Practical Python parsing with the `ast` standard module

Domenico Nucera,
Independent Contractor

Backstory

I discovered `ast` during a lunch break at PyCon Italia 2023

Used it for personal projects, sometimes it helped me at work

Agenda

Why

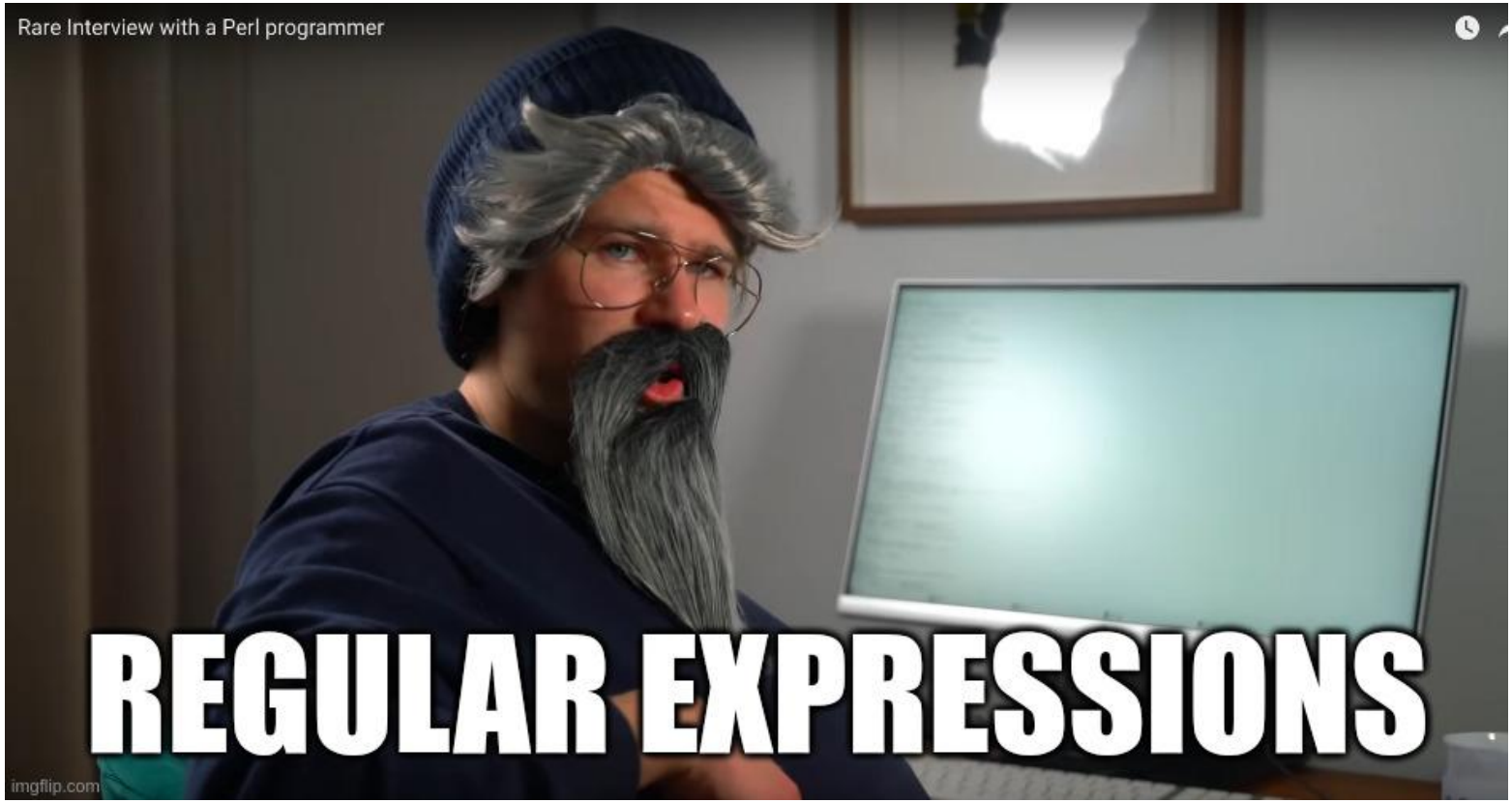
AST - tree and nodes

AST - walking our tree

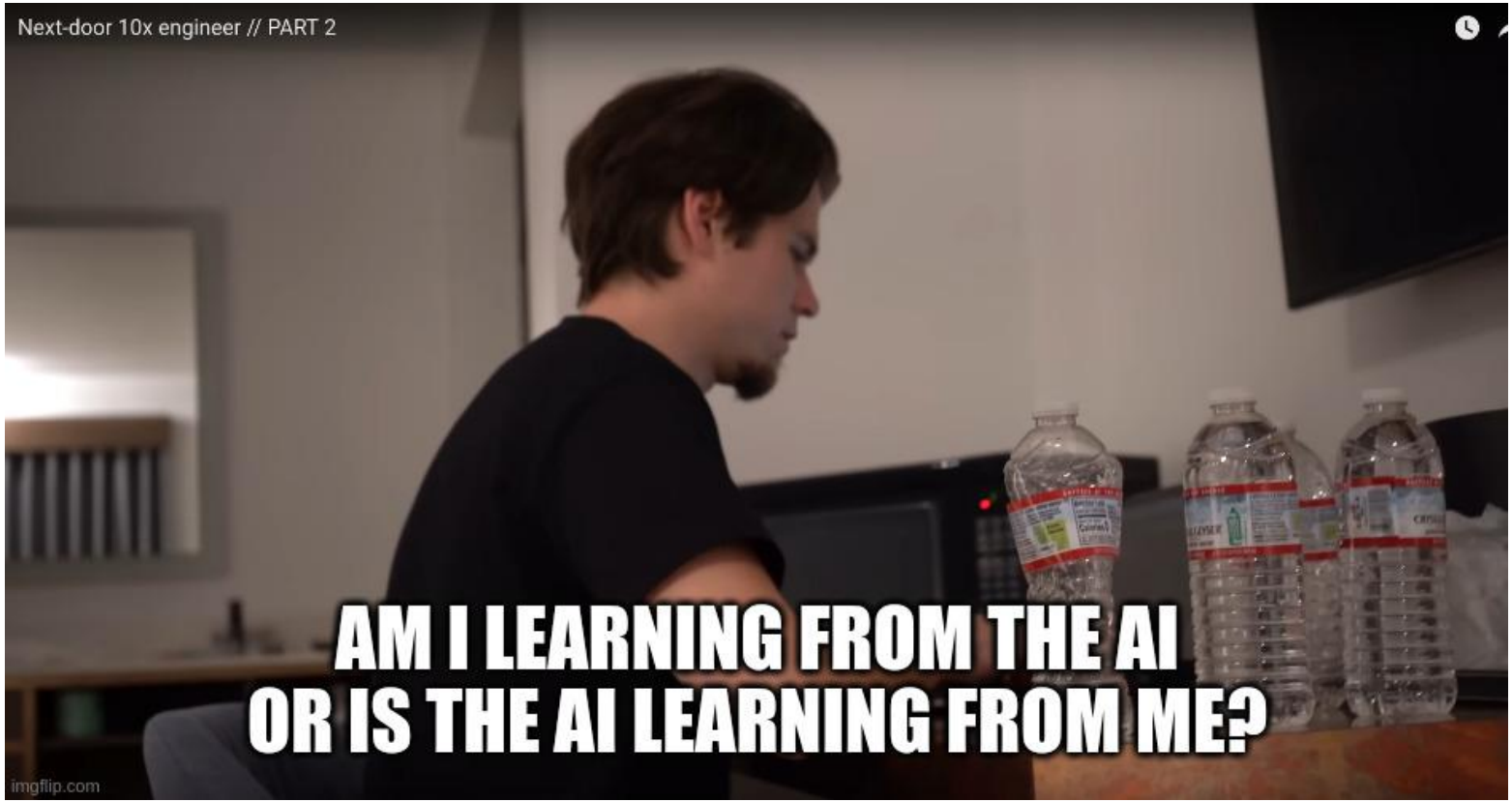
Demo time

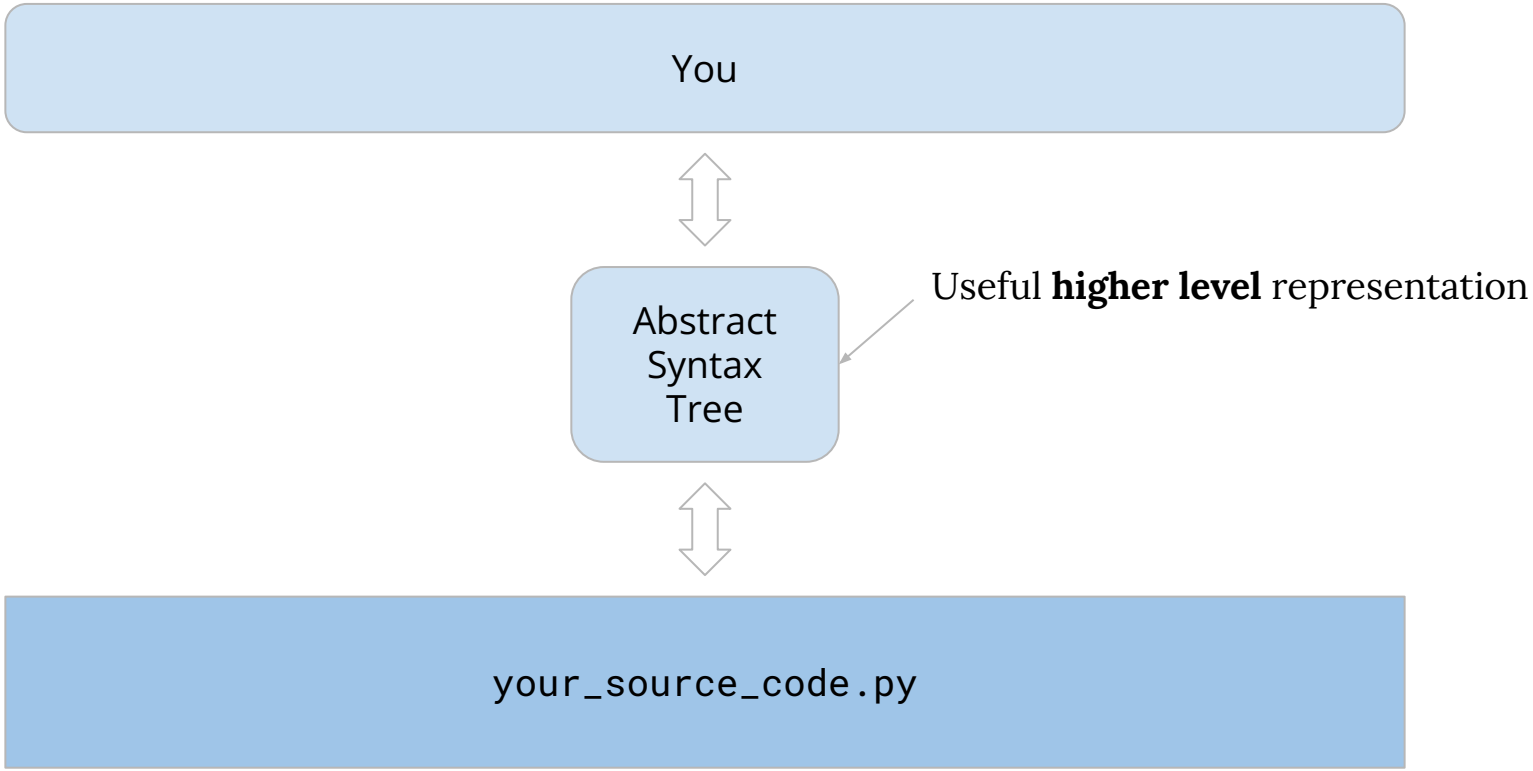
Conclusion

Imagine your program shall process some python code


Rare Interview with a Perl programmer
REGULAR EXPRESSIONS
imgflip.com


Interview with Senior C++ Developer
WE REWRITE EVERYTHING OURSELVES
imgflip.com


Next-door 10x engineer // PART 2
AM I LEARNING FROM THE AI
OR IS THE AI LEARNING FROM ME?
imgflip.com


You
Abstract Syntax Tree
Useful **higher level** representation
your_source_code.py

psycopg

psycopg3 Sponsors

Automatic async to sync code conversion

Posted by Daniele Varrazzo on 2024-09-23
*Tagged as **psycopg3**, **development***

Psycopg 3 provides both a sync and an async Python interface: for each object used to perform I/O operations, such as Connection, Cursor, there is an async counterpart: AsyncConnection, AsyncCursor, with an intuitive interface: just add the right `async` or `await` keyword where needed:

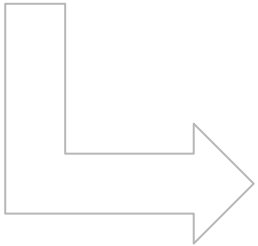
```
# Familiar sync code
conn = psycopg.Connection.connect("")
cur = conn.execute("select now()")
print(cur.fetchone()[0])

# Similar async code
aconn = await psycopg.AsyncConnection.connect("")
acur = await aconn.execute("select now()")
print((await acur.fetchone())[0])
```

https://www.psycopg.org/articles/2024/09/23/async-to-sync/

How do we obtain this AST?

```
import ast
tree = ast.parse(‘print(“hello world”)’)
ast.dump(tree)
```



```
Module(
  body=[
    Expr(
    value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
        Constant(value='hello world')]))])
```

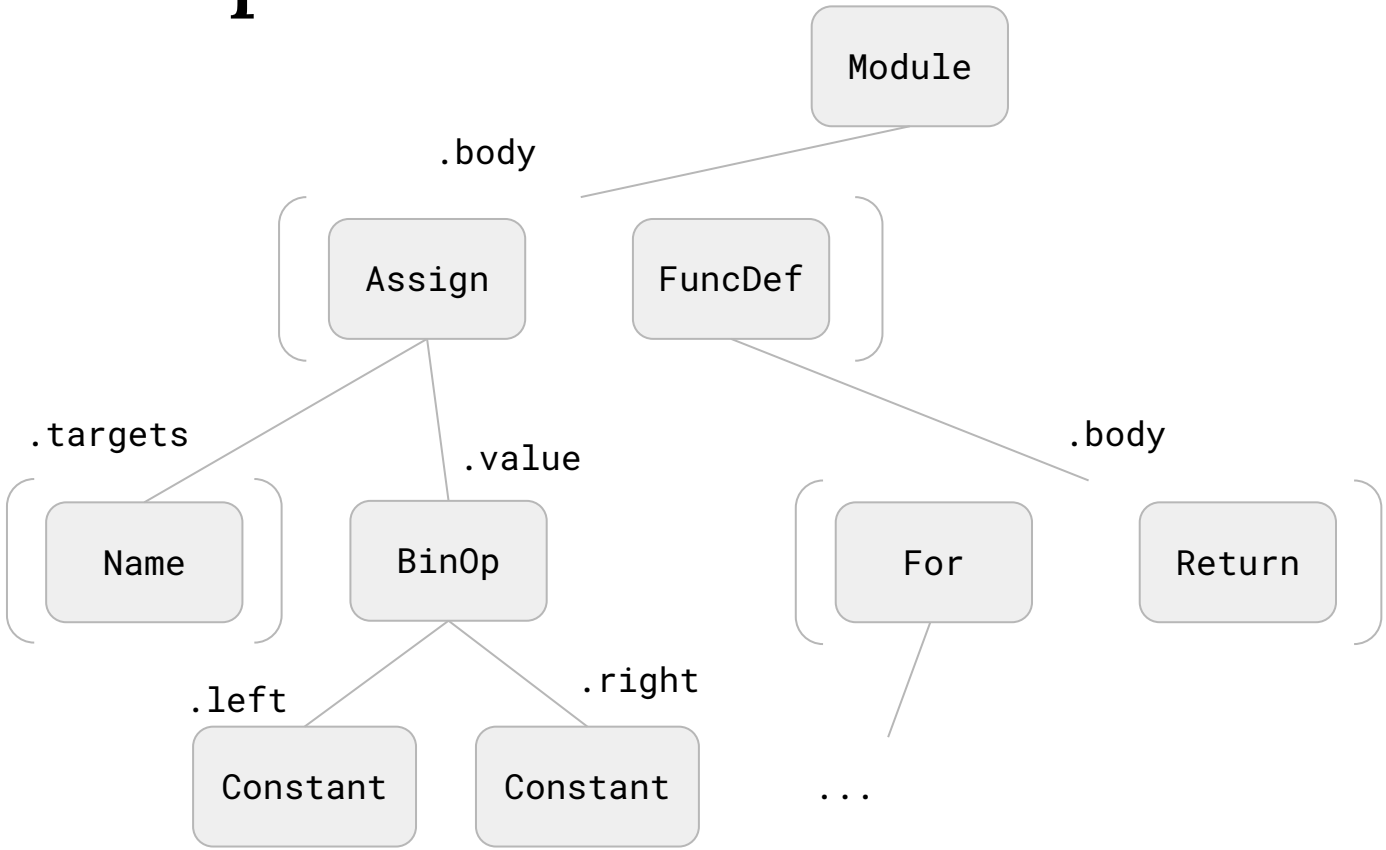
Nodes of different types

Grammatical expression

$\Leftrightarrow$

ast Node Type

“Polymorphic” tree



Nodes of **many** different types

Things in common

Most nodes possess the attributes `lineno`, `end_lineno`, `col_offset` and `end_col_offset` to indicate the node's position in the code.

Functions, classes, loops, ifs, ecc have a `body` attribute, a list of its "sub-statements"

.body attribute

```
def whatever_function() -> int:
    a = 0
    b = 0
    for i in range(66):
        a += 1
        b += a
    return b
```

.body

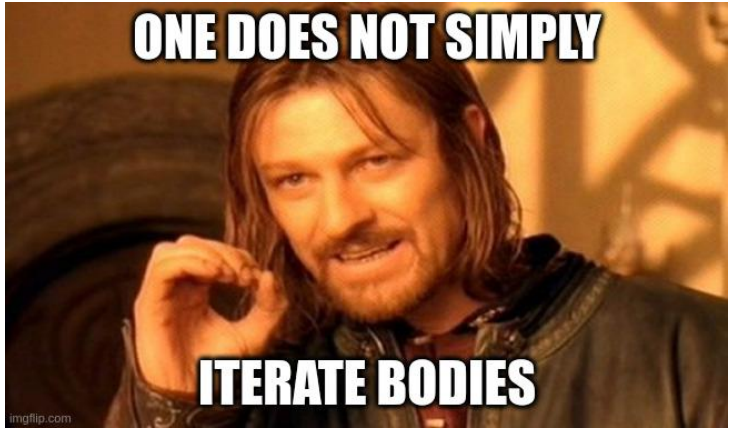
.body

.body attribute

```
def whatever_function() -> int:
    a = 0
    b = 0
    for i in range(66):
        a += 1
        b += a
    return b
```

.body

.body



.handlers

.orelse

.finalbody

Examples of nodes

they will be useful with visitors and transformers!

Node examples

`class MyClass:`	->	**`ast.ClassDef`**
`def f() -> None:`	->	**`ast.FunctionDef`**

Node examples

`return None` -> **`ast.Return`**

`print()` -> **`ast.Call`**

Data structures

`[1,2,3,4]` -> **`ast.List`**

`{‘one’:1, ‘two’:2}` -> **`ast.Dict`**

Grammatical expression

$\Leftrightarrow$

ast Node Type

$\Leftrightarrow$

subtleties

Imports

```
import json                 ->  ast.Import

from json import loads      ->  ast.ImportFrom
```

Async

```
def f():                    ->  ast.FunctionDef

async def f():              ->  ast.AsyncFunctionDef
```

Assignments

```
foo = 1

foo: int = 1
```

Assignments

```
foo = 1          ->  ast.Assign

foo: int = 1     ->  ast.AnnAssign
```

TryStar

```
ast.Try
```

```
ast.TryStar
```

Type annotations

Type annotations are **reported** in the AST

For example an `ast.FunctionDef` will have the `returns` attribute to indicate its **return type annotation**

Traversing our tree

`ast.walk`

`ast.walk` allows to **iterate over all the descendants** of a given node.

It does so in **no particular order**

NodeVisitor

```
class MyNodeVisitor(ast.NodeVisitor):


    def visit_FunctionDef(self, node):
        if node.returns is not None:
            print(f'{node.name} -> {node.returns.id}')
        self.generic_visit(node)


MyNodeVisitor().visit( tree )
```

Write to this to let subnodes get visited!

NodeTransformer

```
class MyNodeTransformer(ast.NodeTransformer):


     def visit_ListComp(self, node):
          self.generic_visit(node)
          return ast.SetComp(
               elt=node.elt,
               generators=node.generators
          )


tree = MyNodeTransformer().visit( tree )
```

NodeTransformer

```
class MyNodeTransformer(ast.NodeTransformer):

    def visit_ListComp(self, node):
        self.generic_visit(node)
        return ast.SetComp(
            elt=node.elt,
            generators=node.generators
        )

tree = MyNodeTransformer().visit( tree )
```

Other helpers

`get_source_segment()` -> gives you back a node's code

`get_docstring()` -> gives you the node's docstring

`iter_child_nodes()` -> iterates on the direct children of a node

`dump()` -> prints the ast

`unparse()` -> generates code given an ast

Demo time

There are worlds beyond

You can use `ast.compile()` to obtain bytecode.

Ecosystem of tooling built around `ast`:

https://github.com/gyermolenko/awesome-python-ast

https://www.youtube.com/watch?v=Yq3wTWkoaYY

Personal opinions

Rarely useful… but when it is… a **game changer**, it gives you **great powers**.

Personal hint: if you think it could serve you, **try it**! Easier than expected

Links to the resources

Official docs - https://docs.python.org/3/library/ast.html

Green Tree Snakes - https://greentreesnakes.readthedocs.io/en/latest/

psycopg3 article - https://www.psycopg.org/articles/2024/09/23/async-to-sync/

https://github.com/gyermolenko/awesome-python-ast

Memes at the beginning - https://www.youtube.com/@programmersarealsohuman5909

Thanks for the attention