9 The AST

AST stands for *Abstract Syntax Tree*. It is a tree representation of the abstract structure of the source code of any programming language, including Python. Python as its own AST that is built upon parsing a Python source file.

This area of Python is not heavily documented, and not easy to deal with at first glance. Still, its is very interesting to know and understand some deeper construction of Python as a programming language to masterize its usage.

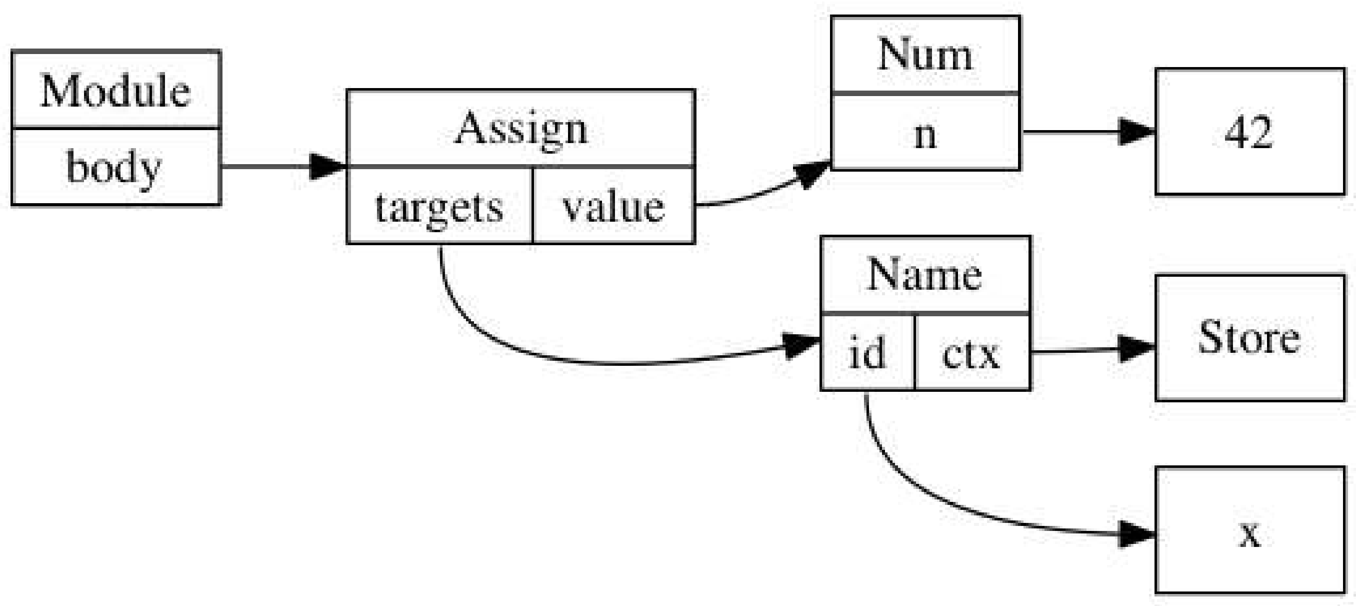
The easiest way to have a view of what the Python AST looks like is to parse a Python code and dumps the generated AST. To do that, the Python `ast` module provides everything you need for.

Example 9.1 Parsing Python code to AST

```
>>> import ast
>>> ast.parse
<function parse at 0x7f062731d950>
>>> ast.parse("x = 42")
<_ast.Module object at 0x7f0628a5ad10>
>>> ast.dump(ast.parse("x = 42"))
"Module(body=[Assign(targets=[Name(id='x', ctx=Store())], value=Num(n=42)) ↩
   ])"
```

The `ast.parse` function returns a `_ast.Module` object that is the root of the tree.

The tree can be entirely dumped using the `ast.dump` module, and in this case is the following:



An AST construction always starts with a root element, which is usually an `ast.Module` object. This object contains a list of statements or expressions to evaluate in its `body` attribute. It usually represents the content of a file.

As you can guess, the `ast.Assign` object represents an assignment, that is mapped to the `=` sign in the Python syntax. Assign has a list of targets, and a value it assignates to it. The list of target in this case consists of one object, `ast.Name`, which represents a variable named `x`. The value is a number with value being 42.

This AST can be passed to Python to be compiled and then evaluated. The `compile` function provided as a Python built-in allows that.

```
>>> compile(ast.parse("x = 42"), '<input>', 'exec')
<code object <module> at 0x111b3b0, file "<input>", line 1>
>>> eval(compile(ast.parse("x = 42"), '<input>', 'exec'))
>>> x
42
```

An abstract syntax tree can be built manually using the classes provided in the `ast` module. Obviously, this is a very long way to write Python code, not a method I would recommend! But it's still interesting to use.

Let's write a good old "Hello world!" in Python using the AST.

Example 9.2 Hello world using Python AST

```
>>> hello_world = ast.Str(s='hello world!', lineno=1, col_offset=1)
>>> print_call = ast.Print(values=[hello_world], lineno=1, col_offset=1, nl ↩
   =True)
>>> module = ast.Module(body=[print_call])
>>> code = compile(module, '', 'exec')
>>> eval(code)
hello world!
```

Note

`lineno` and `col_offset` represents the line number and column offset of the source code that has been used to generate the AST. This doesn't have much sense to set them in this context since we are not parsing any source file, but it's useful to find back the position of the code that generated this AST. It's for example used by Python when generating backtraces. Anyway, Python refused to compile any AST object that doesn't provide this information, this is why we pass it fake values of 1 here. The `ast.fix_missing_locations()` function can fix it for you by setting the missing values to the ones set on the parent node.

The whole list of objects that are available in the AST is easily available by reading the `_ast` module documentation (note the underscore).

The first two categories you should consider are statement and expressions. Statements cover types like *assert*, *assign* (=), augmented assigned (+=, /=, etc), *global*, *def*, *if*, *return*, *for*, *class*, *pass*, *import*, etc. They all inherit from `ast.stmt`. Expressions cover types like *lambda*, *number*, *yield*, *name* (variable), *compare* or *call*. They all inherit from `ast.expr`.

There's also a few other categories, such as `ast.operator` defining standard operator such as *add* (+), *div* (/), *right shift* (>>), etc, or `ast.cmpop` defining comparisons operator.

You can easily imagine that it is then possible to leverage this AST to construct a compiler that would parse strings and generate code by building a Python AST. This is exactly what led to the Hy project discussed in Section 9.1.

In case you need to walk through your tree, the `ast.walk` function will help you with that. But the `ast` module also provides `NodeTransformer`, a class that can be subclassed to walk an AST to modify some nodes. It's therefore easy to use it to change code dynamically.

Example 9.3 Changing all binary operation to addition

```
import ast


class ReplaceBinOp(ast.NodeTransformer):
    """Replace operation by addition in binary operation"""
    def visit_BinOp(self, node):
        return ast.BinOp(left=node.left,
                         op=ast.Add(),
                         right=node.right)


tree = ast.parse("x = 1/3")
ast.fix_missing_locations(tree)
eval(compile(tree, '', 'exec'))
print(ast.dump(tree))
print(x)
tree = ReplaceBinOp().visit(tree)
ast.fix_missing_locations(tree)
print(ast.dump(tree))
eval(compile(tree, '', 'exec'))
```

```
print(x)
```

Which executes to the following:

```
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Div(), right=Num(n=3)))])
0.3333333333333333
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=BinOp(left=Num(n=1), op=Add(), right=Num(n=3)))])
4
```

Tip

If you need to evaluate a string of Python that should return a simple data type, you can use `ast.literal_eval`. Contrary to `eval`, it disallows the input string to execute any code. It's a safer alternative to `eval`.

9.1 Hy

Now that you know about the AST, you can easily dream of creating a new syntax for Python that you would parse and compile down to a standard Python AST. The Hy programming language is doing exactly that. It is a Lisp dialect that parses a Lisp like language and converts it to regular Python AST. It is therefore fully compatible with the Python ecosystem. You could compare it to what Clojure is to Java. Hy could deserve a book for itself, so we will only fly over it in this section.

If you already wrote Lisp [1], the Hy syntax will really look familiar. Once installed, launching the hy interpreter will give you a standard REPL prompt where you can start interact with the interpreter.

[1] If not, you should consider it.

```
% hy
hy 0.9.10
=> (+ 1 1)
2
```

For those not familiar with the Lisp syntax, the parentheses denote a list, the first element is a function, and the rest of the list are the arguments. Here the code is equivalent to Python `1 + 1`.

Most constructs are mapped from Python directly, such as function definition. Setting a variable relies on the `setv` function.

```
=> (defn hello [name]
...   (print "Hello world!")
...   (print (% "Nice to meet you %s" name)))
=> (hello "jd")
Hello world!
Nice to meet you jd
```

Internally, *Hy* parses the code that is provided and compiles it down to Python AST. Luckily, Lisp is an easy to parse tree, as each pair of parentheses represents a node of the list tree. All is needed to be done is to convert this Lisp tree to a Python abstract syntax tree.

Class definition is supported through the `defclass` construct, that is inspired from CLOS [2].

```
(defclass A [object]
  [[x 42]
   [y (fn [self value]
        (+ self.x value))]])
```

[2]Common Lisp Object System

This defines a class named *A*, inheriting from `object`, with a class attribute `x` whose value is 42 and a method `y` that returns the `x` attribute plus the value passed as argument.

What's really wonderful, is that you can import **any Python library** directly into Hy and use it with no penalty.

```
=> (import uuid)
=> (uuid.uuid4)
UUID('f823a749-a65a-4a62-b853-2687c69d0e1e')
=> (str (uuid.uuid4))
'4efa60f2-23a4-4fc1-8134-00f5c271f809'
```

Hy also has more advanced construct and macros. If you ever wanted to have a case or switch statement in Python, admire what `cond` can do for you:

```
(cond
 ((> somevar 50)
  (print "That variable is too big!"))
 ((< somevar 10)
  (print "That variable is too small!"))
 (true
  (print "That variable is jusssst right!")))
```

Hy is a very nice project that allows you to jump into Lisp world without leaving your comfort zone too far behind you, as you are still writing Python. The `hy2py` tool can even show you what your Hy code would look like once translated into Python [3].

9.2 Interview with Paul Tagliamonte

Paul is a Debian developer, who's working at Sunlight Foundation. He created Hy in 2013 and, as a Lisp lover, I joined him in this fabulous adventure some time later.

[3]Though it has some restrictions.

Why did you create Hy in the first place?

Initially, I created Hy following a conversation about how someone should write a Lisp that compiles to Python rather than Java's JVM (Clojure). A few short days later, and I had the first version of Hy – something which resembled a lisp, and even worked like a proper lisp, but it was slow. I mean, really slow. It took about an order of magnitude slower than native Python, since the Lisp runtime itself was implemented in Python.

Frustrated, I almost gave up, only to be pushed forward by a coworker the promise of using AST to implement the runtime, rather than implement the runtime in Python. This insane idea started to really spark the entire project. This set in shortly before the holidays in 2012, leading me to spend my entire break from work hacking on Hy. A week or so later, and I ended up with something that resembled the current Hy codebase quite closely – most Hy devs would even know their way around the compiler.

Just after getting enough working to implement a basic Flask app, I gave a talk at Boston Python about this project, and the reception was incredibly warm – so warm, in fact, that I'd started to view Hy as a good way to teach people about Python internals, such as how the REPL works [4], PEP 302 import hooks, and Python AST – a good introduction to the concept of code that writes code.

After the talk, I was a bit disappointed in a few sections, so I rewrote chunks of the compiler to fix some philosophical issues in the process, leading us to the current iteration of the codebase – which has stood up quite well!

[4] *code.InteractiveConsole*

In addition, Hy (the Language) is a good way to get people to understand how to read Lisp, since they can get comfortable with s-expressions in an environment they know (even using libraries they have lying around), easing the transition to other ("real") Lisps, such as Common Lisp, Scheme or Clojure, as well as experiment with new ideas (such as macro systems, homoiconicity, and working without the concept of a statement).

How did you find out about using the AST correctly? What are the tips and tricks, advice you can give to people looking at it?

Python's AST is quite interesting. It's not quite private (in fact, it's explicitly not private), but it's also not a public interface either. No stability is guaranteed from version to version – in fact, there are some rather annoying differences between Python 2 and 3, and even within different Python 3 releases. In addition, different implementations may interpret the AST differently, or even have a unique AST. Nothing says Jython, PyPy, or CPython must deal with Python AST in the same way.

For instance, CPython can deal with slightly out of order AST entries (by the *lineno* and *col_offset*), whereas PyPy will throw an assertion error. While sometimes annoying, the AST is generally sane. It's not impossible to build AST that works on a vast number of Python instances. With a conditional or two, it's only mildly annoying to create AST that works on CPython 2.6 through 3.3 and PyPy, making this tool quite handy.

The AST is extremely under-documented, so most knowledge comes from reverse engineering generated AST. By writing up simple Python scripts, one can use something similar to `import ast;ast.dump(ast.parse("print foo"))` to generate equivalent AST to help with the task. With a bit of guesswork, and some persistence, it's not untenable to build up a basic understanding this way.

At some point, I'll take on the task of documenting my understanding of the AST module, but I find writing code is the best way to learn the AST.

What's the current status, and future goals of Hy?

Hy is currently in development. It has a few subtle issues that need to be addressed, and fixing the bugs to make Hy virtually indistinguishable from any other LISP-1 variant. This is a monumental task, but it's one that it's ripe for hacking.

I'm also interested in keeping Hy efficient, in so far as it can be.

I hope, in the long run, that Hy will become a sort of teaching tool – one way to explain some of the concepts that are quite foreign to even experienced Pythonistas. I hope it also proves interesting enough to Pythonistas that they take an interest in these tools at our disposal, and continue pushing the bounds of what I think Hy is.

My hope is that people see Hy for what it is – an amazing teaching tool. A way to get people interested in Common Lisp, Clojure or Scheme. I want people to go home and read about why Lisp variants do things the way they do, and how they can borrow this philosophy in their day-to-day coding.

How interoperable with Python is Hy? What about code distribution and packaging?

Amazingly interoperable. Stunningly interoperable, really. So well, in fact, that *pdb* can properly debug Hy without any changes at all. To really drive this point home, I've written Flask apps, Django apps and modules of all sorts. Python can import Python, Hy can import Hy, Hy can import Python and Python can import Hy. This is what really makes Hy unique – even variants like Clojure can't do this, the interop is purely unidirectional (Clojure can import Java, but Java has one hell of a time importing Clojure). This was done to really bring home how powerful these tools we have are.

Hy works by translating Hy code (in s-expressions) into Python AST almost directly. This compilation step means the generated bytecode is fairly sane stuff (so much so that debugging Hy by looking at Python source generated from Python AST is a good way of tracking down pesky AST errors), which means Python has a very hard time of even telling the module isn't written in Python at all.

Common Lisp-isms, such as `*earmuffs*` or `using-dashes` are fully supported by translating them to a Python equivalent (in this case, `*earmuffs*` becomes `EARMUFFS`, and `using-dashes` becomes `using_dashes`), which means Python doesn't have a hard time of using them at all.

Ensuring that we have really good interoperability is one of our highest priorities, so if you see any bugs – file them!

What are the upside and downside of choosing Hy over Python?

This is an interesting question. I'm quite partial, so take this with a grain of salt!

Hy outshines Python in a few special ways because we've taken a bit of effort to smooth behavior over Python versions to allow the new Python 3 future happen sooner. This was done by doing things like using future division in Python 2, and ensuring the syntax is normalized between the two versions.

In addition, Hy has something Python has a very hard time with (even with the outstanding AST module), which is a full macro system. Macros are very special functions that alter the code during it's compile step – not unlike having `ast.NodeVisitor` as a first-class function of the language. This leads to easy creation of new domain-specific languages, which is composed of the base language (in this case, Hy / Python), with the addition of many macros which allow uniquely expressive and succinct code.

Often times, clever DSLs can replace languages designed to perform this role, such as Lua.

As for downsides, what gives Hy it's power can also hurt it. Not technically, but socially. Hy, by virtue of being a Lisp written in s-expressions, suffers from the stigma of being hard to learn, read or maintain. People might be averse to working on projects using Hy due to the fear of Hy being extremely complex.

Hy is the Lisp everyone loves to hate – Python folks tend to not enjoy its syntax, and Lispers tend to avoid Hy due to, well, being Python. Hy uses Python objects directly, so the behavior of fundamental objects can sometimes be surprising to the seasoned Lisper.

Hopefully people will look past it's syntax and consider using it for a project to expand one's horizons, and explore parts of Python previously untouched.