching words composed of allowed character sets.

Parameters:

- `init_chars` - string of all characters that should be used to match as a word; "ABC" will match "AAA", "ABAB", "CBAC", etc.; if `body_chars` is also specified, then this is the string of initial characters
- `body_chars` - string of characters that can be used for matching after a matched initial character as given in `init_chars`; if omitted, same as the initial characters (default=''None'')
- `min` - minimum number of characters to match (default=1)
- `max` - maximum number of characters to match (default=0)
- `exact` - exact number of characters to match (default=0)
- `as_keyword` - match as a keyword (default=''False'')
- `exclude_chars` - characters that might be found in the input `body_chars` string but which should not be accepted for matching ;useful to define a word of all printables except for one or two characters, for instance (default=''None'')

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)

alphas, nums, and printables are also defined in several Unicode sets - see pyparsing_unicode`.

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, exclude_chars=",")
```

__init__(*init_chars: str = ''*, *body_chars: Optional[str] = None*, *min: int = 1*, *max: int = 0*, *exact: int = 0*, *as_keyword: bool = False*, *exclude_chars: Optional[str] = None*, ***, *initChars: Optional[str] = None*, *bodyChars: Optional[str] = None*, *asKeyword: bool = False*, *excludeChars: Optional[str] = None*)

Initialize self. See help(type(self)) for accurate signature.

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

Bases: pyparsing.core.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 word_chars (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.

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

Initialize self. See help(type(self)) for accurate signature.

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

Bases: pyparsing.core.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 word_chars (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.

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

Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**ZeroOrMore**(*expr: Union[str, pyparsing.core.ParserElement], stop_on: Union[pyparsing.core.ParserElement, str, None] = None, *, stopOn: Union[pyparsing.core.ParserElement, str, None] = None*)

Bases: pyparsing.core._MultipleMatch

Optional repetition of zero or more of the given expression.

Parameters:

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

Example: similar to *OneOrMore*

__init__(*expr: Union[str, pyparsing.core.ParserElement], stop_on: Union[pyparsing.core.ParserElement, str, None] = None, *, stopOn: Union[pyparsing.core.ParserElement, str, None] = None*)

Initialize self. See help(type(self)) for accurate signature.

class pyparsing.**Char**(*charset: str*, *as_keyword: bool = False*, *exclude_chars: Optional[str] = None*, *, *asKeyword: bool = False*, *excludeChars: Optional[str] = None*)

Bases: pyparsing.core.Word

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

__init__(*charset: str*, *as_keyword: bool = False*, *exclude_chars: Optional[str] = None*, *, *asKeyword: bool = False*, *excludeChars: Optional[str] = None*)

Initialize self. See help(type(self)) for accurate signature.

pyparsing.**autoname_elements**() → None

Utility to simplify mass-naming of parser elements, for generating railroad diagram with named subdiagrams.

pyparsing.**col**

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.parse_string* 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.**condition_as_parse_action**(*fn: Union[Callable[[], bool], Callable[[pyparsing.results.ParseResults], bool], Callable[[int, pyparsing.results.ParseResults], bool], Callable[[str, int, pyparsing.results.ParseResults], bool]], message: Optional[str] = None, fatal: bool = False*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

Function to convert a simple predicate function that returns `True` or `False` into a parse action. Can be used in places when a parse action is required and *ParserElement.add_condition* cannot be used (such as when adding a condition to an operator level in *infix_notation*).

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*

pyparsing.**counted_array**(*expr: pyparsing.core.ParserElement, int_expr: Optional[pyparsing.core.ParserElement] = None, *, intExpr: Optional[pyparsing.core.ParserElement] = None*) → pyparsing.core.ParserElement

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

Example:

```
counted_array(Word(alphas)).parse_string('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
binary_constant = Word('01').set_parse_action(lambda t: int(t[0], 2))
counted_array(Word(alphas), int_expr=binary_constant).parse_string('10 ab cd ef')␣
↪ # -> ['ab', 'cd']

# if other fields must be parsed after the count but before the
# list items, give the fields results names and they will
# be preserved in the returned ParseResults:
count_with_metadata = integer + Word(alphas)("type")
typed_array = counted_array(Word(alphanums), int_expr=count_with_metadata)("items
↪")
result = typed_array.parse_string("3 bool True True False")
print(result.dump())

# prints
# ['True', 'True', 'False']
# - items: ['True', 'True', 'False']
# - type: 'bool'
```

pyparsing.**delimited_list**(*expr: Union[str, pyparsing.core.ParserElement], delim: Union[str, pyparsing.core.ParserElement] = ',', combine: bool = False, min: Optional[int] = None, max: Optional[int] = None, *, allow_trailing_delim: bool = False*)

Deprecated - use *DelimitedList*

pyparsing.**dict_of**(*key: pyparsing.core.ParserElement*, *value: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

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, stop_on=label).set_
↪parse_action(' '.join))
print(attr_expr[1, ...].parse_string(text).dump())

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

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

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.**infix_notation**(*base_expr: pyparsing.core.ParserElement, op_list: List[Union[Tuple[Union[pyparsing.core.ParserElement, str], Tuple[Union[pyparsing.core.ParserElement, str], Union[pyparsing.core.ParserElement, str]]], int, pyparsing.helpers.OpAssoc, Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any], None]], Tuple[Union[pyparsing.core.ParserElement, str], Tuple[Union[pyparsing.core.ParserElement, str], Union[pyparsing.core.ParserElement, str]]], int, pyparsing.helpers.OpAssoc]]], lpar: Union[str, pyparsing.core.ParserElement] = Suppress:('('), rpar: Union[str, pyparsing.core.ParserElement] = Suppress:(')')*) → pyparsing.core.ParserElement

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 infix_notation. See *ParserElement.enable_packrat* for a mechanism to potentially improve your parser performance.

Parameters:

- `base_expr` - expression representing the most basic operand to be used in the expression
- `op_list` - list of tuples, one for each operator precedence level in the expression grammar; each tuple is of the form `(op_expr, num_operands, right_left_assoc, (optional)parse_action)`, where:
  - `op_expr` is the pyparsing expression for the operator; may also be a string, which will be converted to a Literal; if `num_operands` is 3, `op_expr` is a tuple of two expressions, for the two operators separating the 3 terms
  - `num_operands` is the number of terms for this operator (must be 1, 2, or 3)
  - `right_left_assoc` is the indicator whether the operator is right or left associative, using the pyparsing-defined constants `OpAssoc.RIGHT` and `OpAssoc.LEFT`.
  - `parse_action` 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 `set_parse_action(*fn)` (*ParserElement.set_parse_action*)
- `lpar` - expression for matching left-parentheses; if passed as a str, then will be parsed as `Suppress(lpar)`. If lpar is passed as an expression (such as `Literal('(')`), then it will be kept in the parsed results, and grouped with them. (default= `Suppress('(')`)
- `rpar` - expression for matching right-parentheses; if passed as a str, then will be parsed as `Suppress(rpar)`. If rpar is passed as an expression (such as `Literal(')')`), then it will be kept in the parsed results, and grouped with them. (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 = infix_notation(integer | varname,
    [
    ('-', 1, OpAssoc.RIGHT),
    (one_of('* /'), 2, OpAssoc.LEFT),
    (one_of('+ -'), 2, OpAssoc.LEFT),
    ])

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

prints:

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

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

(5+x)*y
[[[5, '+', 'x'], '*', 'y']]

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

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

pyparsing.**lineno**
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.parse_string* 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.**make_html_tags**(*tag_str: Union[str, pyparsing.core.ParserElement]*) → Tuple[pyparsing.core.ParserElement, pyparsing.core.ParserElement]
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>'
# make_html_tags returns pyparsing expressions for the opening and
# closing tags as a 2-tuple
a, a_end = make_html_tags("A")
link_expr = a + SkipTo(a_end)("link_text") + a_end

for link in link_expr.search_string(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.**make_xml_tags**(*tag_str: Union[str, pyparsing.core.ParserElement]*) → Tuple[pyparsing.core.ParserElement, pyparsing.core.ParserElement]
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 *make_html_tags*

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

pyparsing.**match_previous_expr**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement
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 = match_previous_expr(first)
match_expr = 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.**match_previous_literal**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

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 = match_previous_literal(first)
match_expr = 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 *match_previous_expr*. Do *not* use with packrat parsing enabled.

pyparsing.**nested_expr**(*opener: Union[str, pyparsing.core.ParserElement] = '(', closer: Union[str, pyparsing.core.ParserElement] = ')', content: Optional[pyparsing.core.ParserElement] = None, ignore_expr: pyparsing.core.ParserElement = quoted string using single or double quotes, *, ignoreExpr: pyparsing.core.ParserElement = quoted string using single or double quotes*) → pyparsing.core.ParserElement

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`)
- `ignore_expr` - expression for ignoring opening and closing delimiters (default= `quoted_string`)
- `ignoreExpr` - this pre-PEP8 argument is retained for compatibility but will be removed in a future release

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 `ignore_expr` 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 quoted_string or a comment expression. Specify multiple expressions using an *Or* or *MatchFirst*. The default is `quoted_string`, but if no expressions are to be ignored, then pass `None` for this argument.

Example:

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

code_body = nested_expr('{', '}', ignore_expr=(quoted_string | c_style_comment))
```

(continues on next page)

(continued from previous page)

```
c_function = (decl_data_type("type")
              + ident("name")
              + LPAR + Opt(DelimitedList(arg), [])("args") + RPAR
              + code_body("body"))
c_function.ignore(c_style_comment)

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'));
        }
    }
'''
for func in c_function.search_string(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.**null_debug_action**(**args*)

'Do-nothing' debug action, to suppress debugging output during parsing.

pyparsing.**one_of**(*strs: Union[Iterable[str], str], caseless: bool = False, use_regex: bool = True, as_keyword: bool = False, *, useRegex: bool = True, asKeyword: bool = False*) → pyparsing.core.ParserElement

Helper to quickly define a set of alternative *Literal* s, 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` - treat all literals as caseless - (default= `False`)
- `use_regex` - as an optimization, will generate a *Regex* object; otherwise, will generate a *MatchFirst* object (if `caseless=True` or `as_keyword=True`, or if creating a *Regex* raises an exception) - (default= `True`)
- `as_keyword` - enforce *Keyword*-style matching on the generated expressions - (default= `False`)
- `asKeyword` and `useRegex` are retained for pre-PEP8 compatibility, but will be removed in a future release

Example:

```
comp_oper = one_of("< = > <= >= !=")
var = Word(alphas)
number = Word(nums)
term = var | number
```

(continues on next page)

(continued from previous page)

```
comparison_expr = term + comp_oper + term
print(comparison_expr.search_string("B = 12  AA=23 B<=AA AA>12"))
```

prints:

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

pyparsing.**original_text_for**(*expr: pyparsing.core.ParserElement*, *as_string: bool = True*, ***, *asString: bool = True*) → pyparsing.core.ParserElement

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 a string containing the original parsed text.

If the optional `as_string` 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 *`original_text_for`* contains expressions with defined results names, you must set `as_string` to `False` if you want to preserve those results name values.

The `asString` pre-PEP8 argument is retained for compatibility, but will be removed in a future release.

Example:

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

prints:

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

class pyparsing.**pyparsing_common**

Bases: `object`

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`*
- *`url`*

Parse actions:

- *`convert_to_integer`*
- *`convert_to_float`*
- *`convert_to_date`*
- *`convert_to_datetime`*
- *`strip_html_tags`*

- *upcase_tokens*
- *downcase_tokens*

Example:

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

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

pyparsing_common.hex_integer.run_tests('''
    # hex numbers
    100
    FF
    ''')

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

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

import uuid
pyparsing_common.uuid.set_parse_action(token_map(uuid.UUID))
pyparsing_common.uuid.run_tests('''
    # uuid
    12345678-1234-5678-1234-567812345678
    ''')
```

prints:

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

(continues on next page)

(continued from previous page)

```
-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]

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]
```

(continues on next page)

(continued from previous page)

```
1-3/4
[1.75]

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

__weakref__
list of weak references to the object (if defined)

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

static convertToDate (*fmt: str = '%Y-%m-%d'*)
Deprecated - use *convert_to_date*

static convertToDatetime (*fmt: str = '%Y-%m-%dT%H:%M:%S.%f'*)
Deprecated - use *convert_to_datetime*

convertToFloat (*l*, *t*)
Deprecated - use *convert_to_float*

convertToInteger (*l*, *t*)
Deprecated - use *convert_to_integer*

static convert_to_date (*fmt: str = '%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.set_parse_action(pyparsing_common.convert_to_date())
print(date_expr.parse_string("1999-12-31"))
```

prints:

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

static convert_to_datetime (*fmt: str = '%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.set_parse_action(pyparsing_common.convert_to_datetime())
print(dt_expr.parse_string("1999-12-31T23:59:59.999"))
```

prints:

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

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

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

static downcaseTokens(*s*, *l*, *t*)
Deprecated - use *downcase_tokens*

static downcase_tokens(*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 = number
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: str*, *l: int*, *tokens: pyparsing.results.ParseResults*)
Deprecated - use *strip_html_tags*

static strip_html_tags(*s: str*, *l: int*, *tokens: pyparsing.results.ParseResults*)
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 = make_html_tags("TD")
table_text = td + SkipTo(td_end).set_parse_action(pyparsing_common.strip_html_
↪tags)("body") + td_end
print(table_text.parse_string(text).body)
```

Prints:

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

static upcaseTokens(*s*, *l*, *t*)
Deprecated - use *upcase_tokens*

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

url = url
URL (http/https/ftp scheme)

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

class pyparsing.**pyparsing_test**
Bases: object

namespace class for classes useful in writing unit tests

class TestParseResultsAsserts
Bases: object

A mixin class to add parse results assertion methods to normal unittest.TestCase classes.

__weakref__
list of weak references to the object (if defined)

assertParseAndCheckDict(*expr*, *test_string*, *expected_dict*, *msg=None*, *verbose=True*)
Convenience wrapper assert to test a parser element and input string, and assert that the resulting ParseResults.asDict() is equal to the expected_dict.

assertParseAndCheckList(*expr*, *test_string*, *expected_list*, *msg=None*, *verbose=True*)
Convenience wrapper assert to test a parser element and input string, and assert that the resulting ParseResults.asList() is equal to the expected_list.

assertParseResultsEquals(*result*, *expected_list=None*, *expected_dict=None*, *msg=None*)
Unit test assertion to compare a *ParseResults* object with an optional expected_list, and compare any defined results names with an optional expected_dict.

assertRunTestResults(*run_tests_report*, *expected_parse_results=None*, *msg=None*)
Unit test assertion to evaluate output of ParserElement.runTests(). If a list of list-dict tuples is given as the expected_parse_results argument, then these are zipped with the report tuples returned by runTests and evaluated using assertParseResultsEquals. Finally, asserts that the overall runTests() success value is True.

Parameters
- **run_tests_report** – tuple(bool, [tuple(str, ParseResults or Exception)]) returned from runTests
- **(optional)** (*expected_parse_results*) – [tuple(str, list, dict, Exception)]

__weakref__
list of weak references to the object (if defined)

class reset_pyparsing_context
Bases: object

Context manager to be used when writing unit tests that modify pyparsing config values: - packrat parsing - bounded recursion parsing - default whitespace characters. - default keyword characters - literal string auto-conversion class - __diag__ settings

Example:

```
with reset_pyparsing_context():
    # test that literals used to construct a grammar are automatically␣
↪suppressed
    ParserElement.inlineLiteralsUsing(Suppress)

    term = Word(alphas) | Word(nums)
    group = Group('(' + term[...] + ')')

    # assert that the '()' characters are not included in the parsed tokens
    self.assertParseAndCheckList(group, "(abc 123 def)", ['abc', '123', 'def
↪'])

# after exiting context manager, literals are converted to Literal␣
↪expressions again
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

static with_line_numbers(*s: str*, *start_line: Optional[int] = None*, *end_line: Optional[int] = None*, *expand_tabs: bool = True*, *eol_mark: str = '|'*, *mark_spaces: Optional[str] = None*, *mark_control: Optional[str] = None*) → str
Helpful method for debugging a parser - prints a string with line and column numbers. (Line and column numbers are 1-based.)

Parameters

- **s** – tuple(bool, str - string to be printed with line and column numbers
- **start_line** – int - (optional) starting line number in s to print (default=1)
- **end_line** – int - (optional) ending line number in s to print (default=len(s))
- **expand_tabs** – bool - (optional) expand tabs to spaces, to match the pyparsing default
- **eol_mark** – str - (optional) string to mark the end of lines, helps visualize trailing spaces (default="|")
- **mark_spaces** – str - (optional) special character to display in place of spaces
- **mark_control** – str - (optional) convert non-printing control characters to a placeholding character; valid values: - "unicode" - replaces control chars with Unicode symbols, such as "" and "" - any single character string - replace control characters with given string - None (default) - string is displayed as-is

Returns str - input string with leading line numbers and column number headers

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

A namespace class for defining common language unicode_sets.

class Arabic
Bases: pyparsing.unicode.unicode_set

Unicode set for Arabic Unicode Character Range

BMP
alias of *pyparsing_unicode.BasicMultilingualPlane*

class BasicMultilingualPlane
Bases: pyparsing.unicode.unicode_set

Unicode set for the Basic Multilingual Plane

class CJK
Bases: pyparsing.unicode.Chinese, pyparsing.unicode.Japanese, pyparsing.unicode.Hangul

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

class Chinese
Bases: pyparsing.unicode.unicode_set

Unicode set for Chinese Unicode Character Range

class Cyrillic
Bases: pyparsing.unicode.unicode_set

Unicode set for Cyrillic Unicode Character Range

class Devanagari
Bases: pyparsing.unicode.unicode_set

Unicode set for Devanagari Unicode Character Range

class Greek
Bases: pyparsing.unicode.unicode_set

Unicode set for Greek Unicode Character Ranges

class Hangul
Bases: pyparsing.unicode.unicode_set

Unicode set for Hangul (Korean) Unicode Character Range

class Hebrew
Bases: pyparsing.unicode.unicode_set

Unicode set for Hebrew Unicode Character Range

class Japanese
Bases: pyparsing.unicode.unicode_set

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

class Hiragana
Bases: pyparsing.unicode.unicode_set

Unicode set for Hiragana Unicode Character Range

class Kanji
Bases: pyparsing.unicode.unicode_set

Unicode set for Kanji Unicode Character Range

class Katakana
Bases: pyparsing.unicode.unicode_set

Unicode set for Katakana Unicode Character Range

alias of *pyparsing_unicode.Japanese.Hiragana*

alias of *pyparsing_unicode.Japanese.Katakana*

alias of *pyparsing_unicode.Japanese.Kanji*

Korean
alias of *pyparsing_unicode.Hangul*

class Latin1
Bases: pyparsing.unicode.unicode_set

Unicode set for Latin-1 Unicode Character Range

class LatinA
Bases: pyparsing.unicode.unicode_set

Unicode set for Latin-A Unicode Character Range

class LatinB
Bases: pyparsing.unicode.unicode_set

Unicode set for Latin-B Unicode Character Range

class Thai
Bases: pyparsing.unicode.unicode_set

Unicode set for Thai Unicode Character Range

Ελληνικ
alias of *pyparsing_unicode.Greek*

alias of *pyparsing_unicode.Cyrillic*

alias of *pyparsing_unicode.Arabic*

alias of *pyparsing_unicode.Thai*

alias of *pyparsing_unicode.Chinese*

alias of *pyparsing_unicode.Japanese*

alias of *pyparsing_unicode.Hangul*

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

Example:

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

# use remove_quotes to strip quotation marks from parsed results
quoted_string.set_parse_action(remove_quotes)
quoted_string.parse_string("'Now is the Winter of our Discontent'") # -> ["Now is␣
↪the Winter of our Discontent"]
```

pyparsing.**replace_with**(*repl_str*)

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

Example:

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

term[1, ...].parse_string("324 234 N/A 234") # -> [324, 234, nan, 234]
```

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

Helper parser action to replace common HTML entities with their special characters

pyparsing.**srange**(*s: str*) → str

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.**token_map**(*func*, **args*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

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).set_parse_action(token_map(int, 16))`, which will convert the parsed data to an integer using base 16.

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

```
hex_ints = Word(hexnums)[1, ...].set_parse_action(token_map(int, 16))
hex_ints.run_tests('''
    00 11 22 aa FF 0a 0d 1a
    ''')

upperword = Word(alphas).set_parse_action(token_map(str.upper))
upperword[1, ...].run_tests('''
    my kingdom for a horse
    ''')

wd = Word(alphas).set_parse_action(token_map(str.title))
wd[1, ...].set_parse_action(' '.join).run_tests('''
    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']
```

pyparsing.**trace_parse_action**(*f: Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

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)

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

wds = wd[1, ...].set_parse_action(remove_duplicate_chars)
print(wds.parse_string("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.**ungroup**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

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

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. Ranges can be specified using 2-tuples or a 1-tuple, such as:

```
_ranges = [
    (0x0020, 0x007e),
    (0x00a0, 0x00ff),
    (0x0100,),
    ]
```

Ranges are left- and right-inclusive. A 1-tuple of (x,) is treated as (x, x).

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

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

__weakref__
list of weak references to the object (if defined)

pyparsing.**with_attribute**(**args*, ***attr_dict*)
Helper to create a validating parse action to be used with start tags created with *make_xml_tags* or *make_html_tags*. Use with_attribute to qualify a starting tag with a required attribute value, to avoid false matches on common tags such as <TD> or <DIV>.

Call with_attribute 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 *with_class*.

To verify that the attribute exists, but without specifying a value, pass with_attribute.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 = make_html_tags("div")
```

(continues on next page)

(continued from previous page)

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

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

prints:

```
1 4 0 1 0

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

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

Simplified version of *with_attribute* 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 = make_html_tags("div")
div_grid = div().set_parse_action(with_class("grid"))

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

div_any_type = div().set_parse_action(with_class(withAttribute.ANY_VALUE))
div_expr = div_any_type + SkipTo(div | div_end)("body")
for div_header in div_expr.search_string(html):
    print(div_header.body)
```

prints:

```
1 4 0 1 0

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

pyparsing.**conditionAsParseAction**(*fn: Union[Callable[[], bool], Callable[[pyparsing.results.ParseResults], bool], Callable[[int, pyparsing.results.ParseResults], bool], Callable[[str, int, pyparsing.results.ParseResults], bool]], message: Optional[str] = None, fatal: bool = False*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

Deprecated - use *condition_as_parse_action*

pyparsing.**countedArray**(*expr: pyparsing.core.ParserElement, int_expr: Optional[pyparsing.core.ParserElement] = None, *, intExpr: Optional[pyparsing.core.ParserElement] = None*) → pyparsing.core.ParserElement

Deprecated - use *counted_array*

pyparsing.**delimitedList**(*expr: Union[str, pyparsing.core.ParserElement], delim: Union[str, pyparsing.core.ParserElement] = ',', combine: bool = False, min: Optional[int] = None, max: Optional[int] = None, *, allow_trailing_delim: bool = False*)

Deprecated - use *DelimitedList*

pyparsing.**dictOf**(*key: pyparsing.core.ParserElement, value: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

Deprecated - use *dict_of*

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

(DEPRECATED - use *IndentedBlock* class instead) 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`.

(Note that indentedBlock uses internal parse actions which make it incompatible with packrat parsing.)

Example:

```
data = '''
def A(z):
  A1
  B = 100
  G = A2
  A2
  A3
B
def BB(a,b,c):
  BB1
  def BBA():
    bba1
    bba2
```

(continues on next page)

(continued from previous page)

```
    bba3
C
D
def spam(x,y):
     def eggs(z):
         pass
'''


indentStack = [1]
stmt = Forward()

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

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

module_body = stmt[1, ...]

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.**infixNotation**(*base_expr: pyparsing.core.ParserElement, op_list: List[Union[Tuple[Union[pyparsing.core.ParserElement, str, Tuple[Union[pyparsing.core.ParserElement, str], Union[pyparsing.core.ParserElement, str]]], int, pyparsing.helpers.OpAssoc, Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any], None]], Tuple[Union[pyparsing.core.ParserElement, str, Tuple[Union[pyparsing.core.ParserElement, str], Union[pyparsing.core.ParserElement, str]]], int, pyparsing.helpers.OpAssoc]]], lpar: Union[str, pyparsing.core.ParserElement] = Suppress:('('), rpar: Union[str, pyparsing.core.ParserElement] = Suppress:(')')*) → pyparsing.core.ParserElement

Deprecated - use *infix_notation*

pyparsing.**locatedExpr**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

(DEPRECATED - future code should use the *Located* class) 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.parse_with_tabs*

Example:

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

prints:

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

pyparsing.**makeHTMLTags**(*tag_str: Union[str, pyparsing.core.ParserElement]*) → Tuple[pyparsing.core.ParserElement, pyparsing.core.ParserElement]

Deprecated - use *make_html_tags*

pyparsing.**makeXMLTags**(*tag_str: Union[str, pyparsing.core.ParserElement]*) → Tuple[pyparsing.core.ParserElement, pyparsing.core.ParserElement]

Deprecated - use *make_xml_tags*

pyparsing.**matchOnlyAtCol**(*n*)

Deprecated - use *match_only_at_col*

pyparsing.**matchPreviousExpr**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

Deprecated - use *match_previous_expr*

pyparsing.**matchPreviousLiteral**(*expr: pyparsing.core.ParserElement*) → pyparsing.core.ParserElement

Deprecated - use *match_previous_literal*

pyparsing.**nestedExpr**(*opener: Union[str, pyparsing.core.ParserElement] = '(', closer: Union[str, pyparsing.core.ParserElement] = ')', content: Optional[pyparsing.core.ParserElement] = None, ignore_expr: pyparsing.core.ParserElement = quoted string using single or double quotes, *, ignoreExpr: pyparsing.core.ParserElement = quoted string using single or double quotes*) → pyparsing.core.ParserElement

Deprecated - use *nested_expr*

pyparsing.**nullDebugAction**(**args*)

Deprecated - use *null_debug_action*

pyparsing.**oneOf**(*strs: Union[Iterable[str], str], caseless: bool = False, use_regex: bool = True, as_keyword: bool = False, *, useRegex: bool = True, asKeyword: bool = False*) → pyparsing.core.ParserElement

Deprecated - use *one_of*

pyparsing.**opAssoc**

alias of pyparsing.helpers.OpAssoc

pyparsing.**originalTextFor**(*expr: pyparsing.core.ParserElement, as_string: bool = True, *, asString: bool = True*) → pyparsing.core.ParserElement

Deprecated - use *original_text_for*

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

Deprecated - use *remove_quotes*

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

Deprecated - use *replace_html_entity*

pyparsing.**replaceWith**(*repl_str*)

Deprecated - use *replace_with*

pyparsing.**tokenMap**(*func*, **args*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

Deprecated - use *token_map*

pyparsing.**traceParseAction**(*f: Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]*) → Union[Callable[[], Any], Callable[[pyparsing.results.ParseResults], Any], Callable[[int, pyparsing.results.ParseResults], Any], Callable[[str, int, pyparsing.results.ParseResults], Any]]

Deprecated - use *trace_parse_action*

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

Deprecated - use *with_attribute*

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

Deprecated - use *with_class*

pyparsing.**common**

alias of pyparsing.common.pyparsing_common

pyparsing.**unicode**

alias of pyparsing.unicode.pyparsing_unicode

pyparsing.**testing**

alias of pyparsing.testing.pyparsing_test

CHAPTER 4

Contributor Covenant Code of Conduct

4.1 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.

4.2 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

4.3 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.

4.4 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.

4.5 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.

4.6 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 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z