Green Tree Snakes Documentation

Release 1.0

Thomas Kluyver

Feb 10, 2020

Contents

Abstract Syntax Trees, ASTs, are a powerful feature of Python. You can write programs that inspect and modify Python code, after the syntax has been parsed, but before it gets compiled to byte code. That opens up a world of possibilities for introspection, testing, and mischief.

The official documentation for the ast module is good, but somewhat brief. *Green Tree Snakes* is more like a field guide (or should that be forest guide?) for working with ASTs. To contribute to the guide, see the source repository.

Contents:

CHAPTER 1

Getting to and from ASTs

To build an ast from code stored as a string, use `ast.parse()`. To turn the ast into executable code, pass it to `compile()` (which can also compile a string directly).

```
>>> tree = ast.parse("print('hello world')")
>>> tree
<_ast.Module object at 0x9e3df6c>
>>> exec(compile(tree, filename="<ast>", mode="exec"))
hello world
```

1.1 Modes

Python code can be compiled in three modes. The root of the AST depends on the *mode* parameter you pass to `ast.parse()`, and it must correspond to the *mode* parameter when you call `compile()`.

- **exec** - Normal Python code is run with `mode='exec'`. The root of the AST is a `ast.Module`, whose `body` attribute is a list of nodes.
- **eval** - Single expressions are compiled with `mode='eval'`, and passing them to `eval()` will return their result. The root of the AST is an `ast.Expression`, and its `body` attribute is a single node, such as `ast.Call` or `ast.BinOp`. This is different from `ast.Expr`, which holds an expression within an AST.
- **single** - Single statements or expressions can be compiled with `mode='single'`. If it's an expression, `sys.displayhook()` will be called with the result, like when code is run in the interactive shell. The root of the AST is an `ast.Interactive`, and its `body` attribute is a list of nodes.

Note: The `type_comment` and `ignore_types` fields introduced in Python 3.8 are only populated if `ast.parse()` is called with `type_comment=True`.

1.2 Fixing locations

To compile an AST, every node must have `lineno` and `col_offset` attributes. Nodes produced by parsing regular code already have these, but nodes you create programmatically don't. There are a few helper functions for this:

- `ast.fix_missing_locations()` recursively fills in any missing locations by copying from the parent node. The rough and ready answer.
- `ast.copy_location()` copies `lineno` and `col_offset` from one node to another. Useful when you're replacing a node.
- `ast.increment_lineno()` increases `lineno` for a node and its children, pushing them further down a file.

1.3 Going backwards

Python itself doesn't provide a way to turn a compiled code object into an AST, or an AST into a string of code. Some third party tools can do these things:

- astor can convert an AST back to readable Python code.
- Meta also tries to decompile Python bytecode to an AST, but it appears to be unmaintained.
- uncompyle6 is an actively maintained Python decompiler at the time of writing. Its documented interface is a command line program producing Python source code.

CHAPTER 2

Meet the Nodes

An AST represents each element in your code as an object. These are instances of the various subclasses of `AST` described below. For instance, the code `a + 1` is a *`BinOp`*, with a *`Name`* on the left, a *`Num`* on the right, and an *`Add`* operator.

2.1 Literals

class Constant(*value*)
New in version 3.6.

A constant. The `value` attribute holds the Python object it represents. This can be simple types such as a number, string or `None`, but also immutable container types (tuples and frozensets) if all of their elements are constant.

This class is available in the `ast` module from Python 3.6, but it isn't produced by parsing code until Python 3.8.

class Num(*n*)
Deprecated since version 3.8: Replaced by *`Constant`*

A number - integer, float, or complex. The `n` attribute stores the value, already converted to the relevant type.

class Str(*s*)
Deprecated since version 3.8: Replaced by *`Constant`*

A string. The `s` attribute hold the value. In Python 2, the same type holds unicode strings too.

class FormattedValue(*value*, *conversion*, *format_spec*)
New in version 3.6.

Node representing a single formatting field in an f-string. If the string contains a single formatting field and nothing else the node can be isolated otherwise it appears in *`JoinedStr`*.

- `value` is any expression node (such as a literal, a variable, or a function call).
- `conversion` is an integer:

  - -1: no formatting
  - 115: `!s` string formatting
  - 114: `!r` repr formatting
  - 97: `!a` ascii formatting
- `format_spec` is a *JoinedStr* node reprensenting the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time.

class JoinedStr(*values*)

New in version 3.6.

An f-string, comprising a series of *FormattedValue* and *Str* nodes.

```
>>> parseprint('f"sin({a}) is {sin(a):.3}"')
Module(body=[
    Expr(value=JoinedStr(values=[
        Str(s='sin('),
        FormattedValue(value=Name(id='a', ctx=Load()), conversion=-1, format_
↪spec=None),
        Str(s=') is '),
        FormattedValue(value=Call(func=Name(id='sin', ctx=Load()), args=[
            Name(id='a', ctx=Load()),
          ], keywords=[]), conversion=-1, format_spec=JoinedStr(values=[
            Str(s='.3'),
          ])),
      ])),
  ])
```

Note: The pretty-printer used in these examples is available in the source repository for Green Tree Snakes.

class Bytes(*s*)

Deprecated since version 3.8: Replaced by *Constant*

A `bytes` object. The `s` attribute holds the value. Python 3 only.

class List(*elts*, *ctx*)

class Tuple(*elts*, *ctx*)

A list or tuple. `elts` holds a list of nodes representing the elements. `ctx` is *Store* if the container is an assignment target (i.e. `(x,y)=pt`), and *Load* otherwise.

class Set(*elts*)

A set. `elts` holds a list of nodes representing the elements.

class Dict(*keys*, *values*)

A dictionary. `keys` and `values` hold lists of nodes with matching order (i.e. they could be paired with `zip()`).

Changed in version 3.5: It is now possible to expand one dictionary into another, as in `{'a': 1, **d}`. In the AST, the expression to be expanded (a *Name* node in this example) goes in the `values` list, with a `None` at the corresponding position in `keys`.

class Ellipsis

Deprecated since version 3.8: Replaced by *Constant*

Represents the `...` syntax for the `Ellipsis` singleton.

class NameConstant(*value*)

New in version 3.4: Previously, these constants were instances of *Name*.

Deprecated since version 3.8: Replaced by *Constant*

`True`, `False` or `None`. `value` holds one of those constants.

2.2 Variables

class Name(*id*, *ctx*)

A variable name. `id` holds the name as a string, and `ctx` is one of the following types.

class Load
class Store
class Del

Variable references can be used to load the value of a variable, to assign a new value to it, or to delete it. Variable references are given a context to distinguish these cases.

```
>>> parseprint("a")       # Loading a
Module(body=[
    Expr(value=Name(id='a', ctx=Load())),
  ])

>>> parseprint("a = 1")   # Storing a
Module(body=[
    Assign(targets=[
        Name(id='a', ctx=Store()),
      ], value=Num(n=1)),
  ])

>>> parseprint("del a")   # Deleting a
Module(body=[
    Delete(targets=[
        Name(id='a', ctx=Del()),
      ]),
  ])
```

class Starred(*value*, *ctx*)

A `*var` variable reference. `value` holds the variable, typically a *Name* node.

Note that this *isn't* used to define a function with `*args` - *FunctionDef* nodes have special fields for that. In Python 3.5 and above, though, *Starred* is needed when building a *Call* node with `*args`.

```
>>> parseprint("a, *b = it")
Module(body=[
    Assign(targets=[
        Tuple(elts=[
            Name(id='a', ctx=Store()),
            Starred(value=Name(id='b', ctx=Store()), ctx=Store()),
          ], ctx=Store()),
      ], value=Name(id='it', ctx=Load())),
  ])
```

2.3 Expressions

class Expr(*value*)

When an expression, such as a function call, appears as a statement by itself (an expression statement), with

its return value not used or stored, it is wrapped in this container. `value` holds one of the other nodes in this section, or a literal, a *Name*, a *Lambda*, or a *Yield* or *YieldFrom* node.

```
>>> parseprint('-a')
Module(body=[
    Expr(value=UnaryOp(op=USub(), operand=Name(id='a', ctx=Load()))),
  ])
```

class UnaryOp(*op*, *operand*)
A unary operation. `op` is the operator, and `operand` any expression node.

class UAdd
class USub
class Not
class Invert
Unary operator tokens. *Not* is the `not` keyword, *Invert* is the ~ operator.

class BinOp(*left*, *op*, *right*)
A binary operation (like addition or division). `op` is the operator, and `left` and `right` are any expression nodes.

class Add
class Sub
class Mult
class Div
class FloorDiv
class Mod
class Pow
class LShift
class RShift
class BitOr
class BitXor
class BitAnd
class MatMult
Binary operator tokens.

New in version 3.5: *MatMult* - the `@` operator for matrix multiplication.

class BoolOp(*op*, *values*)
A boolean operation, 'or' or 'and'. `op` is *Or* or *And*. `values` are the values involved. Consecutive operations with the same operator, such as `a or b or c`, are collapsed into one node with several values.

This doesn't include `not`, which is a *UnaryOp*.

class And
class Or
Boolean operator tokens.

class Compare(*left*, *ops*, *comparators*)
A comparison of two or more values. `left` is the first value in the comparison, `ops` the list of operators, and `comparators` the list of values after the first. If that sounds awkward, that's because it is:

```
>>> parseprint("1 < a < 10")
Module(body=[
  Expr(value=Compare(left=Num(n=1), ops=[
      Lt(),
      Lt(),
    ], comparators=[
```

(continues on next page)

(continued from previous page)

```
      Name(id='a', ctx=Load()),
      Num(n=10),
    ])),
  ])
```

class Eq
class NotEq
class Lt
class LtE
class Gt
class GtE
class Is
class IsNot
class In
class NotIn

Comparison operator tokens.

class Call(*func*, *args*, *keywords*, *starargs*, *kwargs*)

A function call. `func` is the function, which will often be a *Name* or *Attribute* object. Of the arguments:

- `args` holds a list of the arguments passed by position.
- `keywords` holds a list of *keyword* objects representing arguments passed by keyword.
- `starargs` and `kwargs` each hold a single node, for arguments passed as `*args` and `**kwargs`. These are removed in Python 3.5 - see below for details.

When compiling a Call node, `args` and `keywords` are required, but they can be empty lists. `starargs` and `kwargs` are optional.

```
>>> parseprint("func(a, b=c, *d, **e)") # Python 3.4
Module(body=[
    Expr(value=Call(func=Name(id='func', ctx=Load()),
                    args=[Name(id='a', ctx=Load())],
                    keywords=[keyword(arg='b', value=Name(id='c', ctx=Load()))],
                    starargs=Name(id='d', ctx=Load()),     # gone in 3.5
                    kwargs=Name(id='e', ctx=Load()))),     # gone in 3.5
  ])

>>> parseprint("func(a, b=c, *d, **e)") # Python 3.5
Module(body=[
    Expr(value=Call(func=Name(id='func', ctx=Load()),
         args=[
                Name(id='a', ctx=Load()),
                Starred(value=Name(id='d', ctx=Load()), ctx=Load()) # new in 3.5
            ],
         keywords=[
                keyword(arg='b', value=Name(id='c', ctx=Load())),
                keyword(arg=None, value=Name(id='e', ctx=Load()))   # new in 3.5
            ]))
    ])
```

You can see here that the signature of *Call* has changed in Python 3.5. Instead of `starargs`, *Starred* nodes can now appear in `args`, and `kwargs` is replaced by *keyword* nodes in `keywords` for which `arg` is `None`.

class keyword(*arg*, *value*)

A keyword argument to a function call or class definition. `arg` is a raw string of the parameter name, `value` is

a node to pass in.

class IfExp(*test*, *body*, *orelse*)
An expression such as `a if b else c`. Each field holds a single node, so in that example, all three are *Name* nodes.

class Attribute(*value*, *attr*, *ctx*)
Attribute access, e.g. `d.keys`. `value` is a node, typically a *Name*. `attr` is a bare string giving the name of the attribute, and `ctx` is *Load*, *Store* or *Del* according to how the attribute is acted on.

```
>>> parseprint('snake.colour')
Module(body=[
    Expr(value=Attribute(value=Name(id='snake', ctx=Load()), attr='colour',␣
↪ctx=Load())),
  ])
```

2.3.1 Subscripting

class Subscript(*value*, *slice*, *ctx*)
A subscript, such as `l[1]`. `value` is the object, often a *Name*. `slice` is one of *Index*, *Slice* or *ExtSlice*. `ctx` is *Load*, *Store* or *Del* according to what it does with the subscript.

class Index(*value*)
Simple subscripting with a single value:

```
>>> parseprint("l[1]")
Module(body=[
  Expr(value=Subscript(value=Name(id='l', ctx=Load()),
                       slice=Index(value=Num(n=1)), ctx=Load())),
  ])
```

class Slice(*lower*, *upper*, *step*)
Regular slicing:

```
>>> parseprint("l[1:2]")
Module(body=[
  Expr(value=Subscript(value=Name(id='l', ctx=Load()),
                  slice=Slice(lower=Num(n=1), upper=Num(n=2), step=None),
                  ctx=Load())),
  ])
```

class ExtSlice(*dims*)
Advanced slicing. `dims` holds a list of *Slice* and *Index* nodes:

```
>>> parseprint("l[1:2, 3]")
Module(body=[
    Expr(value=Subscript(value=Name(id='l', ctx=Load()), slice=ExtSlice(dims=[
        Slice(lower=Num(n=1), upper=Num(n=2), step=None),
        Index(value=Num(n=3)),
      ]), ctx=Load())),
  ])
```

2.3.2 Comprehensions

class ListComp(*elt*, *generators*)

class SetComp(*elt*, *generators*)
class GeneratorExp(*elt*, *generators*)
class DictComp(*key*, *value*, *generators*)
List and set comprehensions, generator expressions, and dictionary comprehensions. `elt` (or `key` and `value`) is a single node representing the part that will be evaluated for each item.

`generators` is a list of *`comprehension`* nodes. Comprehensions with more than one `for` part are legal, if tricky to get right - see the example below.

class comprehension(*target*, *iter*, *ifs*, *is_async*)
One `for` clause in a comprehension. `target` is the reference to use for each element - typically a *`Name`* or *`Tuple`* node. `iter` is the object to iterate over. `ifs` is a list of test expressions: each `for` clause can have multiple `ifs`.

New in version 3.6: `is_async` indicates a comprehension is asynchronous (using an `async for` instead of `for`). The value is an integer (0 or 1).

```
  >>> parseprint("[ord(c) for line in file for c in line]", mode='eval') # Multiple␣
↪comprehensions in one.
  Expression(body=ListComp(elt=Call(func=Name(id='ord', ctx=Load()), args=[
      Name(id='c', ctx=Load()),
    ], keywords=[], starargs=None, kwargs=None), generators=[
      comprehension(target=Name(id='line', ctx=Store()), iter=Name(id='file',␣
↪ctx=Load()), ifs=[], is_async=0),
      comprehension(target=Name(id='c', ctx=Store()), iter=Name(id='line',␣
↪ctx=Load()), ifs=[], is_async=0),
    ]))

  >>> parseprint("(n**2 for n in it if n>5 if n<10)", mode='eval')       # Multiple␣
↪if clauses
  Expression(body=GeneratorExp(elt=BinOp(left=Name(id='n', ctx=Load()), op=Pow(),␣
↪right=Num(n=2)), generators=[
      comprehension(target=Name(id='n', ctx=Store()), iter=Name(id='it', ctx=Load()),␣
↪ifs=[
          Compare(left=Name(id='n', ctx=Load()), ops=[
              Gt(),
            ], comparators=[
              Num(n=5),
            ]),
          Compare(left=Name(id='n', ctx=Load()), ops=[
              Lt(),
            ], comparators=[
              Num(n=10),
            ]),
        ],
        is_async=0),
    ]))

  >>> parseprint(("async def f():"
                  "   return [i async for i in soc]")) # Async comprehension.
  Module(body=[
  AsyncFunctionDef(name='f', args=arguments(args=[], vararg=None, kwonlyargs=[], kw_
↪defaults=[], kwarg=None, defaults=[]), body=[
      Return(value=ListComp(elt=Name(id='i', ctx=Load()), generators=[
          comprehension(target=Name(id='i', ctx=Store()), iter=Name(id='soc',␣
↪ctx=Load()), ifs=[], is_async=1),
        ])),
    ], decorator_list=[], returns=None),
])
```

2.4 Statements

class Assign(*targets*, *value*, *type_comment*)

An assignment. `targets` is a list of nodes, and `value` is a single node. `type_comment` is optional. It is a string containing the PEP 484 type comment associated to the assignment.

```
>>> parseprint("a = 1 # type: int", type_comments=True)
Module(body=[
    Assign(targets=[
       Name(id='b', ctx=Store()),
     ], value=Num(n=1)), type_comment="int"
  ], type_ignores=[])
```

Multiple nodes in `targets` represents assigning the same value to each. Unpacking is represented by putting a *`Tuple`* or *`List`* within `targets`.

```
>>> parseprint("a = b = 1")      # Multiple assignment
Module(body=[
    Assign(targets=[
       Name(id='a', ctx=Store()),
       Name(id='b', ctx=Store()),
     ], value=Num(n=1)),
  ])
```

```
>>> parseprint("a,b = c")        # Unpacking
Module(body=[
    Assign(targets=[
        Tuple(elts=[
            Name(id='a', ctx=Store()),
            Name(id='b', ctx=Store()),
          ], ctx=Store()),
      ], value=Name(id='c', ctx=Load())),
  ])
```

class AnnAssign(*target*, *annotation*, *value*, *simple*)

New in version 3.6.

An assignment with a type annotation. `target` is a single node and can be a *`Name`*, a *`Attribute`* or a *`Subscript`*. `annotation` is the annotation, such as a *`Str`* or *`Name`* node. `value` is a single optional node. `simple` is a boolean integer set to True for a *`Name`* node in `target` that do not appear in between parenthesis and are hence pure names and not expressions.

```
>>> parseprint("c: int")
Module(body=[
    AnnAssign(target=Name(id='c', ctx=Store()),
              annotation=Name(id='int', ctx=Load()),
              value=None,
              simple=1),
  ])
```

```
>>> parseprint("(a): int = 1")  # Expression like name
Module(body=[
    AnnAssign(target=Name(id='a', ctx=Store()),
    annotation=Name(id='int', ctx=Load()),
    value=Num(n=1),
```

(continues on next page)

(continued from previous page)

```
    simple=0),
  ])
```

```
>>> parseprint("a.b: int")  # Attribute annotation
Module(body=[
    AnnAssign(target=Attribute(value=Name(id='a', ctx=Load()),
                               attr='b', ctx=Store()),
              annotation=Name(id='int', ctx=Load()),
              value=None,
              simple=0),
  ])
```

```
>>> parseprint("a[1]: int")  # Subscript annotation
Module(body=[
    AnnAssign(target=Subscript(value=Name(id='a', ctx=Load()),
                               slice=Index(value=Num(n=1)), ctx=Store()),
              annotation=Name(id='int', ctx=Load()),
              value=None,
              simple=0),
 ])
```

Changed in version 3.8: `type_comment` was introduced in Python 3.8

class AugAssign(*target*, *op*, *value*)
Augmented assignment, such as `a += 1`. In that example, `target` is a *Name* node for `a` (with the *Store* context), op is *Add*, and `value` is a *Num* node for 1. `target` can be *Name*, *Subscript* or *Attribute*, but not a *Tuple* or *List* (unlike the targets of *Assign*).

class Print(*dest*, *values*, *nl*)
Print statement, Python 2 only. `dest` is an optional destination (for `print >>dest`. `values` is a list of nodes. `nl` (newline) is True or False depending on whether there's a comma at the end of the statement.

class Raise(*exc*, *cause*)
Raising an exception, Python 3 syntax. `exc` is the exception object to be raised, normally a *Call* or *Name*, or `None` for a standalone `raise`. `cause` is the optional part for `y` in `raise x from y`.

In Python 2, the parameters are instead `type, inst, tback`, which correspond to the old `raise x, y, z` syntax.

class Assert(*test*, *msg*)
An assertion. `test` holds the condition, such as a *Compare* node. `msg` holds the failure message, normally a *Str* node.

class Delete(*targets*)
Represents a `del` statement. `targets` is a list of nodes, such as *Name*, *Attribute* or *Subscript* nodes.

class Pass
A `pass` statement.

Other statements which are only applicable inside functions or loops are described in other sections.

2.4.1 Imports

class Import(*names*)
An import statement. `names` is a list of *alias* nodes.

class ImportFrom(*module*, *names*, *level*)
Represents `from x import y`. `module` is a raw string of the 'from' name, without any leading dots, or `None` for statements such as `from . import foo`. `level` is an integer holding the level of the relative import (0 means absolute import).

class alias(*name*, *asname*)
Both parameters are raw strings of the names. `asname` can be `None` if the regular name is to be used.

```
>>> parseprint("from ..foo.bar import a as b, c")
Module(body=[
    ImportFrom(module='foo.bar', names=[
        alias(name='a', asname='b'),
        alias(name='c', asname=None),
      ], level=2),
  ])
```

2.5 Control flow

Note: Optional clauses such as `else` are stored as an empty list if they're not present.

class If(*test*, *body*, *orelse*)
An `if` statement. `test` holds a single node, such as a *Compare* node. `body` and `orelse` each hold a list of nodes.

`elif` clauses don't have a special representation in the AST, but rather appear as extra *If* nodes within the `orelse` section of the previous one.

class For(*target*, *iter*, *body*, *orelse*, *type_comment*)
A `for` loop. `target` holds the variable(s) the loop assigns to, as a single *Name*, *Tuple* or *List* node. `iter` holds the item to be looped over, again as a single node. `body` and `orelse` contain lists of nodes to execute. Those in `orelse` are executed if the loop finishes normally, rather than via a `break` statement. `type_comment` is optional. It is a string containing the PEP 484 type comment associated to for statement.

Changed in version 3.8: `type_comment` was introduced in Python 3.8

class While(*test*, *body*, *orelse*)
A `while` loop. `test` holds the condition, such as a *Compare* node.

class Break
class Continue
The `break` and `continue` statements.

```
In [2]: %%dump_ast
   ...: for a in b:
   ...:   if a > 5:
   ...:     break
   ...:   else:
   ...:     continue
   ...:
Module(body=[
    For(target=Name(id='a', ctx=Store()), iter=Name(id='b', ctx=Load()), body=[
        If(test=Compare(left=Name(id='a', ctx=Load()), ops=[
            Gt(),
          ], comparators=[
            Num(n=5),
```

(continues on next page)

(continued from previous page)

```
          ]), body=[
            Break(),
          ], orelse=[
            Continue(),
          ]),
      ], orelse=[]),
  ])
```

class Try(*body*, *handlers*, *orelse*, *finalbody*)
`try` blocks. All attributes are list of nodes to execute, except for `handlers`, which is a list of *`ExceptHandler`* nodes.

New in version 3.3.

class TryFinally(*body*, *finalbody*)
class TryExcept(*body*, *handlers*, *orelse*)
`try` blocks up to Python 3.2, inclusive. A `try` block with both `except` and `finally` clauses is parsed as a *`TryFinally`*, with the body containing a *`TryExcept`*.

class ExceptHandler(*type*, *name*, *body*)
A single `except` clause. `type` is the exception type it will match, typically a *`Name`* node (or `None` for a catch-all `except:` clause). `name` is a raw string for the name to hold the exception, or `None` if the clause doesn't have `as foo`. `body` is a list of nodes.

In Python 2, `name` was a *`Name`* node with `ctx=Store()`, instead of a raw string.

```
In [3]: %%dump_ast
   ...: try:
   ...:   a + 1
   ...: except TypeError:
   ...:   pass
   ...:
Module(body=[
   Try(body=[
       Expr(value=BinOp(left=Name(id='a', ctx=Load()), op=Add(), right=Num(n=1))),
     ], handlers=[
       ExceptHandler(type=Name(id='TypeError', ctx=Load()), name=None, body=[
           Pass(),
         ]),
     ], orelse=[], finalbody=[]),
 ])
```

class With(*items*, *body*, *type_comment*)
A `with` block. `items` is a list of *`withitem`* nodes representing the context managers, and `body` is the indented block inside the context. `type_comment` is optional. It is a string containing the PEP 484 type comment associated to the assignment (added in Python 3.8).

Changed in version 3.3: Previously, a *`With`* node had `context_expr` and `optional_vars` instead of `items`. Multiple contexts were represented by nesting a second *`With`* node as the only item in the `body` of the first.

Changed in version 3.8: `type_comment` was introduced in Python 3.8

class withitem(*context_expr*, *optional_vars*)
A single context manager in a `with` block. `context_expr` is the context manager, often a *`Call`* node. `optional_vars` is a *`Name`*, *`Tuple`* or *`List`* for the `as foo` part, or `None` if that isn't used.

```
In [3]: %%dump_ast
  ...: with a as b, c as d:
  ...:     do_things(b, d)
  ...:
Module(body=[
    With(items=[
        withitem(context_expr=Name(id='a', ctx=Load()), optional_vars=Name(id='b',␣
↪ctx=Store())),
        withitem(context_expr=Name(id='c', ctx=Load()), optional_vars=Name(id='d',␣
↪ctx=Store())),
      ], body=[
        Expr(value=Call(func=Name(id='do_things', ctx=Load()), args=[
            Name(id='b', ctx=Load()),
            Name(id='d', ctx=Load()),
          ], keywords=[], starargs=None, kwargs=None)),
      ]),
  ])
```

2.6 Function and class definitions

class FunctionDef(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)
A function definition.

- `name` is a raw string of the function name.
- `args` is a *`arguments`* node.
- `body` is the list of nodes inside the function.
- `decorator_list` is the list of decorators to be applied, stored outermost first (i.e. the first in the list will be applied last).
- `returns` is the return annotation (Python 3 only).
- `type_comment` is optional. It is a string containing the PEP 484 type comment of the function (added in Python 3.8)

Changed in version 3.8: `type_comment` was introduced in Python 3.8

class Lambda(*args*, *body*)
`lambda` is a minimal function definition that can be used inside an expression. Unlike *`FunctionDef`*, `body` holds a single node.

class arguments(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)
The arguments for a function. In **Python 3**:

- `args`, `posonlyargs` and `kwonlyargs` are lists of *`arg`* nodes.
- `vararg` and `kwarg` are single *`arg`* nodes, referring to the `*args, **kwargs` parameters.
- `kw_defaults` is a list of default values for keyword-only arguments. If one is `None`, the corresponding argument is required.
- `defaults` is a list of default values for arguments that can be passed positionally. If there are fewer defaults, they correspond to the last n arguments.

Changed in version 3.8: `posonlyargs` was introduced in Python 3.8

Changed in version 3.4: Up to Python 3.3, `vararg` and `kwarg` were raw strings of the argument names, and there were separate `varargannotation` and `kwargannotation` fields to hold their annotations.

Also, the order of the remaining parameters was different up to Python 3.3.

In **Python 2**, the attributes for keyword-only arguments are not needed.

class arg(*arg*, *annotation*, *type_comment*)
A single argument in a list; Python 3 only. `arg` is a raw string of the argument name, `annotation` is its annotation, such as a *Str* or *Name* node. `type_comment` is optional. It is a string containing the PEP 484 type comment of the argument.

In Python 2, arguments are instead represented as *Name* nodes, with `ctx=Param()`.

```
In [52]: %%dump_ast
   ....: @dec1
   ....: @dec2
   ....: def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
   ....:   pass
   ....:
Module(body=[
    FunctionDef(name='f', args=arguments(posonlyargs=[],
      args=[
        arg(arg='a', annotation=Str(s='annotation')),
        arg(arg='b', annotation=None),
        arg(arg='c', annotation=None),
      ], vararg=arg(arg='d', annotation=None), kwonlyargs=[
        arg(arg='e', annotation=None),
        arg(arg='f', annotation=None),
      ], kw_defaults=[
        None,
        Num(n=3),
      ], kwarg=arg(arg='g', annotation=None), defaults=[
        Num(n=1),
        Num(n=2),
      ]), body=[
        Pass(),
      ], decorator_list=[
        Name(id='dec1', ctx=Load()),
        Name(id='dec2', ctx=Load()),
      ], returns=Str(s='return annotation')),
  ])

.. versionchanged:: 3.8

   ``type_comment`` was introduced in Python 3.8
```

class Return(*value*)
A `return` statement.

class Yield(*value*)
class YieldFrom(*value*)
A `yield` or `yield from` expression. Because these are expressions, they must be wrapped in a *Expr* node if the value sent back is not used.

New in version 3.3: The *YieldFrom* node type.

class Global(*names*)
class Nonlocal(*names*)
`global` and `nonlocal` statements. `names` is a list of raw strings.

class ClassDef(*name*, *bases*, *keywords*, *starargs*, *kwargs*, *body*, *decorator_list*)
A class definition.

- `name` is a raw string for the class name
- `bases` is a list of nodes for explicitly specified base classes.
- `keywords` is a list of *`keyword`* nodes, principally for 'metaclass'. Other keywords will be passed to the metaclass, as per PEP-3115.
- `starargs` and `kwargs` are each a single node, as in a function call. starargs will be expanded to join the list of base classes, and kwargs will be passed to the metaclass. These are removed in Python 3.5 - see below for details.
- `body` is a list of nodes representing the code within the class definition.
- `decorator_list` is a list of nodes, as in *`FunctionDef`*.

```
In [59]: %%dump_ast
   ....: @dec1
   ....: @dec2
   ....: class foo(base1, base2, metaclass=meta):
   ....:   pass
   ....:
Module(body=[
    ClassDef(name='foo', bases=[
        Name(id='base1', ctx=Load()),
        Name(id='base2', ctx=Load()),
      ], keyword=
        keyword(arg='metaclass', value=Name(id='meta', ctx=Load())),
      ], starargs=None,      # gone in 3.5
         kwargs=None,        # gone in 3.5
         body=[
        Pass(),
      ], decorator_list=[
        Name(id='dec1', ctx=Load()),
        Name(id='dec2', ctx=Load()),
      ]),
  ])
```

2.7 Async and await

New in version 3.5: All of these nodes were added. See the What's New notes on the new syntax.

class AsyncFunctionDef (*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)
An `async def` function definition. Has the same fields as *`FunctionDef`*.

class Await (*value*)
An `await` expression. `value` is what it waits for. Only valid in the body of an *`AsyncFunctionDef`*.

```
In [2]: %%dump_ast
  ...: async def f():
  ...:   await g()
  ...:
Module(body=[
   AsyncFunctionDef(name='f', args=arguments(args=[], vararg=None, kwonlyargs=[], kw_
↪defaults=[], kwarg=None, defaults=[]), body=[
       Expr(value=Await(value=Call(func=Name(id='g', ctx=Load()), args=[],␣
↪keywords=[]))),
     ], decorator_list=[], returns=None),
])
```

class AsyncFor(*target*, *iter*, *body*, *orelse*)
class AsyncWith(*items*, *body*)
`async for` loops and `async with` context managers. They have the same fields as *For* and *With*, respectively. Only valid in the body of an *AsyncFunctionDef*.

2.8 Top level nodes

Those nodes are at the top-level of the AST. The manner by which you obtain the AST determine the top-level node used.

class Module(*stmt* body*, *type_ignore *type_ignores*)
The root of the AST for code parsed using the *exec* mode. The `body` attribute is a list of nodes. `type_ignores` is a list of `TypeIgnore` indicating the lines on which `type: ignore` comments are present. If type comments are not stored in the ast it is an empty list.

Changed in version 3.8: `type_ignores` was introduced in Python 3.8 and is mandatory when manually creating a *Module*

class Interactive(*stmt* body*)
The root of the AST for single statements or expressions parsed using the *single* mode. The `body` attribute is a list of nodes.

class Expression(*expr body*)
The root of the AST for single expressions parsed using the *eval* mode. The `body` attribute is a single node, such as `ast.Call` or `ast.BinOp`. This is different from `ast.Expr`, which holds an expression within an AST.

CHAPTER 3

Working on the Tree

`ast.NodeVisitor` is the primary tool for 'scanning' the tree. To use it, subclass it and override methods `visit_Foo`, corresponding to the node classes (see *Meet the Nodes*).

For example, this visitor will print the names of any functions defined in the given code, including methods and functions defined within other functions:

```
class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        print(node.name)
        self.generic_visit(node)

FuncLister().visit(tree)
```

Note: If you want child nodes to be visited, remember to call `self.generic_visit(node)` in the methods you override.

Alternatively, you can run through a list of all the nodes in the tree using `ast.walk()`. There are no guarantees about the order in which nodes will appear. The following example again prints the names of any functions defined within the given code:

```
for node in ast.walk(tree):
    if isinstance(node, ast.FunctionDef):
        print(node.name)
```

You can also get the direct children of a node, using `ast.iter_child_nodes()`. Remember that many nodes have children in several sections: for example, an `If` has a node in the `test` field, and list of nodes in `body` and `orelse`. `ast.iter_child_nodes()` will go through all of these.

Finally, you can navigate directly, using the attributes of the nodes. For example, if you want to get the last node within a function's body, use `node.body[-1]`. Of course, all the normal Python tools for iterating and indexing work. In particular, `isinstance()` is very useful for checking what nodes are.

3.1 Inspecting nodes

The `ast` module has a couple of functions for inspecting nodes:

- `ast.iter_fields()` iterates over the fields defined for a node.
- `ast.get_docstring()` gets the docstring of a `FunctionDef`, `ClassDef` or `Module` node.
- `ast.dump()` returns a string showing the node and any children. See also the pretty printer used in this guide.

3.2 Modifying the tree

The key tool is `ast.NodeTransformer`. Like `ast.NodeVisitor`, you subclass this and override `visit_Foo` methods. The method should return the original node, a replacement node, or `None` to remove that node from the tree.

The `ast` module docs have this example, which rewrites name lookups, so `foo` becomes `data['foo']`:

```
class RewriteName(ast.NodeTransformer):

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

tree = RewriteName().visit(tree)
```

When replacing a node, the new node doesn't automatically have the `lineno` and `col_offset` parameters. The example above doesn't deal with this completely: it copies the location to the `Subscript` node, but not to any of the newly created children of that node. See *Fixing locations*.

Be careful when removing nodes. You can quite easily remove a node from a required field, such as the `test` field of an `If` node. Python won't complain about the invalid AST until you try to `compile()` it, when a `TypeError` is raised.

CHAPTER 4

Examples of working with ASTs

Working versions of these examples are in the examples directory of the source repository.

4.1 Wrapping integers

In Python code, `1/3` would normally be evaluated to a floating-point number, that can never be exactly one third. Mathematical software, like SymPy or Sage, often wants to use exact fractions instead. One way to make `1/3` produce an exact fraction is to wrap the integer literals `1` and `3` in a class:

```
class IntegerWrapper(ast.NodeTransformer):
    """Wraps all integers in a call to Integer()"""
    def visit_Num(self, node):
        if isinstance(node.n, int):
            return ast.Call(func=ast.Name(id='Integer', ctx=ast.Load()),
                            args=[node], keywords=[])
        return node

tree = ast.parse("1/3")
tree = IntegerWrapper().visit(tree)
# Add lineno & col_offset to the nodes we created
ast.fix_missing_locations(tree)

# The tree is now equivalent to Integer(1)/Integer(3)
# We would also need to define the Integer class and its __truediv__ method.
```

See wrap_integers.py for a working demonstration.

4.2 Simple test framework

These two manipulations let you write test scripts as a simple series of `assert` statements. First, we need to run the statements one by one, so execution doesn't stop at the first test failure:

```
tree = ast.parse(code)
lines = [None] + code.splitlines()  # None at [0] so we can index lines from 1
test_namespace = {}

for node in tree.body:
    wrapper = ast.Module(body=[node])
    try:
        co = compile(wrapper, "<ast>", 'exec')
        exec(co, test_namespace)
    except AssertionError as e:
        print("Assertion failed on line", node.lineno, ":")
        print(lines[node.lineno])
        # If the error has a message, show it.
        if e.args:
            print(e)
        print()
```

Next, we transform `assert a == b` into a function call `assert_equal(a, b)`, which can give more information about the failure. We could turn many other assertions into similar function calls.

```
class AssertCmpTransformer(ast.NodeTransformer):
    def visit_Assert(self, node):
        if isinstance(node.test, ast.Compare) and \
                len(node.test.ops) == 1 and \
                isinstance(node.test.ops[0], ast.Eq):
            call = ast.Call(func=ast.Name(id='assert_equal', ctx=ast.Load()),
                            args=[node.test.left, node.test.comparators[0]],
                            keywords=[])
            # Wrap the call in an Expr node, because the return value isn't used.
            newnode = ast.Expr(value=call)
            ast.copy_location(newnode, node)
            ast.fix_missing_locations(newnode)
            return newnode

        # Remember to return the original node if we don't want to change it.
        return node
```

See test_framework/run.py for a working demonstration of both parts.

4.3 Real projects

- pytest uses the AST to produce useful error messages when assertions fail.
- astsearch lets you search through Python code based on semantics rather than text, e.g. to find every `+= 1` in your code.
- astpath is a more powerful search tool using XPath expressions on Python code.
- bellybutton is a linter designed to be readily customised.

See also:

Python AST explorer Web-based AST viewer: paste some code in and see the AST

Thonny A Python IDE with AST explorer built in (*Main menu => View => AST*)

showast An IPython extension to show ASTs in Jupyter notebooks

Instrumenting the AST Using AST tools to assess code coverage

CHAPTER 5

Indices and tables

- genindex
- search

Index