32.2. `ast` — Abstract Syntax Trees

Source code: Lib/ast.py

The `ast` module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing `ast.PyCF_ONLY_AST` as a flag to the `compile()` built-in function, or using the `parse()` helper provided in this module. The result will be a tree of objects whose classes all inherit from `ast.AST`. An abstract syntax tree can be compiled into a Python code object using the built-in `compile()` function.

32.2.1. Node classes

class `ast.`**AST**

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced below. They are defined in the `_ast` C module and re-exported in `ast`.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, `ast.BinOp` inherits from `ast.expr`. For production rules with alternatives (aka "sums"), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

`_fields`

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute `left` of type `ast.expr`.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with `compile()`.

lineno
col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have `lineno` and `col_offset` attributes. The `lineno` is the line number of source text (1-indexed so the first line is line 1) and the `col_offset` is the UTF-8 byte offset of the first token that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

The constructor of a class `ast.T` parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in `T._fields`; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

or the more compact

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

32.2.2. Abstract Grammar

The abstract grammar is currently defined as follows:

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)
```

```
        -- not really an actual node but useful in Jython's typesystem
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? r

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without p
          | AnnAssign(expr target, expr annotation, expr? value, int s

          -- use 'orelse' because else is a keyword in target language
          | For(expr target, expr iter, stmt* body, stmt* orelse)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)
          | AsyncWith(withitem* items, stmt* body)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stm
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- XXX Jython will be different
          -- col_offset is the byte offset in the utf8 string the pars
          attributes (int lineno, int col_offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
```

```
      | Set(expr* elts)
      | ListComp(expr elt, comprehension* generators)
      | SetComp(expr elt, comprehension* generators)
      | DictComp(expr key, expr value, comprehension* generators)
      | GeneratorExp(expr elt, comprehension* generators)
      -- the grammar constrains where yield expressions can occur
      | Await(expr value)
      | Yield(expr? value)
      | YieldFrom(expr value)
      -- need sequences for compare to distinguish between
      -- x < 4 < 3 and (x < 4) < 3
      | Compare(expr left, cmpop* ops, expr* comparators)
      | Call(expr func, expr* args, keyword* keywords)
      | Num(object n) -- a number as a PyObject.
      | Str(string s) -- need to specify raw, unicode, etc?
      | FormattedValue(expr value, int? conversion, expr? format_sp
      | JoinedStr(expr* values)
      | Bytes(bytes s)
      | NameConstant(singleton value)
      | Ellipsis
      | Constant(constant value)

      -- the following expression can appear in assignment context
      | Attribute(expr value, identifier attr, expr_context ctx)
      | Subscript(expr value, slice slice, expr_context ctx)
      | Starred(expr value, expr_context ctx)
      | Name(identifier id, expr_context ctx)
      | List(expr* elts, expr_context ctx)
      | Tuple(expr* elts, expr_context ctx)

       -- col_offset is the byte offset in the utf8 string the pars
       attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Slice(expr? lower, expr? upper, expr? step)
      | ExtSlice(slice* dims)
      | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs, int is_async)

excepthandler = ExceptHandler(expr? type, identifier? name, stmt*
                attributes (int lineno, int col_offset)
```

```
    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_def
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwarg
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

32.2.3. `ast` Helpers

Apart from the node classes, the `ast` module defines these utility functions and classes for traversing abstract syntax trees:

`ast.`**`parse`**(*source*, *filename='<unknown>'*, *mode='exec'*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, ast.PyCF_ONLY_AST)`.

> **Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

`ast.`**`literal_eval`**(*node_or_string*)

Safely evaluate an expression node or a string containing a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and `None`.

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

> **Warning:** It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python's AST compiler.

Changed in version 3.2: Now allows bytes and set literals.

ast.**get_docstring**(*node*, *clean=True*)

Return the docstring of the given *node* (which must be a `FunctionDef`, `ClassDef` or `Module` node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with `inspect.cleandoc()`.

ast.**fix_missing_locations**(*node*)

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

ast.**increment_lineno**(*node*, *n=1*)

Increment the line number of each node in the tree starting at *node* by *n*. This is useful to "move code" to a different location in a file.

ast.**copy_location**(*new_node*, *old_node*)

Copy source location (`lineno` and `col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

ast.**iter_fields**(*node*)

Yield a tuple of `(fieldname, value)` for each field in `node._fields` that is present on *node*.

ast.**iter_child_nodes**(*node*)

Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.**walk**(*node*)

Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.**NodeVisitor**

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the `visit()` method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)

Visit a node. The default implementation calls the method called `self.visit_`*`classname`* where *classname* is the name of the node class, or `generic_visit()` if that method doesn't exist.

generic_visit(*node*)

This visitor calls `visit()` on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls `generic_visit()` or visits them itself.

Don't use the `NodeVisitor` if you want to apply changes to nodes during traversal. For this a special visitor exists (`NodeTransformer`) that allows modifications.

class `ast.`**NodeTransformer**

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

`ast.`**dump**(*node*, *annotate_fields=True*, *include_attributes=False*)

Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to `True`.

See also: Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.