32.1. `parser` — Access Python parse trees¶

The `parser` module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

Note

From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the `ast` module.

The `parser` module exports the names documented here also with "st" replaced by "ast"; this is a legacy from the time when there was no other AST and has nothing to do with the AST found in Python 2.5. This is also the reason for the functions' keyword arguments being called *ast*, not *st*. The "ast" functions will be removed in Python 3.0.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the `parser` module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to *The Python Language Reference*. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the `expr()` or `suite()` functions, described below. The ST objects created by `sequence2st()` faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by `st2list()` or `st2tuple()` has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module `symbol`. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module `token`.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The `parser` module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

See also

Module `symbol`

Useful constants representing internal nodes of the parse tree.

Module `token`

Useful constants representing leaf nodes of the parse tree and functions for testing node values.

32.1.1. Creating ST Objects¶

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.expr`(*source*)¶
The `expr()` function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

`parser.suite`(*source*)¶
The `suite()` function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is thrown.

`parser.sequence2st`(*sequence*)¶

This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a `ParserError` exception is thrown. An ST object created this way should not be assumed to compile correctly; normal exceptions thrown by compilation may still be initiated when the ST object is passed to `compilest()`. This may indicate problems not related to syntax (such as a `MemoryError` exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

`parser.tuple2st`(*sequence*)¶
This is the same function as `sequence2st()`. This entry point is maintained for backward compatibility.

32.1.2. Converting ST Objects¶

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple- trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

`parser.st2list`(*ast*[, *line_info*])¶

This function accepts an ST object from the caller in *ast* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, `st2tuple()` should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

`parser.st2tuple`(*ast*[, *line_info*])¶

This function accepts an ST object from the caller in *ast* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to `st2list()`.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

`parser.compilest`(*ast*[, *filename='<syntax-tree>'*])¶

The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of an `exec` statement or a call to the built-in `eval()` function. This function provides the interface to the compiler, passing the internal parse tree from *ast* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a `SyntaxError` caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The `SyntaxError` raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the `parser` module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3. Queries on ST Objects¶

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via `expr()` or `suite()` or from a parse tree via `sequence2st()`.

`parser.isexpr`(*ast*)¶

When *ast* represents an `'eval'` form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by `compilest()` cannot be queried like this either, and are identical to those created by the built-in `compile()` function.

`parser.issuite`(*ast*)¶
This function mirrors `isexpr()` in that it reports whether an ST object represents an `'exec'` form, commonly known as a “suite.” It is not safe to assume that this function is equivalent to `not isexpr(ast)`, as additional syntactic fragments may be supported in the future.

32.1.4. Exceptions and Error Handling¶

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception `parser.ParserError`¶
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built in `SyntaxError` thrown during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to `sequence2st()` and an explanatory string. Calls to `sequence2st()` need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions `compilest()`, `expr()`, and `suite()` may throw exceptions which are normally thrown by the parsing and compilation process. These include the built in exceptions `MemoryError`, `OverflowError`, `SyntaxError`, and `SystemError`. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5. ST Objects¶

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the `pickle` module) is also supported.

`parser.STType`¶
The type of the objects returned by `expr()`, `suite()` and `sequence2st()`.

ST objects have the following methods:

`ST.compile`(*[filename]*)¶
Same as `compilest(st, filename)`.

`ST.isexpr`()¶
Same as `isexpr(st)`.

`ST.issuite`()¶
Same as `issuite(st)`.

`ST.tolist`(*[line_info]*)¶
Same as `st2list(st, line_info)`.

`ST.totuple`(*[line_info]*)¶
Same as `st2tuple(st, line_info)`.

32.1.6. Examples¶

The parser modules allows operations to be performed on the parse tree of Python source code before the *bytecode* is generated, and provides for inspection of the parse tree for information gathering purposes. Two examples are presented. The simple example demonstrates emulation of the `compile()` built-in function and the complex example shows the use of a parse tree for information discovery.

32.1.6.1. Emulation of `compile()`¶

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the `parser` module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the `parser` module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser
```

```
def load_suite(source_string):
   st = parser.suite(source_string)
   return st, st.compile()

def load_expression(source_string):
   st = parser.expr(source_string)
   return st, st.compile()
```

32.1.6.2. Information Discovery¶

Some applications benefit from direct access to the parse tree. The remainder of this section demonstrates how the parse tree provides access to module documentation defined in docstrings without requiring that the code being examined be loaded into a running interpreter via `import`. This can be very useful for performing analyses of untrusted code.

Generally, the example will demonstrate how the parse tree may be traversed to distill interesting information. Two functions and a set of classes are developed which provide programmatic access to high level function and class definitions provided by a module. The classes extract information from the parse tree and provide access to the information at a useful semantic level, one function provides a simple low-level pattern matching capability, and the other function defines a high-level interface to the classes by handling file operations on behalf of the caller. All source files mentioned here which are not part of the Python installation are located in the `Demo/parser/` directory of the distribution.

The dynamic nature of Python allows the programmer a great deal of flexibility, but most modules need only a limited measure of this when defining classes, functions, and methods. In this example, the only definitions that will be considered are those which are defined in the top level of their context, e.g., a function defined by a `def` statement at column zero of a module, but not a function defined within a branch of an `if` ... `else` construct, though there are some good reasons for doing so in some situations. Nesting of definitions will be handled by the code developed in the example.

To construct the upper-level extraction methods, we need to know what the parse tree structure looks like and how much of it we actually need to be concerned about. Python uses a moderately deep parse tree so there are a large number of intermediate nodes. It is important to read and understand the formal grammar used by Python. This is specified in the file `Grammar/Grammar` in the distribution. Consider the simplest case of interest when searching for docstrings: a module consisting of a docstring and nothing else. (See file `docstring.py`.)

```
"""Some documentation.
"""
```

Using the interpreter to take a look at the parse tree, we find a bewildering mass of numbers and parentheses, with the documentation buried deep in nested tuples.

```
>>> import parser
>>> import pprint
>>> st = parser.suite(open('docstring.py').read())
>>> tup = st.totuple()
>>> pprint.pprint(tup)
(257,
(264,
 (265,
  (266,
   (267,
    (307,
     (287,
      (288,
       (289,
        (290,
         (292,
          (293,
           (294,
            (295,
             (296,
              (297,
               (298,
                (299,
                 (300, (3, '"""Some documentation.\n"""')))))))))))))))),
  (4, ''))),
(4, ''),
(0, ''))
```

The numbers at the first element of each node in the tree are the node types; they map directly to terminal and non-terminal symbols in the grammar. Unfortunately, they are represented as integers in the internal representation, and the Python structures generated do not change that. However, the `symbol` and `token` modules provide symbolic names for the node types and dictionaries which map from the integers to the symbolic names for the node types.

In the output presented above, the outermost tuple contains four elements: the integer `257` and three additional tuples. Node type `257` has the symbolic name `file_input`. Each of these inner tuples contains an integer as the first element; these integers, `264`, `4`, and `0`, represent the node types `stmt`, `NEWLINE`, and

ENDMARKER, respectively. Note that these values may change depending on the version of Python you are using; consult `symbol.py` and `token.py` for details of the mapping. It should be fairly clear that the outermost node is related primarily to the input source rather than the contents of the file, and may be disregarded for the moment. The `stmt` node is much more interesting. In particular, all docstrings are found in subtrees which are formed exactly as this node is formed, with the only difference being the string itself. The association between the docstring in a similar tree and the defined entity (class, function, or module) which it describes is given by the position of the docstring subtree within the tree defining the described structure.

By replacing the actual docstring with something to signify a variable component of the tree, we allow a simple pattern matching approach to check any given subtree for equivalence to the general pattern for docstrings. Since the example demonstrates information extraction, we can safely require that the tree be in tuple form rather than list form, allowing a simple variable representation to be `['variable_name']`. A simple recursive function can implement the pattern matching, returning a Boolean and a dictionary of variable name to value mappings. (See file `example.py`.)

```
from types import ListType, TupleType

def match(pattern, data, vars=None):
   if vars is None:
       vars = {}
   if type(pattern) is ListType:
       vars[pattern[0]] = data
       return 1, vars
   if type(pattern) is not TupleType:
       return (pattern == data), vars
   if len(data) != len(pattern):
       return 0, vars
   for pattern, data in map(None, pattern, data):
       same, vars = match(pattern, data, vars)
       if not same:
           break
   return same, vars
```

Using this simple representation for syntactic variables and the symbolic node types, the pattern for the candidate docstring subtrees becomes fairly readable. (See file `example.py`.)

```
import symbol
import token

DOCSTRING_STMT_PATTERN = (
   symbol.stmt,
   (symbol.simple_stmt,
    (symbol.small_stmt,
     (symbol.expr_stmt,
      (symbol.testlist,
       (symbol.test,
        (symbol.and_test,
         (symbol.not_test,
          (symbol.comparison,
           (symbol.expr,
            (symbol.xor_expr,
             (symbol.and_expr,
              (symbol.shift_expr,
               (symbol.arith_expr,
                (symbol.term,
                 (symbol.factor,
                  (symbol.power,
                   (symbol.atom,
                    (token.STRING, ['docstring'])
                    ))))))))))))),
    (token.NEWLINE, '')
    ))
```

Using the `match()` function with this pattern, extracting the module docstring from the parse tree created previously is easy:

```
>>> found, vars = match(DOCSTRING_STMT_PATTERN, tup[1])
>>> found
1
>>> vars
{'docstring': '"""Some documentation.\n"""'}
```

Once specific data can be extracted from a location where it is expected, the question of where information can be expected needs to be answered. When dealing with docstrings, the answer is fairly simple: the docstring is the first `stmt` node in a code block (`file_input` or `suite` node types). A module consists of a single `file_input` node, and class and function definitions each contain exactly one `suite` node. Classes and functions are readily identified as subtrees

of code block nodes which start with `(stmt, (compound_stmt, (classdef, ...` or `(stmt, (compound_stmt, (funcdef, ...`. Note that these subtrees cannot be matched by `match()` since it does not support multiple sibling nodes to match without regard to number. A more elaborate matching function could be used to overcome this limitation, but this is sufficient for the example.

Given the ability to determine whether a statement might be a docstring and extract the actual string from the statement, some work needs to be performed to walk the parse tree for an entire module and extract information about the names defined in each context of the module and associate any docstrings with the names. The code to perform this work is not complicated, but bears some explanation.

The public interface to the classes is straightforward and should probably be somewhat more flexible. Each “major” block of the module is described by an object providing several methods for inquiry and a constructor which accepts at least the subtree of the complete parse tree which it represents. The `ModuleInfo` constructor accepts an optional *name* parameter since it cannot otherwise determine the name of the module.

The public classes include `ClassInfo`, `FunctionInfo`, and `ModuleInfo`. All objects provide the methods `get_name()`, `get_docstring()`, `get_class_names()`, and `get_class_info()`. The `ClassInfo` objects support `get_method_names()` and `get_method_info()` while the other classes provide `get_function_names()` and `get_function_info()`.

Within each of the forms of code block that the public classes represent, most of the required information is in the same form and is accessed in the same way, with classes having the distinction that functions defined at the top level are referred to as “methods.” Since the difference in nomenclature reflects a real semantic distinction from functions defined outside of a class, the implementation needs to maintain the distinction. Hence, most of the functionality of the public classes can be implemented in a common base class, `SuiteInfoBase`, with the accessors for function and method information provided elsewhere. Note that there is only one class which represents function and method information; this parallels the use of the `def` statement to define both types of elements.

Most of the accessor functions are declared in `SuiteInfoBase` and do not need to be overridden by subclasses. More importantly, the extraction of most information from a parse tree is handled through a method called by the `SuiteInfoBase` constructor. The example code for most of the classes is clear when read alongside the formal grammar, but the method which recursively creates new information objects requires further examination. Here is the relevant part of the `SuiteInfoBase` definition from `example.py`:

```
class SuiteInfoBase:
   _docstring = ''
   _name = ''

   def __init__(self, tree = None):
       self._class_info = {}
       self._function_info = {}
       if tree:
           self._extract_info(tree)

   def _extract_info(self, tree):
       # extract docstring
       if len(tree) == 2:
           found, vars = match(DOCSTRING_STMT_PATTERN[1], tree[1])
       else:
           found, vars = match(DOCSTRING_STMT_PATTERN, tree[3])
       if found:
           self._docstring = eval(vars['docstring'])
       # discover inner definitions
       for node in tree[1:]:
           found, vars = match(COMPOUND_STMT_PATTERN, node)
           if found:
               cstmt = vars['compound']
               if cstmt[0] == symbol.funcdef:
                   name = cstmt[2][1]
                   self._function_info[name] = FunctionInfo(cstmt)
               elif cstmt[0] == symbol.classdef:
                   name = cstmt[2][1]
                   self._class_info[name] = ClassInfo(cstmt)
```

After initializing some internal state, the constructor calls the `_extract_info()` method. This method performs the bulk of the information extraction which takes place in the entire example. The extraction has two distinct phases: the location of the docstring for the parse tree passed in, and the discovery of additional definitions within the code block represented by the parse tree.

The initial `if` test determines whether the nested suite is of the “short form” or the “long form.” The short form is used when the code block is on the same line as the definition of the code block, as in

```
def square(x): "Square an argument."; return x ** 2
```

while the long form uses an indented block and allows nested definitions:

```
def make_power(exp):
   "Make a function that raises an argument to the exponent `exp`."
   def raiser(x, y=exp):
```

```
        return x ** y
    return raiser
```

When the short form is used, the code block may contain a docstring as the first, and possibly only, `small_stmt` element. The extraction of such a docstring is slightly different and requires only a portion of the complete pattern used in the more common case. As implemented, the docstring will only be found if there is only one `small_stmt` node in the `simple_stmt` node. Since most functions and methods which use the short form do not provide a docstring, this may be considered sufficient. The extraction of the docstring proceeds using the `match()` function as described above, and the value of the docstring is stored as an attribute of the `SuiteInfoBase` object.

After docstring extraction, a simple definition discovery algorithm operates on the `stmt` nodes of the `suite` node. The special case of the short form is not tested; since there are no `stmt` nodes in the short form, the algorithm will silently skip the single `simple_stmt` node and correctly not discover any nested definitions.

Each statement in the code block is categorized as a class definition, function or method definition, or something else. For the definition statements, the name of the element defined is extracted and a representation object appropriate to the definition is created with the defining subtree passed as an argument to the constructor. The representation objects are stored in instance variables and may be retrieved by name using the appropriate accessor methods.

The public classes provide any accessors required which are more specific than those provided by the `SuiteInfoBase` class, but the real extraction algorithm remains common to all forms of code blocks. A high-level function can be used to extract the complete set of information from a source file. (See file `example.py`.)

```
def get_docs(fileName):
    import os
    import parser

    source = open(fileName).read()
    basename = os.path.basename(os.path.splitext(fileName)[0])
    st = parser.suite(source)
    return ModuleInfo(st.totuple(), basename)
```

This provides an easy-to-use interface to the documentation of a module. If information is required which is not extracted by the code of this example, the code may be extended at clearly defined points to provide additional capabilities.

Last updated on Feb 26, 2010. Created using Sphinx 0.6.3.