PyCep Documentation

Release 1.0

Daniel Lorch

Sep 24, 2018

Contents

PyCep (Python Inception) is a Python 2.7 interpreter written in Python 2.7, with the goal to be fully compatible with CPython 2.7.

PyCep was created as study project to teach myself about writing interpreters. PyCep's externally exported methods are modelled after the Python standard library, thus have identical method signatures and return the same data structures as those offered by Python itself, while being entirely written in Python.

CHAPTER 1

Try it out!

CHAPTER 2

Implementation Status

Module	Status	Comments
Tokenizer	0%	Forwarding calls to `tokenize.generate_tokens()`.
Parser	98%	Very good coverage (except handling of encodings)
Analyzer	14%	Analyzing a handful of example programs.
Interpreter	3%	Interpreting very few example programs.

CHAPTER 3

Design Goals

- External methods have the same signatures and produce the same results as those provided by the standard Python library
- Each of the interpreter's phases is built upon the previous phase, can be intercepted and studied independently of the other parts of the system

CHAPTER 4

Sources

Get the sources from https://github.com/dlorch/pycep/

CHAPTER 5

Contents

5.1 pycep Package

Sub-Modules:

5.1.1 pycep.tokenizer Module

pycep.tokenizer.**generate_tokens**(*readline*)

The tokenizer takes a file handle containing the source code as an input and returns a sequence of tokens.

```
>>> import pycep.tokenizer
>>> from StringIO import StringIO
>>> tokens = pycep.tokenizer.generate_tokens(StringIO('print "Hello, world!"').
↪readline)
>>> list(tokens)
[(1, 'print', (1, 0), (1, 5), 'print "Hello, world!"'), (3, '"Hello, world!"', (1,
↪ 6), (1, 19), 'print "Hello, world!"'), (0, '', (2, 0), (2, 0), '')]
```

See also:

- Python Language Reference: Lexical Analysis: https://docs.python.org/2/reference/lexical_analysis.html
- Python: Myths about Indentation: http://www.secnetix.de/olli/Python/block_indentation.hawk
- Python is not context free: http://trevorjim.com/python-is-not-context-free/

5.1.2 pycep.parser Module

pycep.parser.**suite**(*source*, *totuple=False*)

The parser takes a string containing the source code as an input and returns a parse tree.

```
>>> import pycep.parser
>>> st = pycep.parser.suite('print "Hello, world!"')
>>> st.totuple()
(257, (267, (268, (269, (272, (1, 'print'), (304, (305, (306, (307, (308, (310,␣
↪(311, (312, (313, (314, (315,     (316, (317, (318, (3, '"Hello, world!"
↪'))))))))))))))))), (4, ''))), (4, ''), (0, ''))
```

Formally, this is an LL(2) (Left-to-right, Leftmost derivation with two-token lookahead), recursive-descent parser.

Parameters

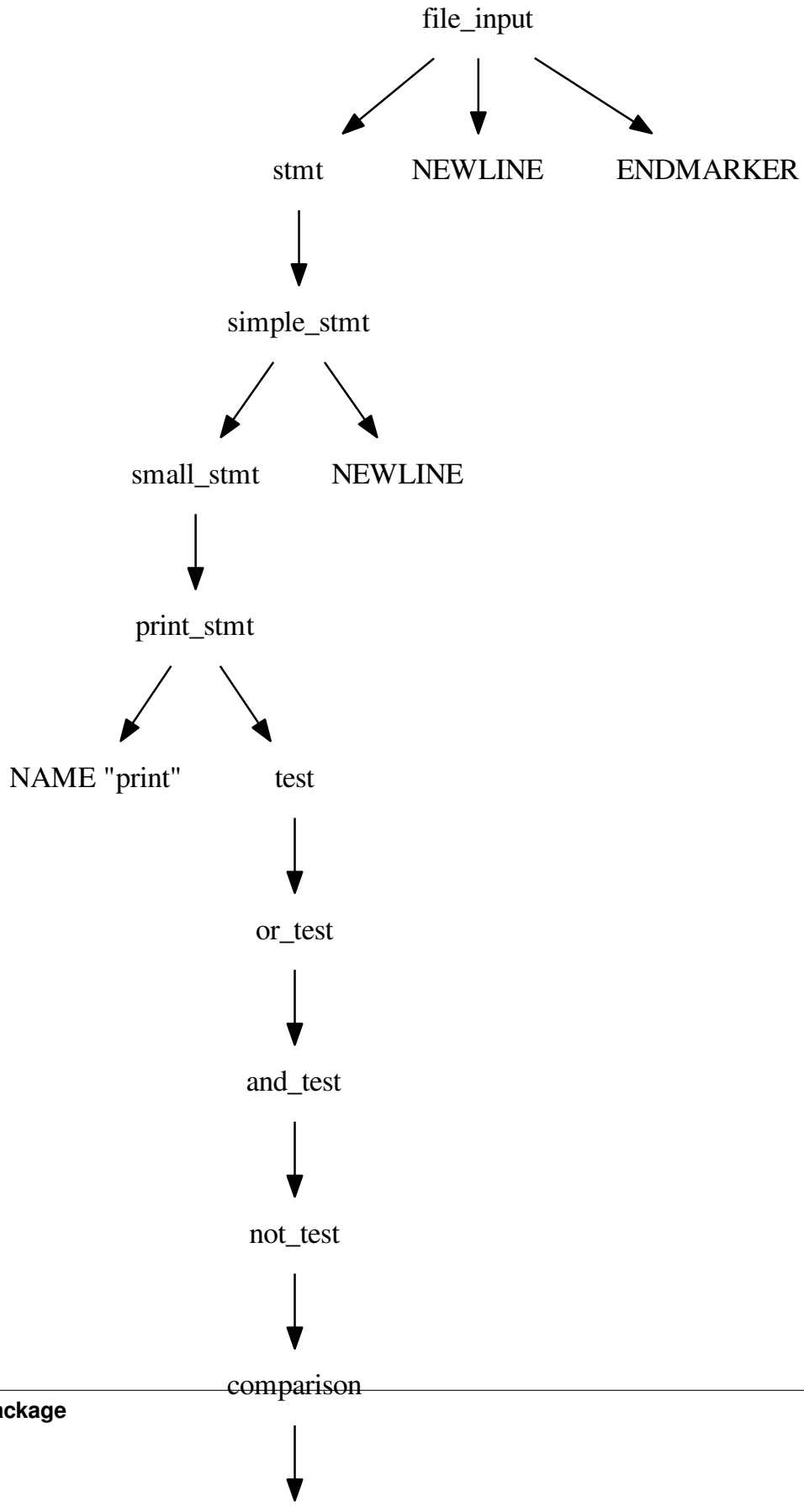
- **source** (*string*) – Source code
- **totuple** (*boolean*) – (for testing) Return internal parser data structure, don't convert to `parser.st` object

Returns Parse Tree

Return type parser.st

Raises `SyntaxError` – Syntax error in the source code

Parse Tree of Hello World Example:


file_input
stmt
NEWLINE
ENDMARKER
simple_stmt
small_stmt
NEWLINE
print_stmt
NAME "print"
test
or_test
and_test
not_test
comparison

See also:

- Python Language Reference: https://docs.python.org/2/reference/grammar.html
- Non-Terminal Symbols: https://hg.python.org/cpython/file/2.7/Lib/symbol.py
- Leaf Nodes: https://docs.python.org/2/library/token.html
- LL Parser: https://en.wikipedia.org/wiki/LL_parser
- Recursive-Descent Parser: https://en.wikipedia.org/wiki/Recursive_descent_parser
- Future Statement Definitions: https://docs.python.org/2/library/__future__.html

pycep.parser.**listit**(*tup*)
Recursively convert list-of-lists to tuples-of-tuples

pycep.parser.**_single_input**(*tokens*)
Parse a single input.

```
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
```

pycep.parser.**_file_input**(*tokens*)
Parse a module or sequence of command read from an input file.

```
file_input: (NEWLINE | stmt)* ENDMARKER
```

pycep.parser.**_eval_input**(*tokens*)
Parse an evaluation input.

```
eval_input: testlist NEWLINE* ENDMARKER
```

pycep.parser.**_decorator**(*tokens*)
Parse a decorator.

```
decorator: '@' dotted_name [ '(' [arglist] ')' ] NEWLINE
```

pycep.parser.**_decorators**(*tokens*)
Parse a list of decorators.

```
decorators: decorator+
```

pycep.parser.**_decorated**(*tokens*)
Parse a decorated statement.

```
decorated: decorators (classdef | funcdef)
```

pycep.parser.**_funcdef**(*tokens*)
Parse a function definition.

```
funcdef: 'def' NAME parameters ':' suite
```

pycep.parser.**_parameters**(*tokens*)
Parse a parameter list.

```
parameters: '(' [varargslist] ')'
```

pycep.parser.**_varargslist**(*tokens*)
Parse a variable argument list.

```
varargslist: ((fpdef ['=' test] ',')*
              ('*' NAME [',' '**' NAME] | '**' NAME) |
              fpdef ['=' test] (',' fpdef ['=' test])* [','])
```

pycep.parser.**_fpdef**(*tokens*)
Parse function parameter definition.

```
fpdef: NAME | '(' fplist ')'
```

pycep.parser.**_fplist**(*tokens*)
Parse a function parameter list

```
fplist: fpdef (',' fpdef)* [',']
```

pycep.parser.**_stmt**(*tokens*)
Parse a statement.

```
stmt: simple_stmt | compound_stmt
```

pycep.parser.**_simple_stmt**(*tokens*)
Parse a simple statement.

```
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
```

pycep.parser.**_small_stmt**(*tokens*)
Parse a small statement.

```
small_stmt: (expr_stmt | print_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | exec_stmt | assert_stmt)
```

pycep.parser.**_expr_stmt**(*tokens*)
Parse an expr stmt.

```
expr_stmt: testlist (augassign (yield_expr|testlist) |
                     ('=' (yield_expr|testlist))*)
```

pycep.parser.**_augassign**(*tokens*)
Parse an augmented assign statement.

```
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
'<<=' | '>>=' | '**=' | '//=')
```

pycep.parser.**_print_stmt**(*tokens*)
Parse a print statement.

```
print_stmt: 'print' ( [ test (',' test)* [','] ] |
                      '>>' test [ (',' test)+ [','] ] )
```

pycep.parser.**_del_stmt**(*tokens*)
Parse a delete statement.

```
del_stmt: 'del' exprlist
```

pycep.parser.**_pass_stmt**(*tokens*)
Parse a pass statement.

```
pass_stmt: 'pass'
```

pycep.parser.**_flow_stmt**(*tokens*)
Parse a flow statement.

```
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
```

pycep.parser.**_break_stmt**(*tokens*)
Parse a break statement.

```
break_stmt: 'break'
```

pycep.parser.**_continue_stmt**(*tokens*)
Parse a continue statement.

```
continue_stmt: 'continue'
```

pycep.parser.**_return_stmt**(*tokens*)
Parse a return statement.

```
return_stmt: 'return' [testlist]
```

pycep.parser.**_yield_stmt**(*tokens*)
Parse a yield statement.

```
yield_stmt: yield_expr
```

pycep.parser.**_raise_stmt**(*tokens*)
Parse a raise statement.

```
raise_stmt: 'raise' [test [',' test [',' test]]]
```

pycep.parser.**_import_stmt**(*tokens*)
Parse an import statement.

```
import_stmt: import_name | import_from
```

pycep.parser.**_import_name**(*tokens*)
Parse an import name.

```
import_name: 'import' dotted_as_names
```

pycep.parser.**_import_from**(*tokens*)
Parse an import from.

```
import_from: ('from' ('.'* dotted_name | '.'+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
```

pycep.parser.**_import_as_name**(*tokens*)
Parse an import as names.

```
import_as_name: NAME ['as' NAME]
```

pycep.parser.**_dotted_as_name**(*tokens*)
Parse a dotted as name.

```
dotted_as_name: dotted_name ['as' NAME]
```

pycep.parser.**_import_as_names**(*tokens*)
Parse import as names.

```
import_as_names: import_as_name (',' import_as_name)* [',']
```

pycep.parser.**_dotted_as_names**(*tokens*)
Parse dotted as names.

```
dotted_as_names: dotted_as_name (',' dotted_as_name)*
```

pycep.parser.**_dotted_name**(*tokens*)
Parse a dotted name.

```
dotted_name: NAME ('.' NAME)*
```

pycep.parser.**_global_stmt**(*tokens*)
Parse a global statement.

```
global_stmt: 'global' NAME (',' NAME)*
```

pycep.parser.**_exec_stmt**(*tokens*)
Parse an exec statement.

```
exec_stmt: 'exec' expr ['in' test [',' test]]
```

pycep.parser.**_assert_stmt**(*tokens*)
Parse an assert statement.

```
assert_stmt: 'assert' test [',' test]
```

pycep.parser.**_compound_stmt**(*tokens*)
Parse a compound statement.

```
compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef |␣
↪classdef | decorated
```

pycep.parser.**_if_stmt**(*tokens*)
Parse and if statement.

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
```

pycep.parser.**_while_stmt**(*tokens*)
Parse a while statement.

```
while_stmt: 'while' test ':' suite ['else' ':' suite]
```

pycep.parser.**_for_stmt**(*tokens*)
Parse a for statement.

```
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
```

pycep.parser.**_try_stmt**(*tokens*)
Parse a try statement.

```
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
```

pycep.parser.**_with_stmt**(*tokens*)
Parse a with statement.

```
with_stmt: 'with' with_item (',' with_item)*  ':' suite
```

pycep.parser.**_with_item**(*tokens*)
Parse a with item.

```
with_item: test ['as' expr]
```

pycep.parser.**_except_clause**(*tokens*)
Parse an except clause.

```
except_clause: 'except' [test [('as' | ',') test]]
```

pycep.parser.**_suite**(*tokens*)
Parse a suite.

```
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
```

pycep.parser.**_testlist_safe**(*tokens*)
Parse a testlist safe.

```
testlist_safe: old_test [(',' old_test)+ [',']]
```

pycep.parser.**_old_test**(*tokens*)
Parse an old test.

```
old_test: or_test | old_lambdef
```

pycep.parser.**_old_lambdef**(*tokens*)
Parse an old lambda definition.

```
old_lambdef: 'lambda' [varargslist] ':' old_test
```

pycep.parser.**_test**(*tokens*)
Parse a test statement.

```
test: or_test ['if' or_test 'else' test] | lambdef
```

pycep.parser.**_or_test**(*tokens*)
Parse an or_test statement

```
or_test: and_test ('or' and_test)*
```

pycep.parser.**_and_test**(*tokens*)
Parse an and test statement.

```
and_test: not_test ('and' not_test)*
```

pycep.parser.**_not_test**(*tokens*)
Parse a not test statement.

```
not_test: 'not' not_test | comparison
```

pycep.parser.**_comparison**(*tokens*)
Parse a comparison.

```
comparison: expr (comp_op expr)*
```

pycep.parser.**_comp_op**(*tokens*)
Parse a compare operator statement.

```
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
```

pycep.parser.**_expr**(*tokens*)
Parse an expression statement.

```
expr: xor_expr ('|' xor_expr)*
```

pycep.parser.**_xor_expr**(*tokens*)
Parse an xor expression statement.

```
xor_expr: and_expr ('^' and_expr)*
```

pycep.parser.**_and_expr**(*tokens*)
Parse an and expression statement.

```
and_expr: shift_expr ('&' shift_expr)*
```

pycep.parser.**_shift_expr**(*tokens*)
Parse a shift_expr statement

```
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
```

pycep.parser.**_arith_expr**(*tokens*)
Parse an arithmetic expression statement.

```
arith_expr: term (('+'|'-') term)*
```

pycep.parser.**_term**(*tokens*)
Parse a term statement.

```
term: factor (('*'|'/'|'%'|'//') factor)*
```

pycep.parser.**_factor**(*tokens*)
Parse a factor statement.

```
factor: ('+'|'-'|'~') factor | power
```

pycep.parser.**_power**(*tokens*)
Parse a power statement.

```
power: atom trailer* ['**' factor]
```

pycep.parser.**_atom**(*tokens*)
Parse an atom statement.

```
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [listmaker] ']' |
       '{' [dictorsetmaker] '}' |
       '`' testlist1 '`' |
       NAME | NUMBER | STRING+)
```

pycep.parser.**_listmaker**(*tokens*)
Parse a listmaker statement.

```
listmaker: test ( list_for | (',' test)* [','] )
```

pycep.parser.**_testlist_comp**(*tokens*)
Parse a testlist comp statement.

```
testlist_comp: test ( comp_for | (',' test)* [','] )
```

pycep.parser.**_lambdef**(*tokens*)
Parse a lambda definition.

```
lambdef: 'lambda' [varargslist] ':' test
```

pycep.parser.**_trailer**(*tokens*)
Parse a trailer.

```
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
```

pycep.parser.**_subscriptlist**(*tokens*)
Parse a subscriptlist.

```
subscriptlist: subscript (',' subscript)* [',']
```

pycep.parser.**_subscript**(*tokens*)
Parse a subscript.

```
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]
```

pycep.parser.**_sliceop**(*tokens*)
Parse a slice operation.

```
sliceop: ':' [test]
```

pycep.parser.**_exprlist**(*tokens*)
Parse an expression list.

```
exprlist: expr (',' expr)* [',']
```

pycep.parser.**_testlist**(*tokens*)
Parse a testlist.

```
testlist: test (',' test)* [',']
```

pycep.parser.**_dictorsetmaker**(*tokens*)
Parse a dict or set maker statement.

```
dictorsetmaker: ( (test ':' test (comp_for | (',' test ':' test)* [','])) |
                  (test (comp_for | (',' test)* [','])) )
```

pycep.parser.**_classdef**(*tokens*)
Parse a class definition.

```
classdef: 'class' NAME ['(' [testlist] ')'] ':' suite
```

pycep.parser.**_arglist**(*tokens*)
Parse an argument list.

```
arglist: (argument ',')* (argument [',']
                         |'*' test (',' argument)* [',' '**' test]
                         |'**' test)
```

pycep.parser.**_argument**(*tokens*)
Parse an argument.

```
argument: test [comp_for] | test '=' test
```

pycep.parser.**_list_iter**(*tokens*)
Parse a list iteration.

```
list_iter: list_for | list_if
```

pycep.parser.**_list_for**(*tokens*)
Parse a list for.

```
list_for: 'for' exprlist 'in' testlist_safe [list_iter]
```

pycep.parser.**_list_if**(*tokens*)
Parse a list if.

```
list_if: 'if' old_test [list_iter]
```

pycep.parser.**_comp_iter**(*tokens*)
Parse a generator expression.

```
comp_iter: comp_for | comp_if
```

pycep.parser.**_comp_for**(*tokens*)
Parse a for generator expression.

```
comp_for: 'for' exprlist 'in' or_test [comp_iter]
```

pycep.parser.**_comp_if**(*tokens*)
Parse an if generator expression.

```
comp_if: 'if' old_test [comp_iter]
```

pycep.parser.**_testlist1**(*tokens*)
Parse a testlist1.

```
testlist1: test (',' test)*
```

pycep.parser.**_encoding_decl**(*tokens*)
Parse an encoding declaration.

```
encoding_decl: NAME
```

pycep.parser.**_yield_expr**(*tokens*)
Parse a yield expression.

```
yield_expr: 'yield' [testlist]
```

class pycep.parser.**TokenIterator**(*generator*, *skip_tokens=(54, 53)*)
Wrapper for token generator which allows checking for and accepting tokens. Physical line breaks (tokenize.NL) and comments (token.N_TOKENS) are skipped from the input.

check_test(*lookahead=1*)
Shorthand notation to check whether next statement is a test

check_comp_op()
Shorthand notation to check whether next statement is a comp_op

check_expr(*lookahead=1*)
Shorthand notation to check whether next statement is an expr

5.1.3 pycep.analyzer Module

pycep.analyzer.**parse**(*source*)
The analyzer takes a string containing the source code as an input and returns an abstract syntax tree.

```
>>> import pycep.analyzer
>>> import ast
>>> tree = pycep.analyzer.parse('print "Hello, world!"')
>>> ast.dump(tree)
"Module(body=[Print(dest=None, values=[Str(s='Hello, world!')], nl=True)])"
```

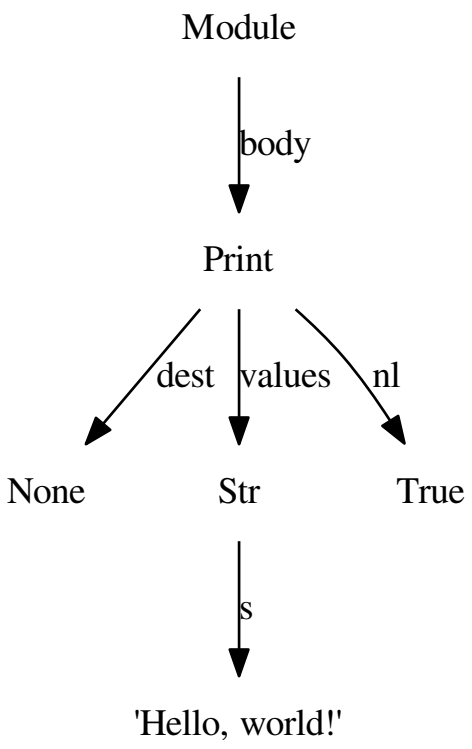
Parameters **source** (*string*) – Source code

Returns Abstract Syntax Tree

Return type ast.AST

Raises SyntaxError – Syntax Error

Abstract Syntax Tree of Hello World Example:



See also:

- Python Abstract Grammar: https://docs.python.org/2/library/ast.html
- Green Tree Snakes - the missing Python AST docs: https://greentreesnakes.readthedocs.org/

class pycep.analyzer.**Analyzer**
Convert a parse tree into an abstract syntax tree.

See also:

- Visitor Design Pattern https://sourcemaking.com/design_patterns/visitor

visit_single_input(*values*, *ctx*)
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE

visit_file_input(*values*, *ctx*)
file_input: (NEWLINE | stmt)* ENDMARKER

visit_eval_input(*values*, *ctx*)
eval_input: testlist NEWLINE* ENDMARKER

visit_decorator(*values*, *ctx*)
decorator: '@' dotted_name ['(' [arglist] ')'] NEWLINE

visit_decorators (*values*, *ctx*)
decorators: decorator+

visit_decorated (*values*, *ctx*)
decorated: decorators (classdef | funcdef)

visit_funcdef (*values*, *ctx*)
funcdef: 'def' NAME parameters ':' suite

visit_parameters (*values*, *ctx*)
parameters: '(' [varargslist] ')'

visit_varargslist (*values*, *ctx*)
varargslist: ((fpdef ['=' test] ',')* ('*' NAME [',' '**' NAME] | '**' NAME) | fpdef ['=' test] (',' fpdef ['=' test])* [','])

visit_fpdef (*values*, *ctx*)
fpdef: NAME | '(' fplist ')'

visit_fplist (*values*, *ctx*)
fplist: fpdef (',' fpdef)* [',']

visit_stmt (*values*, *ctx*)
stmt: simple_stmt | compound_stmt

visit_simple_stmt (*values*, *ctx*)
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE

visit_small_stmt (*values*, *ctx*)
small_stmt: (expr_stmt | print_stmt | del_stmt | pass_stmt | flow_stmt | import_stmt | global_stmt | exec_stmt | assert_stmt)

visit_expr_stmt (*values*, *ctx*)
expr_stmt: testlist (augassign (yield_expr|testlist) | ('=' (yield_expr|testlist))*)

visit_augassign (*values*, *ctx*)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<=' | '>>=' | '**=' | '//=')

visit_print_stmt (*values*, *ctx*)
print_stmt: 'print' ([test (',' test)* [',']] | '>>' test [(',' test)+ [',']])

visit_del_stmt (*values*, *ctx*)
del_stmt: 'del' exprlist

visit_pass_stmt (*values*, *ctx*)
pass_stmt: 'pass'

visit_flow_stmt (*values*, *ctx*)
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt

visit_break_stmt (*values*, *ctx*)
break_stmt: 'break'

visit_continue_stmt (*values*, *ctx*)
continue_stmt: 'continue'

visit_return_stmt (*values*, *ctx*)
return_stmt: 'return' [testlist]

visit_yield_stmt (*values*, *ctx*)
yield_stmt: yield_expr

visit_raise_stmt (*values*, *ctx*)
raise_stmt: 'raise' [test [',' test [',' test]]]

visit_import_stmt(*values*, *ctx*)
import_stmt: import_name | import_from

visit_import_name(*values*, *ctx*)
import_name: 'import' dotted_as_names

visit_import_from(*values*, *ctx*)
import_from: ('from' ('.'* dotted_name | '.'+) 'import' ('*' | '(' import_as_names ')' | import_as_names))

visit_import_as_name(*values*, *ctx*)
import_as_name: NAME ['as' NAME]

visit_dotted_as_name(*values*, *ctx*)
dotted_as_name: dotted_name ['as' NAME]

visit_import_as_names(*values*, *ctx*)
import_as_names: import_as_name (',' import_as_name)* [',']

visit_dotted_as_names(*values*, *ctx*)
dotted_as_names: dotted_as_name (',' dotted_as_name)*

visit_dotted_name(*values*, *ctx*)
dotted_name: NAME ('.' NAME)*

visit_global_stmt(*values*, *ctx*)
global_stmt: 'global' NAME (',' NAME)*

visit_exec_stmt(*values*, *ctx*)
exec_stmt: 'exec' expr ['in' test [',' test]]

visit_assert_stmt(*values*, *ctx*)
assert_stmt: 'assert' test [',' test]

visit_compound_stmt(*values*, *ctx*)
compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decorated

visit_if_stmt(*values*, *ctx*)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]

visit_while_stmt(*values*, *ctx*)
while_stmt: 'while' test ':' suite ['else' ':' suite]

visit_for_stmt(*values*, *ctx*)
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]

visit_try_stmt(*values*, *ctx*)
try_stmt: ('try' ':' suite ((except_clause ':' suite)+

['else' ':' suite] ['finally' ':' suite] |

'finally' ':' suite))

visit_with_stmt(*values*, *ctx*)
with_stmt: 'with' with_item (',' with_item)* ':' suite

visit_with_item(*values*, *ctx*)
with_item: test ['as' expr]

visit_except_clause(*values*, *ctx*)
except_clause: 'except' [test [('as' | ',') test]]

visit_suite(*values*, *ctx*)
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

visit_testlist_safe(*values*, *ctx*)
testlist_safe: old_test [(',' old_test)+ [',']]

visit_old_test(*values*, *ctx*)
old_test: or_test | old_lambdef

visit_old_lambdef(*values*, *ctx*)
old_lambdef: 'lambda' [varargslist] ':' old_test

visit_test(*values*, *ctx*)
test: or_test ['if' or_test 'else' test] | lambdef

visit_or_test(*values*, *ctx*)
or_test: and_test ('or' and_test)*

visit_and_test(*values*, *ctx*)
and_test: not_test ('and' not_test)*

visit_not_test(*values*, *ctx*)
not_test: 'not' not_test | comparison

visit_comparison(*values*, *ctx*)
comparison: expr (comp_op expr)*

visit_comp_op(*values*, *ctx*)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'

visit_expr(*values*, *ctx*)
expr: xor_expr ('|' xor_expr)*

visit_xor_expr(*values*, *ctx*)
xor_expr: and_expr ('^' and_expr)*

visit_and_expr(*values*, *ctx*)
and_expr: shift_expr ('&' shift_expr)*

visit_shift_expr(*values*, *ctx*)
shift_expr: arith_expr (('<<'|'>>') arith_expr)*

visit_arith_expr(*values*, *ctx*)
arith_expr: term (('+'|'-') term)*

visit_term(*values*, *ctx*)
term: factor (('*'|'/'|'%'|'//') factor)*

visit_factor(*values*, *ctx*)
factor: ('+'|'-'|'~') factor | power

visit_power(*values*, *ctx*)
power: atom trailer* ['**' factor]

visit_atom(*values*, *ctx*)
atom: ('(' [yield_expr|testlist_comp] ')' | '[' [listmaker] ']' | '{' [dictorsetmaker] '}' | '`' testlist1 '`' | NAME | NUMBER | STRING+)

visit_listmaker(*values*, *ctx*)
listmaker: test (list_for | (',' test)* [','])

visit_testlist_comp(*values*, *ctx*)
testlist_comp: test (comp_for | (',' test)* [','])

visit_lambdef(*values*, *ctx*)
lambdef: 'lambda' [varargslist] ':' test

visit_trailer(*values*, *ctx*)
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME

visit_subscriptlist(*values*, *ctx*)
subscriptlist: subscript (',' subscript)* [',']

visit_subscript(*values*, *ctx*)
subscript: '.' '.' '.' | test | [test] ':' [test] [sliceop]

visit_sliceop(*values*, *ctx*)
sliceop: ':' [test]

visit_exprlist(*values*, *ctx*)
exprlist: expr (',' expr)* [',']

visit_testlist(*values*, *ctx*)
testlist: test (',' test)* [',']

visit_dictorsetmaker(*values*, *ctx*)
dictorsetmaker: ((test ':' test (comp_for | (',' test ':' test)* [','])) | (test (comp_for | (',' test)* [','])))

visit_classdef(*values*, *ctx*)
classdef: 'class' NAME ['(' [testlist] ')'] ':' suite

visit_arglist(*values*, *ctx*)
arglist: (argument ',')* (argument [','] |'*' test (',' argument)* [',' '**' test] |'**' test)

visit_argument(*values*, *ctx*)
argument: test [comp_for] | test '=' test

visit_list_iter(*values*, *ctx*)
list_iter: list_for | list_if

visit_list_for(*values*, *ctx*)
list_for: 'for' exprlist 'in' testlist_safe [list_iter]

visit_list_if(*values*, *ctx*)
list_if: 'if' old_test [list_iter]

visit_comp_iter(*values*, *ctx*)
comp_iter: comp_for | comp_if

visit_comp_for(*values*, *ctx*)
comp_for: 'for' exprlist 'in' or_test [comp_iter]

visit_testlist1(*values*, *ctx*)
testlist1: test (',' test)*

visit_encoding_decl(*values*, *ctx*)
encoding_decl: NAME

visit_yield_expr(*values*, *ctx*)
yield_expr: 'yield' [testlist]

5.1.4 pycep.interpreter Module

pycep.interpreter.**execfile**(*filename*)
The interpreter takes a file path pointing to a python program and then executes its contents.

```
>>> import pycep.interpreter
>>> pycep.interpreter.execfile("pycep/tests/programs/helloworld.py")
Hello, world!
```

class pycep.interpreter.**Interpreter**
An AST-based interpreter

See also:

- Visitor Design Pattern https://sourcemaking.com/design_patterns/visitor

visit_Name(*node*, *scope*)
Return the value or raise a NameError if not found.

visit(*node*, *scope=None*)
Visit a node.

bind(*name*, *value*, *local_scope=None*)
Bind a variable name to a value.

Parameters

- **name** (*string*) – The variable name to bind
- **value** (*ast.AST*) – The value to store
- **local_scope** (*ast.AST*) – The local scope to apply, `None` to store in globals

resolve(*name*, *local_scope=None*)
Find a variable name according to the LEGB rule.

A *namespace* maps names to objects, implemented as a dictionary.

A *scope* defines at which hierarchy level to search for a particular variable name, i.e. which namespace to apply.

Python uses the LEGB (Local, Enclosed, Global, Builtin) rule for scope lookups.

Parameters

- **name** (*string*) – The variable name to look up
- **local_scope** (*ast.AST*) – The local scope to apply, `None` if there is no local scope to consider

Returns The object found in the hierarchy corresponding to `name`

Return type parser.st

Raises `NameError` – Name not defined

See also:

- Python's Namespaces, Scope Resolution, and the LEGB Rule: http://spartanideas.msu.edu/2014/05/12/a-beginners-guide-to-pythons-namespaces-scope-resolution-and-the-legb-rule/
- Python Scopes and Namespaces: https://docs.python.org/2/tutorial/classes.html#python-scopes-and-namespaces
- Variables and scope: http://python-textbok.readthedocs.org/en/latest/Variables_and_Scope.html
- Gotcha: Python, scoping and closures: http://eev.ee/blog/2011/04/24/gotcha-python-scoping-closures/

5.2 Parser

The type of parser implemented is called a *recursive-descent parser with backtracking*.

5.2.1 Eliminating Ambiguities

Some parts of the grammar are not suitable for parsing by recursive descent, because they are *not left-factored*. Consider the following example where both productions share the common prefix `test`:

```
argument: test [comp_for] | test '=' test
```

The issue with this production is that the recursive-descent parser could go down the first path, return with a success and never consider the second path. We can left-factor the grammar by making a new nonterminal `option` to represent the two alternatives for the symbols following the common prefix:

```
argument: test option
option:   ϵ | comp_for | '=' test
```

..where ϵ denotes the empty string. This production is now suitable for the recursive-descent parser.

5.2.2 See also

- Eliminating Ambiguities http://ycpcs.github.io/cs340-fall2015/lectures/lecture05.html

5.3 Portability

Although PyCep is written in pure Python and it tries to be as self-sufficient as possible, it defers certain elements to the host language (which conveniently happens to be Python). These items are listed here:

- String literal evaluation (see `ast.literal_eval()`)
- String formatting (see `%`)
- Type checking, e.g. for arithmetic operations
- Built-ins

5.4 Tools

Thank you to the tool builders for these great tools:

- Code hosted on https://github.com/dlorch/pycep/
- Integration test running on https://travis-ci.org/dlorch/pycep
- Package hosting https://pypi.python.org/pypi/pycep
- Documentation / API Doc generated by http://readthedocs.org/ to http://www.pycep.org/
- Visual Studio Code https://code.visualstudio.com/ with Python Extension https://marketplace.visualstudio.com/items?itemName=donjayamanne.python
- Diagrams https://www.gliffy.com/
- Interactive Demo is running as AWS Lambda Functions https://aws.amazon.com/de/lambda/
- Editor for interactive demo is Ace https://ace.c9.io/

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols