Supercharge your Python library using `ast` parsing

Presentation Materials

PyCon US 2025

Pittsburgh, PA

Adam Glustein

Code from Point72/csp

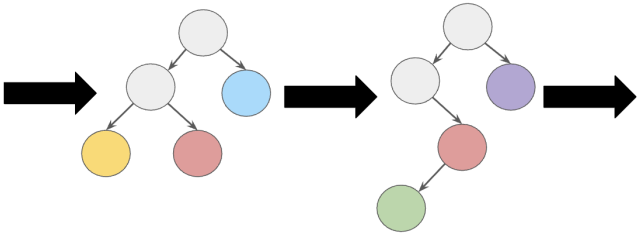
Main idea

- Performant Python libraries are usually implemented in **compiled** languages (C/C++/Rust)

- Between the **user's** Python code and the **library's** underlying implementation, we can transform the `ast` to squeeze out even more performance in the "hot path"

Main idea

```python
import mylibrary

@mylibrary_decorator
def user_function(x):
    ...do something...
    mylibrary.func(x)
    return something
```



```c
#include <mylibrary.h>

static
PyObject * impl()
{
    ...do something...
    return result;
}
```

User Python code | **Abstract syntax tree (AST)** | **Modified AST** | **Compiled implementation**

These are *practical* optimizations

- Strategies presented are taken from **csp**, an open-source Python stream processing library by Point72
- Links to the code are provided in case you don't believe me!

Agenda

Background

1. What is an abstract syntax tree and how can we modify the *ast*?
2. Brief overview of computation graphs and CSP

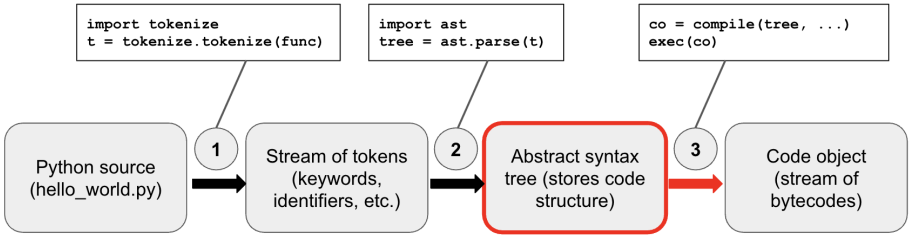
The fun stuff

1. Transforming functions to generators in the AST
2. Reducing member function call overhead with AST transformation
3. Fast local access of C/C++ data using the AST

A bit about myself

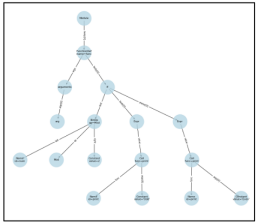
- Quant Developer at **Point72** in New York
- I help maintain the `csp` project
- We do a lot of cool stuff with the *ast*!

How Python code is executed




```
def func(num):
  if num % 2:
    print('Odd')
  else:
    print('Even')
```

```
NAME: 'def'
NAME: 'func'
OP: '('
NAME: 'num'
...
NAME: 'if'
NAME: 'num'
OP: '%'
NUMBER: '2'
...
```

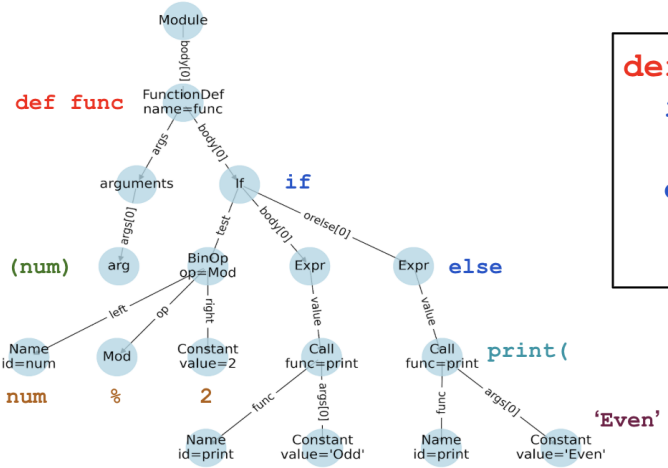


```
RESUME   0
LOAD_FAST 0 (num)
LOAD_CONST 1 (2)
BINARY_OP 6 (%)
...
LOAD_GLOBAL 1 (NULL + print)
LOAD_CONST  2 ('Odd number')
PRECALL 1
            ...
RETURN_VALUE
```

The Abstract Syntax Tree

- Represents the **structure** of a program
- *Abstracts* away unimportant information from the raw tokens (parentheses, newlines etc.)
- Last intermediate representation before bytecode generation
- Create the tree object in Python using `ast.parse`

The Abstract Syntax Tree




```
def func(num):
  if num % 2:
    print('Odd')
  else:
    print('Even')
```

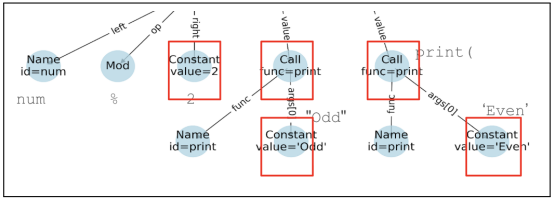
Let's visit an ast

- Subclass `ast.NodeVisitor` to traverse an AST

```python
import ast

class MyVisitor(ast.NodeVisitor):
    def visit_Call(self, node):
        if isinstance(node.func, ast.Name):
            print(f"Function call: {node.func.id}")
        self.generic_visit(node) # continue traversal

    def visit_Constant(self, node):
        print(f"Constant with value: {node.value}")
        self.generic_visit(node) # continue traversal
```

Let's visit an ast

```python
import inspect

def func(num):
    if num % 2:
        print("Odd")
    else:
        print("Even")

src = inspect.getsource(func)
tree = ast.parse(src)
MyVisitor().visit(tree)
```



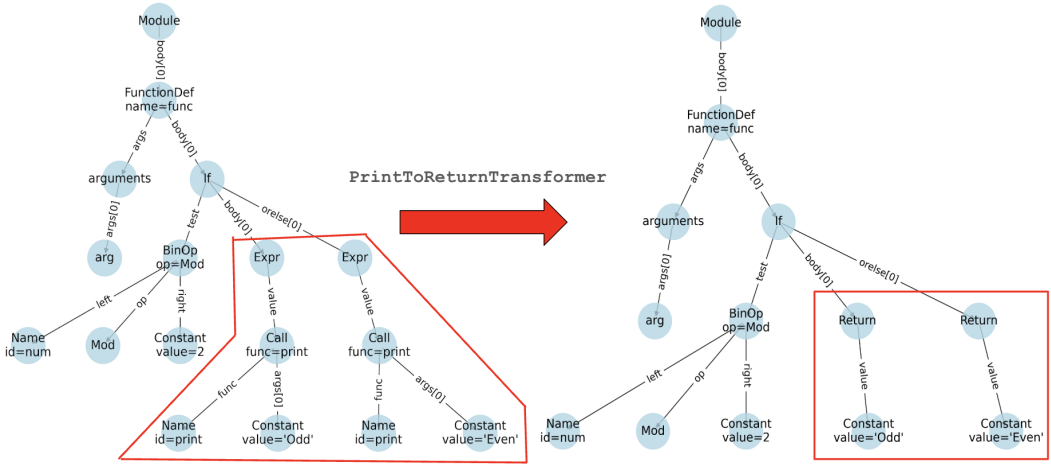

```
Constant with value:  2
Function call:  print
Constant with value:  Odd
Function call:  print
Constant with value:  Even
```

Let's transform an ast

- Subclass `ast.NodeTransformer` to transform an AST

```
import ast

class PrintToReturnTransformer(ast.NodeTransformer):
    def visit_Expr(self, node):
        # Check if this expr contains a print(...)
        if isinstance(node.value, ast.Call)
            and node.value.func.id == "print":
                # replace print with return
                return ast.Return(value=node.value.args[0])
        # Continue normally for other functions
        return self.generic_visit(node)
```

Let's transform an ast



Let's transform an ast

```
tree = ast.parse(func_code)
new_tree = PrintToReturnTransformer().visit(tree)

# Fix line numbers and other details
ast.fix_missing_locations(new_tree)

# Compile and execute
namespace = {}
co = compile(new_tree, "<ast>", mode="exec")
exec(co, namespace)

# Test the new function
func = namespace["func"]
func(5) # returns "Odd"
```

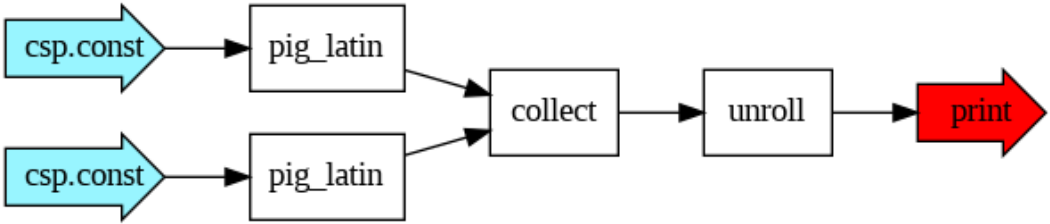
Key points to remember about the AST

- AST is an **abstract** representation of code
- Intermediate stage between Python source code and bytecode instructions
- We can use standard `ast` library to **transform** an AST

Next, we're going to look at something that *seems* entirely unrelated...

Computation graphs

- Directed, acyclic graph (DAG) that executes some task
- Functions → nodes, inputs → sources, outputs → sinks
- Used in many common Python libraries (torch, tensorflow, dask, airflow, etc.)



Computation graph that transforms words to pig latin

A (very) brief overview of CSP

- CSP = Composable Stream Processing
- Computation graph for **real-time streaming** data
  - Financial markets, climate analysis, transportation/supply-chain management, etc.
- Users create **nodes**, compose them into **graphs**, then compose those into larger graphs
- Engine code written in **C++** and exposed via C API

How CSP is implemented

- Users write nodes in Python that manipulate **time series** data
- Nodes are invoked when their inputs **"tick"**
- Nodes call `csp.output` to **"tick out"** a value
- `csp.node` and time series (`csp.ts`) correspond to objects in C++
- Runtime lives in C++, including the values of all `csp.ts` objects

@csp.node

- Functions that execute as part of a graph
- Inputs and outputs are **time-series** data (`csp.ts[T]`)
- **Stateful** and **atomic** units of computation

```python
import csp

@csp.node
def sum(input: csp.ts[int]) -> csp.ts[int]:
    with csp.state():
        s_sum = 0 # initial state

    if csp.ticked(input): # "ticked" = x has a new value
        s_sum += input
        csp.output(s_sum) # "tick" out an output
```

A closer look at the sum node

- Under the hood, there are four different AST transformations we'll look at

```
@csp.node
def sum(input: csp.ts[int]) -> csp.ts[int]:
    with csp.state():
        s_sum = 0 # initial state

    if csp.ticked(input):
        s_sum += input
        csp.output(s_sum)
```

1. Transformed to generator
2. Member function substitution
3. Member function substitution
4. Fast local variable access

1) Making nodes stateful functions

```
with csp.state():
        s_sum = 0 # initial state
```

- Nodes hold **state** between cycles
- **Generators** are useful constructs for maintaining state between cycles

- Transform nodes to generators using ast transformation:
  - Wrap function body with `ast.While`
  - `ast.Return` → `ast.Yield`

2) Reducing member function call overhead: no arguments

```
if csp.ticked(input): # input.ticked()
```

- `csp.ticked`: whether an input has a new value this cycle
- AST logic translates `ticked` to a **member function** on the input (i.e. `input.ticked()`) that is defined in C++
- `csp.ticked` is called *very* often, so how can we optimize it?

2) Reducing member function call overhead: no arguments

- `PyTypeObject` in C/C++ defines commonly called functions (`tp_init`, `tp_repr` etc.)
- We can substitute the `ticked` call for a type method that avoids lookup of named attribute
- AST transformation in csp allows us to hide this implementation from users

`obj.ticked()`

```
0 LOAD_FAST                0 (obj)
2 LOAD_ATTR                0 (ticked)
4 CALL_FUNCTION            0
6 RETURN_VALUE
```

`+obj`

```
0 LOAD_FAST                0 (obj)
2 UNARY_POSITIVE
4 RETURN_VALUE
```

3) Reducing member function call overhead: with arguments

```
csp.output(s_sum) # <out_ts>.output(s_sum)
```

- Same situation as before, except we have an argument
- We transform output to a binary add (`ast.BinOp`)

`obj.output(1)`

```
LOAD_FAST     0 (obj)
LOAD_ATTR     1 (output + NULL|self)
LOAD_CONST    1 (1)
CALL          1
RETURN_VALUE
```

`obj+1`

```
LOAD_FAST     0 (obj)
LOAD_CONST    1 (1)
BINARY_OP     0 (+)
RETURN_VALUE
```

3) Reducing member function call overhead: with arguments

- Simple microbenchmark to show performance improvement

```
static PyObject* MyClass_output(PyObject* self, PyObject* other) {
    Py_INCREF(other);
    return other;
}

// add to method table as normal member function
static PyMethodDef MyClass_methods[] = {
    {"output", (PyCFunction) MyClass_output, METH_VARARGS, ""},
    ...
};

// also define the same method as BINARY_ADD
static PyNumberMethods MyClass_as_number = {
    ( binaryfunc ) MyClass_output,
    ...
};
```

3) Reducing member function call overhead: with arguments

```
IPython 8.12.2 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from mymodule import MyClass

In [2]: obj = MyClass()

In [3]: %timeit obj+1
18 ns ± 1.36 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)

In [4]: %timeit obj.output(1)
34.5 ns ± 0.176 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Python Version	`obj.output(1)`	`obj+1`	**Improvement**
3.8.12	34.5 ns	18 ns	1.92x
3.11.12	30.6 ns	16.4 ns	1.87x
3.13.2	30.4 ns	19.8 ns	1.54x

4) Fast local variable access of C/C++ data

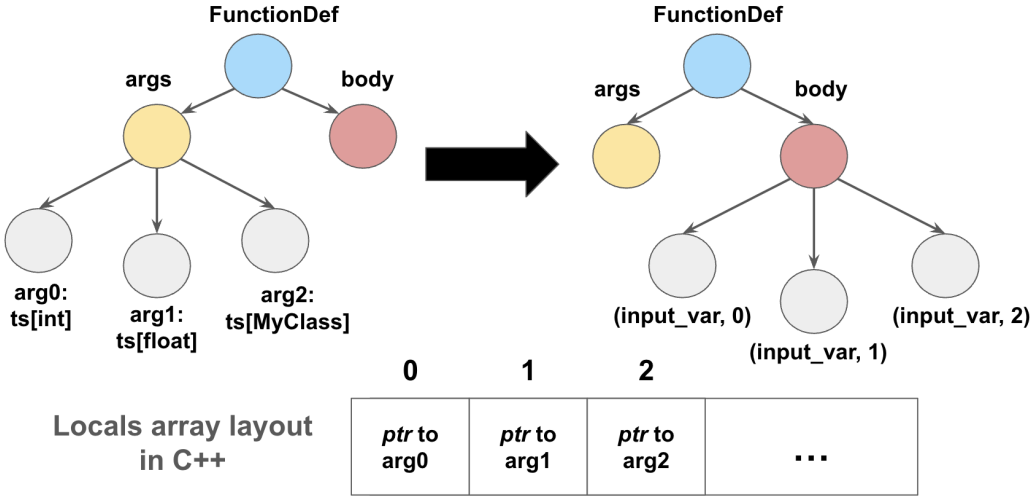
```
s_sum += input
```

- Value of all `csp.ts` objects live in C++ (i.e. `csp.ts[int]` stored as `int64_t` member on input object)
- Users need to access those values in their Python nodes
- This needs to be *as fast as possible*!

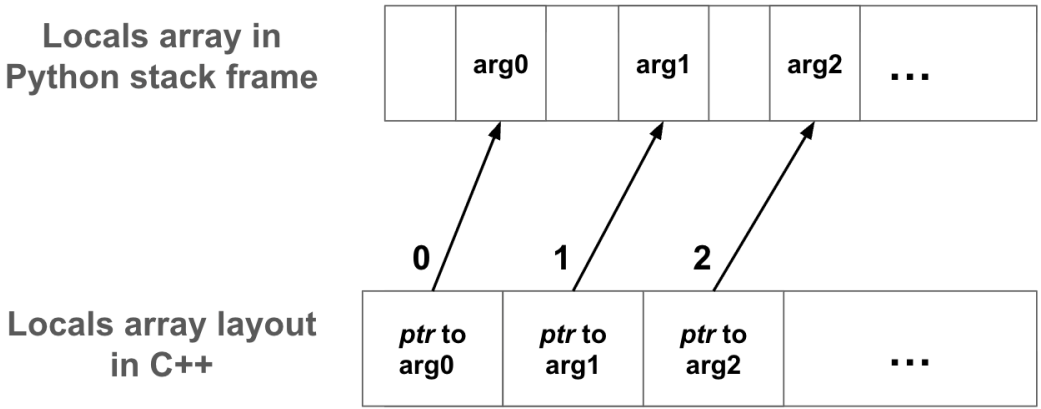
4) Fast local variable access of C/C++ data

- Take a variable defined in C++ and store it directly in the **fast locals array**
- We grab pointers to the input variables on the stack and store those in C++
- Use the `ast` to index each node's inputs, then store the variable with its stack index in an `ast.Tuple`
- `input` $\rightarrow$ `("input_var", idx)`

4) Fast local variable access of C/C++ data



4) Fast local variable access of C/C++ data



4) Fast local variable access of inputs

- Then, when we invoke the node at **runtime** we get `LOAD_FAST` on each `csp.ts`

```
     46 CALL_INTRINSIC_1          5 (INTRINSIC_UNARY_POSITIVE)
     48 POP_JUMP_IF_FALSE        10 (to 70)

15   50 LOAD_FAST                 4 (s_sum)
     52 LOAD_FAST                 3 (input)
     54 BINARY_OP                13 (+=)
     58 STORE_FAST                4 (s_sum)

16   60 LOAD_FAST                 2 (#outp_0)
     62 LOAD_FAST                 4 (s_sum)
     64 BINARY_OP                 0 (+)
```

The final product

```
@csp.node
def sum(input: csp.ts[int])
    -> csp.ts[int]:
    with csp.state():
        s_sum = 0 # initial state

    if csp.ticked(input):
        s_sum += input
        csp.output(s_sum)
```


```
def sum():
    #inp_0 = 'input_proxy', 0
    #outp_0 = 'output_proxy', 0
    input = 'input_var', 0
    yield
    del input
    s_sum = 0
    while True:
        yield
        if +#inp_0:
            s_sum += input
            #outp_0 + s_sum
```

Conclusion

- Transforming the `ast` is a powerful (but seldom used) strategy to boost the performance of Python libraries that interface with compiled code
- Modify a user's Python code to best fit the C/C++ implementation without compromising the API
- Specific techniques shown here (from csp) run in critical, high-performance production code