parso documentation

Release 0.8.3

parso contributors

Nov 30, 2021

Contents

1 Docs **3**
- 1.1 Installation and Configuration . . . 3
- 1.2 Usage . . . 3
- 1.3 Parser Tree . . . 6
- 1.4 Development . . . 17

2 Resources **19**

Python Module Index **21**

Index **23**

Release v0.8.3. (*Installation*)

Parso is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your python file.

Parso has been battle-tested by jedi. It was pulled out of jedi to be useful for other projects as well.

Parso consists of a small API to parse Python and analyse the syntax tree.

A simple example:

```
>>> import parso
>>> module = parso.parse('hello + 1', version="3.9")
>>> expr = module.children[0]
>>> expr
PythonNode(arith_expr, [<Name: hello@1,0>, <Operator: +>, <Number: 1>])
>>> print(expr.get_code())
hello + 1
>>> name = expr.children[0]
>>> name
<Name: hello@1,0>
>>> name.end_pos
(1, 5)
>>> expr.end_pos
(1, 9)
```

To list multiple issues:

```
>>> grammar = parso.load_grammar()
>>> module = grammar.parse('foo +\nbar\ncontinue')
>>> error1, error2 = grammar.iter_errors(module)
>>> error1.message
'SyntaxError: invalid syntax'
>>> error2.message
"SyntaxError: 'continue' not properly in loop"
```

CHAPTER 1

Docs

1.1 Installation and Configuration

1.1.1 The preferred way (pip)

On any system you can install *parso* directly from the Python package index using pip:

```
sudo pip install parso
```

1.1.2 From git

If you want to install the current development version (master branch):

```
sudo pip install -e git://github.com/davidhalter/parso.git#egg=parso
```

1.1.3 Manual installation from a downloaded package (not recommended)

If you prefer not to use an automated package installer, you can download a current copy of *parso* and install it manually.

To install it, navigate to the directory containing *setup.py* on your console and type:

```
sudo python setup.py install
```

1.2 Usage

parso works around grammars. You can simply create Python grammars by calling *parso.load_grammar()*. Grammars (with a custom tokenizer and custom parser trees) can also be created by directly instantiating *parso.Grammar()*. More information about the resulting objects can be found in the *parser tree documentation*.

The simplest way of using parso is without even loading a grammar (*parso.parse()*):

```
>>> import parso
>>> parso.parse('foo + bar')
<Module: @1-1>
```

1.2.1 Loading a Grammar

Typically if you want to work with one specific Python version, use:

parso.**load_grammar**(*, *version: str = None*, *path: str = None*)

Loads a *parso.Grammar*. The default version is the current Python version.

Parameters

- **version** (*str*) – A python version string, e.g. `version='3.8'`.
- **path** (*str*) – A path to a grammar file

1.2.2 Grammar methods

You will get back a grammar object that you can use to parse code and find issues in it:

class parso.**Grammar**(*text: str*, *, *tokenizer*, *parser=<class 'parso.parser.BaseParser'>*, *diff_parser=None*)

parso.load_grammar() returns instances of this class.

Creating custom none-python grammars by calling this is not supported, yet.

Parameters **text** – A BNF representation of your grammar.

parse(*code: Union[str, bytes] = None*, *, *error_recovery=True*, *path: Union[os.PathLike, str] = None*, *start_symbol: str = None*, *cache=False*, *diff_cache=False*, *cache_path: Union[os.PathLike, str] = None*, *file_io: parso.file_io.FileIO = None*) → _NodeT

If you want to parse a Python file you want to start here, most likely.

If you need finer grained control over the parsed instance, there will be other ways to access it.

Parameters

- **code** (*str*) – A unicode or bytes string. When it's not possible to decode bytes to a string, returns a `UnicodeDecodeError`.
- **error_recovery** (*bool*) – If enabled, any code will be returned. If it is invalid, it will be returned as an error node. If disabled, you will get a ParseError when encountering syntax errors in your code.
- **start_symbol** (*str*) – The grammar rule (nonterminal) that you want to parse. Only allowed to be used when error_recovery is False.
- **path** (*str*) – The path to the file you want to open. Only needed for caching.
- **cache** (*bool*) – Keeps a copy of the parser tree in RAM and on disk if a path is given. Returns the cached trees if the corresponding files on disk have not changed. Note that this stores pickle files on your file system (e.g. for Linux in `~/.cache/parso/`).
- **diff_cache** (*bool*) – Diffs the cached python module against the new code and tries to parse only the parts that have changed. Returns the same (changed) module that is found in cache. Using this option requires you to not do anything anymore with the cached modules under that path, because the contents of it might change. This option is still somewhat experimental. If you want stability, please don't use it.

- **cache_path** (*bool*) – If given saves the parso cache in this directory. If not given, defaults to the default cache places on each platform.

Returns A subclass of *parso.tree.NodeOrLeaf*. Typically a *parso.python.tree.Module*.

iter_errors(*node*)
Given a *parso.tree.NodeOrLeaf* returns a generator of *parso.normalizer.Issue* objects. For Python this is a list of syntax/indentation errors.

refactor(*base_node*, *node_to_str_map*)

1.2.3 Error Retrieval

parso is able to find multiple errors in your source code. Iterating through those errors yields the following instances:

class parso.normalizer.**Issue**(*node*, *code*, *message*)

code = None
An integer code that stands for the type of error.

message = None
A message (string) for the issue.

start_pos = None
The start position position of the error as a tuple (line, column). As always in *parso* the first line is 1 and the first column 0.

1.2.4 Utility

parso also offers some utility functions that can be really useful:

parso.**parse**(*code=None*, ***kwargs*)
A utility function to avoid loading grammars. Params are documented in *parso.Grammar.parse()*.

Parameters **version** (*str*) – The version used by *parso.load_grammar()*.

parso.**split_lines**(*string: str*, *keepends: bool = False*) → Sequence[str]
Intended for Python code. In contrast to Python's `str.splitlines()`, looks at form feeds and other special characters as normal text. Just splits `\n` and `\r\n`. Also different: Returns `[""]` for an empty string input.

In Python 2.7 form feeds are used as normal characters when using str.splitlines. However in Python 3 somewhere there was a decision to split also on form feeds.

parso.**python_bytes_to_unicode**(*source: Union[str, bytes]*, *encoding: str = 'utf-8'*, *errors: str = 'strict'*) → str
Checks for unicode BOMs and PEP 263 encoding declarations. Then returns a unicode object like in `bytes.decode()`.

Parameters

- **encoding** – See `bytes.decode()` documentation.
- **errors** – See `bytes.decode()` documentation. `errors` can be `'strict'`, `'replace'` or `'ignore'`.

1.2.5 Used By

- jedi (which is used by IPython and a lot of editor plugins).
- mutmut (mutation tester)

1.3 Parser Tree

The parser tree is returned by calling *parso.Grammar.parse()*.

Note: Note that parso positions are always 1 based for lines and zero based for columns. This means the first position in a file is (1, 0).

1.3.1 Parser Tree Base Classes

Generally there are two types of classes you will deal with: *parso.tree.Leaf* and *parso.tree.BaseNode*.

class parso.tree.**BaseNode**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.tree.NodeOrLeaf*

The super class for all nodes. A node has children, a type and possibly a parent node.

children
A list of *NodeOrLeaf* child nodes.

start_pos
Returns the starting position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

get_start_pos_of_prefix()
Returns the start_pos of the prefix. This means basically it returns the end_pos of the last prefix. The *get_start_pos_of_prefix()* of the prefix + in *2 + 1* would be *(1, 1)*, while the start_pos is *(1, 2)*.

Return tuple of int (line, column)

end_pos
Returns the end position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

get_code(*include_prefix=True*)
Returns the code that was the input for the parser for this node.

Parameters **include_prefix** – Removes the prefix (whitespace and comments) of e.g. a statement.

get_leaf_for_position(*position*, *include_prefixes=False*)
Get the *parso.tree.Leaf* at `position`

Parameters

- **position** (*tuple*) – A position tuple, row, column. Rows start from 1
- **include_prefixes** (*bool*) – If `False`, `None` will be returned if `position` falls on whitespace or comments before a leaf

Returns *parso.tree.Leaf* at `position`, or `None`

get_first_leaf()
Returns the first leaf of a node or itself if this is a leaf.

get_last_leaf()
Returns the last leaf of a node or itself if this is a leaf.

class parso.tree.**Leaf**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.tree.NodeOrLeaf*

Leafs are basically tokens with a better API. Leafs exactly know where they were defined and what text preceeds them.

value
str() The value of the current token.

prefix
str() Typically a mixture of whitespace and comments. Stuff that is syntactically irrelevant for the syntax tree.

start_pos
Returns the starting position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

get_start_pos_of_prefix()
Returns the start_pos of the prefix. This means basically it returns the end_pos of the last prefix. The *get_start_pos_of_prefix()* of the prefix + in *2 + 1* would be *(1, 1)*, while the start_pos is *(1, 2)*.

Return tuple of int (line, column)

get_first_leaf()
Returns the first leaf of a node or itself if this is a leaf.

get_last_leaf()
Returns the last leaf of a node or itself if this is a leaf.

get_code(*include_prefix=True*)
Returns the code that was the input for the parser for this node.

Parameters **include_prefix** – Removes the prefix (whitespace and comments) of e.g. a statement.

end_pos
Returns the end position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

All nodes and leaves have these methods/properties:

class parso.tree.**NodeOrLeaf**
Bases: object

The base class for nodes and leaves.

type = None
The type is a string that typically matches the types of the grammar file.

parent
The parent *BaseNode* of this node or leaf. None if this is the root node.

get_root_node()
Returns the root node of a parser tree. The returned node doesn't have a parent node like all the other nodes/leaves.

get_next_sibling()
Returns the node immediately following this node in this parent's children list. If this node does not have a next sibling, it is None

get_previous_sibling()
Returns the node immediately preceding this node in this parent's children list. If this node does not have a previous sibling, it is None.

get_previous_leaf()
Returns the previous leaf in the parser tree. Returns *None* if this is the first element in the parser tree.

get_next_leaf()
Returns the next leaf in the parser tree. Returns None if this is the last element in the parser tree.

start_pos
Returns the starting position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

end_pos
Returns the end position of the prefix as a tuple, e.g. *(3, 4)*.

Return tuple of int (line, column)

get_start_pos_of_prefix()
Returns the start_pos of the prefix. This means basically it returns the end_pos of the last prefix. The *get_start_pos_of_prefix()* of the prefix + in *2 + 1* would be *(1, 1)*, while the start_pos is *(1, 2)*.

Return tuple of int (line, column)

get_first_leaf()
Returns the first leaf of a node or itself if this is a leaf.

get_last_leaf()
Returns the last leaf of a node or itself if this is a leaf.

get_code(*include_prefix=True*)
Returns the code that was the input for the parser for this node.

Parameters **include_prefix** – Removes the prefix (whitespace and comments) of e.g. a statement.

search_ancestor(**node_types*) → Optional[parso.tree.BaseNode]
Recursively looks at the parents of this node or leaf and returns the first found node that matches `node_types`. Returns `None` if no matching node is found.

Parameters **node_types** – type names that are searched for.

dump(*, *indent: Union[str, int, None] = 4*) → str
Returns a formatted dump of the parser tree rooted at this node or leaf. This is mainly useful for debugging purposes.

The `indent` parameter is interpreted in a similar way as `ast.dump()`. If `indent` is a non-negative integer or string, then the tree will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` selects the single line representation. Using a positive integer indent indents that many spaces per level. If `indent` is a string (such as `"\t"`), that string is used to indent each level.

Parameters **indent** – Indentation style as described above. The default indentation is 4 spaces, which yields a pretty-printed dump.

```
>>> import parso
>>> print(parso.parse("lambda x, y: x + y").dump())
Module([
    Lambda([
        Keyword('lambda', (1, 0)),
        Param([
            Name('x', (1, 7), prefix=' '),
            Operator(',', (1, 8)),
        ]),
        Param([
            Name('y', (1, 10), prefix=' '),
        ]),
        Operator(':', (1, 11)),
        PythonNode('arith_expr', [
            Name('x', (1, 13), prefix=' '),
            Operator('+', (1, 15), prefix=' '),
            Name('y', (1, 17), prefix=' '),
        ]),
    ]),
    EndMarker('', (1, 18)),
])
```

1.3.2 Python Parser Tree

This is the syntax tree for Python 3 syntaxes. The classes represent syntax elements like functions and imports.

All of the nodes can be traced back to the Python grammar file. If you want to know how a tree is structured, just analyse that file (for each Python version it's a bit different).

There's a lot of logic here that makes it easier for Jedi (and other libraries) to deal with a Python syntax tree.

By using *parso.tree.NodeOrLeaf.get_code()* on a module, you can get back the 1-to-1 representation of the input given to the parser. This is important if you want to refactor a parser tree.

```
>>> from parso import parse
>>> parser = parse('import os')
>>> module = parser.get_root_node()
>>> module
<Module: @1-1>
```

Any subclasses of *Scope*, including *Module* has an attribute *iter_imports*:

```
>>> list(module.iter_imports())
[<ImportName: import os@1,0>]
```

Changes to the Python Grammar

A few things have changed when looking at Python grammar files:

- *Param* does not exist in Python grammar files. It is essentially a part of a `parameters` node. *parso* splits it up to make it easier to analyse parameters. However this just makes it easier to deal with the syntax tree, it doesn't actually change the valid syntax.
- A few nodes like *lambdef* and *lambdef_nocond* have been merged in the syntax tree to make it easier to do deal with them.

Parser Tree Classes

class parso.python.tree.**DocstringMixin**
Bases: object

get_doc_node()
Returns the string leaf of a docstring. e.g. `r'''foo'''`.

class parso.python.tree.**PythonMixin**
Bases: object

Some Python specific utilities.

get_name_of_position(*position*)
Given a (line, column) tuple, returns a *Name* or None if there is no name at that position.

class parso.python.tree.**PythonLeaf**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonMixin*, *parso.tree.Leaf*

get_start_pos_of_prefix()
Basically calls *parso.tree.NodeOrLeaf.get_start_pos_of_prefix()*.

class parso.python.tree.**PythonBaseNode**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonMixin*, *parso.tree.BaseNode*

class parso.python.tree.**PythonNode**(*type*, *children*)
Bases: *parso.python.tree.PythonMixin*, parso.tree.Node

class parso.python.tree.**PythonErrorNode**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonMixin*, parso.tree.ErrorNode

class parso.python.tree.**PythonErrorLeaf**(*token_type*, *value*, *start_pos*, *prefix=''*)
Bases: parso.tree.ErrorLeaf, *parso.python.tree.PythonLeaf*

class parso.python.tree.**EndMarker**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: parso.python.tree._LeafWithoutNewlines

type = 'endmarker'

class parso.python.tree.**Newline**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonLeaf*

Contains NEWLINE and ENDMARKER tokens.

type = 'newline'

class parso.python.tree.**Name**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: parso.python.tree._LeafWithoutNewlines

A string. Sometimes it is important to know if the string belongs to a name or not.

type = 'name'

is_definition(*include_setitem=False*)
Returns True if the name is being defined.

get_definition(*import_name_always=False*, *include_setitem=False*)
Returns None if there's no definition for a name.

Parameters **import_name_always** – Specifies if an import name is always a definition. Normally foo in *from foo import bar* is not a definition.

class parso.python.tree.**Literal**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonLeaf*

class parso.python.tree.**Number**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.Literal*

type = 'number'

class parso.python.tree.**String**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.Literal*

type = 'string'

string_prefix

class parso.python.tree.**FStringString**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonLeaf*

f-strings contain f-string expressions and normal python strings. These are the string parts of f-strings.

type = 'fstring_string'

class parso.python.tree.**FStringStart**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonLeaf*

f-strings contain f-string expressions and normal python strings. These are the string parts of f-strings.

type = 'fstring_start'

class parso.python.tree.**FStringEnd**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: *parso.python.tree.PythonLeaf*

f-strings contain f-string expressions and normal python strings. These are the string parts of f-strings.

type = 'fstring_end'

class parso.python.tree.**Operator**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: parso.python.tree._LeafWithoutNewlines, parso.python.tree._StringComparisonMixin

type = 'operator'

class parso.python.tree.**Keyword**(*value: str, start_pos: Tuple[int, int], prefix: str = ''*)
Bases: parso.python.tree._LeafWithoutNewlines, parso.python.tree._StringComparisonMixin

type = 'keyword'

class parso.python.tree.**Scope**(*children*)
Bases: *parso.python.tree.PythonBaseNode*, *parso.python.tree.DocstringMixin*

Super class for the parser tree, which represents the state of a python text file. A Scope is either a function, class or lambda.

iter_funcdefs()
Returns a generator of *funcdef* nodes.

iter_classdefs()
Returns a generator of *classdef* nodes.

iter_imports()
Returns a generator of *import_name* and *import_from* nodes.

get_suite()
Returns the part that is executed by the function.

class parso.python.tree.**Module**(*children*)
Bases: *parso.python.tree.Scope*

The top scope, which is always a module. Depending on the underlying parser this may be a full module or just a part of a module.

type = 'file_input'

get_used_names()
Returns all the *Name* leafs that exist in this module. This includes both definitions and references of names.

class parso.python.tree.**Decorator**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

type = 'decorator'

class parso.python.tree.**ClassOrFunc**(*children*)
Bases: *parso.python.tree.Scope*

name
Returns the *Name* leaf that defines the function or class name.

get_decorators()

Return type list of *Decorator*

class parso.python.tree.**Class**(*children*)
Bases: *parso.python.tree.ClassOrFunc*

Used to store the parsed contents of a python class.

type = 'classdef'

get_super_arglist()
Returns the *arglist* node that defines the super classes. It returns None if there are no arguments.

class parso.python.tree.**Function**(*children*)
Bases: *parso.python.tree.ClassOrFunc*

Used to store the parsed contents of a python function.

Children:

```
0. <Keyword: def>
1. <Name>
2. parameter list (including open-paren and close-paren <Operator>s)
3. or 5. <Operator: :>
4. or 6. Node() representing function body
3. -> (if annotation is also present)
4. annotation (if present)
```

type = 'funcdef'

get_params()
Returns a list of *Param()*.

name
Returns the *Name* leaf that defines the function or class name.

iter_yield_exprs()
Returns a generator of *yield_expr*.

iter_return_stmts()
Returns a generator of *return_stmt*.

iter_raise_stmts()
Returns a generator of *raise_stmt*. Includes raise statements inside try-except blocks

is_generator()

Return bool Checks if a function is a generator or not.

annotation
Returns the test node after -> or *None* if there is no annotation.

class parso.python.tree.**Lambda**(*children*)
Bases: *parso.python.tree.Function*

Lambdas are basically trimmed functions, so give it the same interface.

Children:

```
 0. <Keyword: lambda>
 *. <Param x> for each argument x
-2. <Operator: :>
-1. Node() representing body
```

type = 'lambdef'

name
Raises an AttributeError. Lambdas don't have a defined name.

annotation
Returns *None*, lambdas don't have annotations.

class parso.python.tree.**Flow**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

class parso.python.tree.**IfStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Flow*

type = 'if_stmt'

get_test_nodes()
E.g. returns all the *test* nodes that are named as x, below:

if x: pass

elif x: pass

get_corresponding_test_node(*node*)
Searches for the branch in which the node is and returns the corresponding test node (see function above). However if the node is in the test node itself and not in the suite return None.

is_node_after_else(*node*)
Checks if a node is defined after *else*.

class parso.python.tree.**WhileStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Flow*

type = 'while_stmt'

class parso.python.tree.**ForStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Flow*

type = 'for_stmt'

get_testlist()
Returns the input node `y` from: `for x in y:`.

get_defined_names(*include_setitem=False*)

class parso.python.tree.**TryStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Flow*

type = 'try_stmt'

get_except_clause_tests()
Returns the `test` nodes found in `except_clause` nodes. Returns `[None]` for except clauses without an exception given.

class parso.python.tree.**WithStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Flow*

type = 'with_stmt'

get_defined_names(*include_setitem=False*)
Returns the a list of *Name* that the with statement defines. The defined names are set after *as*.

get_test_node_from_name(*name*)

class parso.python.tree.**Import**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

get_path_for_name(*name*)
The path is the list of names that leads to the searched name.

Return list of Name

is_nested()

is_star_import()

class parso.python.tree.**ImportFrom**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Import*

type = 'import_from'

get_defined_names(*include_setitem=False*)
Returns the a list of *Name* that the import defines. The defined names are set after *import* or in case an alias - *as* - is present that name is returned.

get_from_names()

level
The level parameter of `__import__`.

get_paths()
The import paths defined in an import statement. Typically an array like this: `[<Name: datetime>, <Name: date>]`.

Return list of list of Name

class parso.python.tree.**ImportName**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.Import*

For `import_name` nodes. Covers normal imports without `from`.

type = 'import_name'

get_defined_names(*include_setitem=False*)
Returns the a list of *Name* that the import defines. The defined names is always the first name after *import* or in case an alias - *as* - is present that name is returned.

level
The level parameter of `__import__`.

get_paths()

is_nested()
This checks for the special case of nested imports, without aliases and from statement:

```
import foo.bar
```

class parso.python.tree.**KeywordStatement**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

For the following statements: *assert*, *del*, *global*, *nonlocal*, *raise*, *return*, *yield*.

pass, *continue* and *break* are not in there, because they are just simple keywords and the parser reduces it to a keyword.

type
Keyword statements start with the keyword and end with *_stmt*. You can crosscheck this with the Python grammar.

keyword

get_defined_names(*include_setitem=False*)

class parso.python.tree.**AssertStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.KeywordStatement*

assertion

class parso.python.tree.**GlobalStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.KeywordStatement*

get_global_names()

class parso.python.tree.**ReturnStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.KeywordStatement*

class parso.python.tree.**YieldExpr**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

type = 'yield_expr'

class parso.python.tree.**ExprStmt**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*, *parso.python.tree.DocstringMixin*

type = 'expr_stmt'

get_defined_names(*include_setitem=False*)
Returns a list of *Name* defined before the = sign.

get_rhs()
Returns the right-hand-side of the equals.

yield_operators()
Returns a generator of +=, =, etc. or None if there is no operation.

class parso.python.tree.**NamedExpr**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

type = 'namedexpr_test'

get_defined_names(*include_setitem=False*)

class parso.python.tree.**Param**(*children*, *parent=None*)
Bases: *parso.python.tree.PythonBaseNode*

It's a helper class that makes business logic with params much easier. The Python grammar defines no `param` node. It defines it in a different way that is not really suited to working with parameters.

type = 'param'

star_count
Is *0* in case of *foo*, *1* in case of **foo* or *2* in case of ***foo*.

default
The default is the test node that appears after the =. Is *None* in case no default is present.

annotation
The default is the test node that appears after *:*. Is *None* in case no annotation is present.

name
The *Name* leaf of the param.

get_defined_names(*include_setitem=False*)

position_index
Property for the positional index of a paramter.

get_parent_function()
Returns the function/lambda of a parameter.

get_code(*include_prefix=True*, *include_comma=True*)
Like all the other get_code functions, but includes the param *include_comma*.

Parameters **bool** (*include_comma*) – If enabled includes the comma in the string output.

class parso.python.tree.**SyncCompFor**(*children: List[parso.tree.NodeOrLeaf]*)
Bases: *parso.python.tree.PythonBaseNode*

type = 'sync_comp_for'

get_defined_names(*include_setitem=False*)
Returns the a list of *Name* that the comprehension defines.

parso.python.tree.**CompFor**
alias of *parso.python.tree.SyncCompFor*

class parso.python.tree.**UsedNamesMapping**(*dct*)
Bases: collections.abc.Mapping

This class exists for the sole purpose of creating an immutable dict.

1.3.3 Utility

parso.tree.**search_ancestor**(*node: parso.tree.NodeOrLeaf*, **node_types*) → Optional[parso.tree.BaseNode]
Recursively looks at the parents of a node and returns the first found node that matches node_types. Returns None if no matching node is found.

This function is deprecated, use *NodeOrLeaf.search_ancestor()* instead.

Parameters

- **node** – The ancestors of this node will be checked.
- **node_types** – type names that are searched for.

1.4 Development

If you want to contribute anything to *parso*, just open an issue or pull request to discuss it. We welcome changes! Please check the `CONTRIBUTING.md` file in the repository, first.

1.4.1 Deprecations Process

The deprecation process is as follows:

1. A deprecation is announced in the next major/minor release.
2. We wait either at least a year & at least two minor releases until we remove the deprecated functionality.

1.4.2 Testing

The test suite depends on `pytest`:

```
pip install pytest
```

To run the tests use the following:

```
pytest
```

If you want to test only a specific Python version (e.g. Python 3.9), it's as easy as:

```
python3.9 -m pytest
```

Tests are also run automatically on GitHub Actions.

CHAPTER 2

Resources

- Source Code on Github
- GitHub Actions Testing
- Python Package Index

Python Module Index

p

parso, 1
parso.python.tree, 9

Index

A

annotation (*parso.python.tree.Function attribute*), 13
annotation (*parso.python.tree.Lambda attribute*), 13
annotation (*parso.python.tree.Param attribute*), 16
assertion (*parso.python.tree.AssertStmt attribute*), 15
AssertStmt (*class in parso.python.tree*), 15

B

BaseNode (*class in parso.tree*), 6

C

children (*parso.tree.BaseNode attribute*), 6
Class (*class in parso.python.tree*), 12
ClassOrFunc (*class in parso.python.tree*), 12
code (*parso.normalizer.Issue attribute*), 5
CompFor (*in module parso.python.tree*), 16

D

Decorator (*class in parso.python.tree*), 12
default (*parso.python.tree.Param attribute*), 16
DocstringMixin (*class in parso.python.tree*), 10
dump() (*parso.tree.NodeOrLeaf method*), 8

E

end_pos (*parso.tree.BaseNode attribute*), 6
end_pos (*parso.tree.Leaf attribute*), 7
end_pos (*parso.tree.NodeOrLeaf attribute*), 8
EndMarker (*class in parso.python.tree*), 10
ExprStmt (*class in parso.python.tree*), 15

F

Flow (*class in parso.python.tree*), 13
ForStmt (*class in parso.python.tree*), 13
FStringEnd (*class in parso.python.tree*), 11
FStringStart (*class in parso.python.tree*), 11
FStringString (*class in parso.python.tree*), 11
Function (*class in parso.python.tree*), 12

G

get_code() (*parso.python.tree.Param method*), 16
get_code() (*parso.tree.BaseNode method*), 6
get_code() (*parso.tree.Leaf method*), 7
get_code() (*parso.tree.NodeOrLeaf method*), 8
get_corresponding_test_node() (*parso.python.tree.IfStmt method*), 13
get_decorators() (*parso.python.tree.ClassOrFunc method*), 12
get_defined_names() (*parso.python.tree.ExprStmt method*), 15
get_defined_names() (*parso.python.tree.ForStmt method*), 13
get_defined_names() (*parso.python.tree.ImportFrom method*), 14
get_defined_names() (*parso.python.tree.ImportName method*), 14
get_defined_names() (*parso.python.tree.KeywordStatement method*), 15
get_defined_names() (*parso.python.tree.NamedExpr method*), 15
get_defined_names() (*parso.python.tree.Param method*), 16
get_defined_names() (*parso.python.tree.SyncCompFor method*), 16
get_defined_names() (*parso.python.tree.WithStmt method*), 14
get_definition() (*parso.python.tree.Name method*), 10
get_doc_node() (*parso.python.tree.DocstringMixin method*), 10
get_except_clause_tests() (*parso.python.tree.TryStmt method*), 14
get_first_leaf() (*parso.tree.BaseNode method*), 6

get_first_leaf() (*parso.tree.Leaf method*), 7
get_first_leaf() (*parso.tree.NodeOrLeaf method*), 8
get_from_names() (*parso.python.tree.ImportFrom method*), 14
get_global_names() (*parso.python.tree.GlobalStmt method*), 15
get_last_leaf() (*parso.tree.BaseNode method*), 7
get_last_leaf() (*parso.tree.Leaf method*), 7
get_last_leaf() (*parso.tree.NodeOrLeaf method*), 8
get_leaf_for_position() (*parso.tree.BaseNode method*), 6
get_name_of_position() (*parso.python.tree.PythonMixin method*), 10
get_next_leaf() (*parso.tree.NodeOrLeaf method*), 8
get_next_sibling() (*parso.tree.NodeOrLeaf method*), 7
get_params() (*parso.python.tree.Function method*), 12
get_parent_function() (*parso.python.tree.Param method*), 16
get_path_for_name() (*parso.python.tree.Import method*), 14
get_paths() (*parso.python.tree.ImportFrom method*), 14
get_paths() (*parso.python.tree.ImportName method*), 14
get_previous_leaf() (*parso.tree.NodeOrLeaf method*), 8
get_previous_sibling() (*parso.tree.NodeOrLeaf method*), 8
get_rhs() (*parso.python.tree.ExprStmt method*), 15
get_root_node() (*parso.tree.NodeOrLeaf method*), 7
get_start_pos_of_prefix() (*parso.python.tree.PythonLeaf method*), 10
get_start_pos_of_prefix() (*parso.tree.BaseNode method*), 6
get_start_pos_of_prefix() (*parso.tree.Leaf method*), 7
get_start_pos_of_prefix() (*parso.tree.NodeOrLeaf method*), 8
get_suite() (*parso.python.tree.Scope method*), 11
get_super_arglist() (*parso.python.tree.Class method*), 12
get_test_node_from_name() (*parso.python.tree.WithStmt method*), 14
get_test_nodes() (*parso.python.tree.IfStmt method*), 13
get_testlist() (*parso.python.tree.ForStmt method*), 13
get_used_names() (*parso.python.tree.Module method*), 12
GlobalStmt (*class in parso.python.tree*), 15
Grammar (*class in parso*), 4

I

IfStmt (*class in parso.python.tree*), 13
Import (*class in parso.python.tree*), 14
ImportFrom (*class in parso.python.tree*), 14
ImportName (*class in parso.python.tree*), 14
is_definition() (*parso.python.tree.Name method*), 10
is_generator() (*parso.python.tree.Function method*), 12
is_nested() (*parso.python.tree.Import method*), 14
is_nested() (*parso.python.tree.ImportName method*), 14
is_node_after_else() (*parso.python.tree.IfStmt method*), 13
is_star_import() (*parso.python.tree.Import method*), 14
Issue (*class in parso.normalizer*), 5
iter_classdefs() (*parso.python.tree.Scope method*), 11
iter_errors() (*parso.Grammar method*), 5
iter_funcdefs() (*parso.python.tree.Scope method*), 11
iter_imports() (*parso.python.tree.Scope method*), 11
iter_raise_stmts() (*parso.python.tree.Function method*), 12
iter_return_stmts() (*parso.python.tree.Function method*), 12
iter_yield_exprs() (*parso.python.tree.Function method*), 12

K

Keyword (*class in parso.python.tree*), 11
keyword (*parso.python.tree.KeywordStatement attribute*), 15
KeywordStatement (*class in parso.python.tree*), 15

L

Lambda (*class in parso.python.tree*), 13
Leaf (*class in parso.tree*), 7
level (*parso.python.tree.ImportFrom attribute*), 14
level (*parso.python.tree.ImportName attribute*), 14
Literal (*class in parso.python.tree*), 10
load_grammar() (*in module parso*), 4

M

message (*parso.normalizer.Issue attribute*), 5

Module (*class in parso.python.tree*), 11

N

Name (*class in parso.python.tree*), 10
name (*parso.python.tree.ClassOrFunc attribute*), 12
name (*parso.python.tree.Function attribute*), 12
name (*parso.python.tree.Lambda attribute*), 13
name (*parso.python.tree.Param attribute*), 16
NamedExpr (*class in parso.python.tree*), 15
Newline (*class in parso.python.tree*), 10
NodeOrLeaf (*class in parso.tree*), 7
Number (*class in parso.python.tree*), 10

O

Operator (*class in parso.python.tree*), 11

P

Param (*class in parso.python.tree*), 15
parent (*parso.tree.NodeOrLeaf attribute*), 7
parse() (*in module parso*), 5
parse() (*parso.Grammar method*), 4
parso (*module*), 1
parso.python.tree (*module*), 9
position_index (*parso.python.tree.Param attribute*), 16
prefix (*parso.tree.Leaf attribute*), 7
python_bytes_to_unicode() (*in module parso*), 5
PythonBaseNode (*class in parso.python.tree*), 10
PythonErrorLeaf (*class in parso.python.tree*), 10
PythonErrorNode (*class in parso.python.tree*), 10
PythonLeaf (*class in parso.python.tree*), 10
PythonMixin (*class in parso.python.tree*), 10
PythonNode (*class in parso.python.tree*), 10

R

refactor() (*parso.Grammar method*), 5
ReturnStmt (*class in parso.python.tree*), 15

S

Scope (*class in parso.python.tree*), 11
search_ancestor() (*in module parso.tree*), 16
search_ancestor() (*parso.tree.NodeOrLeaf method*), 8
split_lines() (*in module parso*), 5
star_count (*parso.python.tree.Param attribute*), 16
start_pos (*parso.normalizer.Issue attribute*), 5
start_pos (*parso.tree.BaseNode attribute*), 6
start_pos (*parso.tree.Leaf attribute*), 7
start_pos (*parso.tree.NodeOrLeaf attribute*), 8
String (*class in parso.python.tree*), 11
string_prefix (*parso.python.tree.String attribute*), 11
SyncCompFor (*class in parso.python.tree*), 16

T

TryStmt (*class in parso.python.tree*), 13
type (*parso.python.tree.Class attribute*), 12
type (*parso.python.tree.Decorator attribute*), 12
type (*parso.python.tree.EndMarker attribute*), 10
type (*parso.python.tree.ExprStmt attribute*), 15
type (*parso.python.tree.ForStmt attribute*), 13
type (*parso.python.tree.FStringEnd attribute*), 11
type (*parso.python.tree.FStringStart attribute*), 11
type (*parso.python.tree.FStringString attribute*), 11
type (*parso.python.tree.Function attribute*), 12
type (*parso.python.tree.IfStmt attribute*), 13
type (*parso.python.tree.ImportFrom attribute*), 14
type (*parso.python.tree.ImportName attribute*), 14
type (*parso.python.tree.Keyword attribute*), 11
type (*parso.python.tree.KeywordStatement attribute*), 15
type (*parso.python.tree.Lambda attribute*), 13
type (*parso.python.tree.Module attribute*), 12
type (*parso.python.tree.Name attribute*), 10
type (*parso.python.tree.NamedExpr attribute*), 15
type (*parso.python.tree.Newline attribute*), 10
type (*parso.python.tree.Number attribute*), 11
type (*parso.python.tree.Operator attribute*), 11
type (*parso.python.tree.Param attribute*), 15
type (*parso.python.tree.String attribute*), 11
type (*parso.python.tree.SyncCompFor attribute*), 16
type (*parso.python.tree.TryStmt attribute*), 14
type (*parso.python.tree.WhileStmt attribute*), 13
type (*parso.python.tree.WithStmt attribute*), 14
type (*parso.python.tree.YieldExpr attribute*), 15
type (*parso.tree.NodeOrLeaf attribute*), 7

U

UsedNamesMapping (*class in parso.python.tree*), 16

V

value (*parso.tree.Leaf attribute*), 7

W

WhileStmt (*class in parso.python.tree*), 13
WithStmt (*class in parso.python.tree*), 14

Y

yield_operators() (*parso.python.tree.ExprStmt method*), 15
YieldExpr (*class in parso.python.tree*), 15