Automated Python Refactoring Powered by LibCST

Jimmy Lai https://github.com/jimmylai/talks

Instagram Server Framework
02/18/2020

Python @ Instagram

- Instagram Server:
  - 3 million+ lines of code

- Patterns:
  - Bad pattern copy-paste
  - Dead code

- Lint warnings/errors are numerous and annoying
- It's hard to:
  - deprecate legacy APIs
  - fix existing quality issues
  - refactor

Key Questions

1 Can we find code patterns easily?

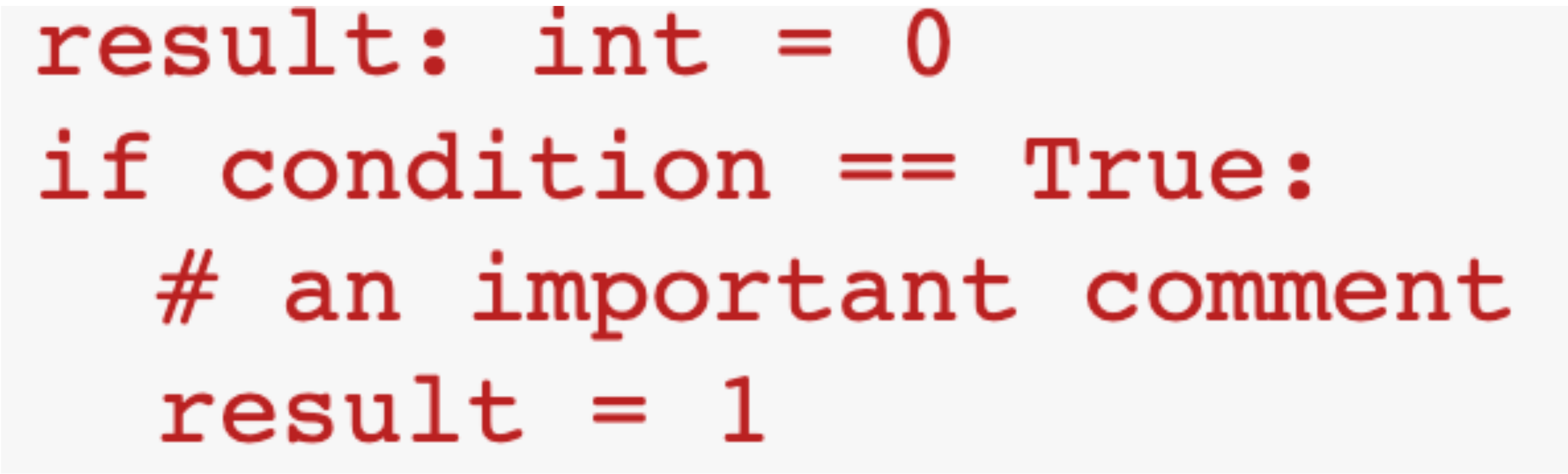
2 Can we fix problems automatically?

3 Can we build tools to help developers to write better code?

Comparison: Is or ==

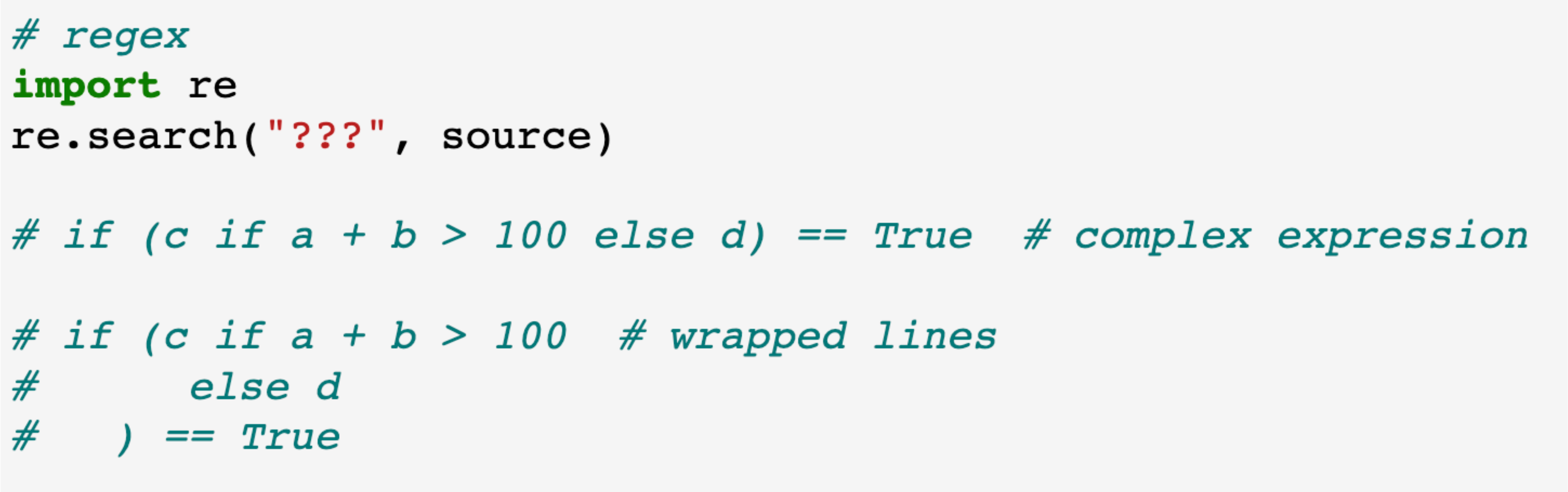
An Python convention example

- [PEP8] "is" compares identity and "==" compares value.
- [Flake8] [E712] Use is boolean value True/False.
- [Flake8] [F632] Use == for str, bytes, int liberals.



```
result: int = 0
if condition == True:
  # an important comment
  result = 1
```

- Let's build something to identify the bad patterns.
  - Regex：too hard to make the regex pattern right.



```
# regex
import re
re.search("???", source)

# if (c if a + b > 100 else d) == True  # complex expression

# if (c if a + b > 100  # wrapped lines
#         else d
#    ) == True
```

Codemod Roundtrip

1 Source Code

2 Structured Data

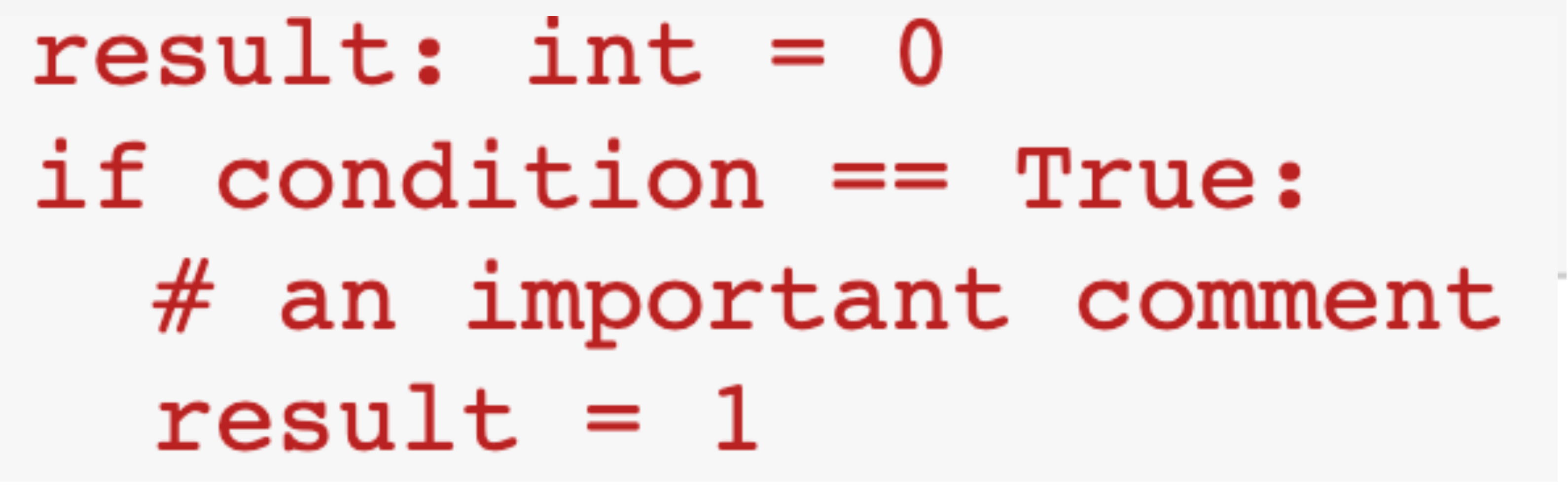
3 Traversal and Modification

4 Modified Source Code

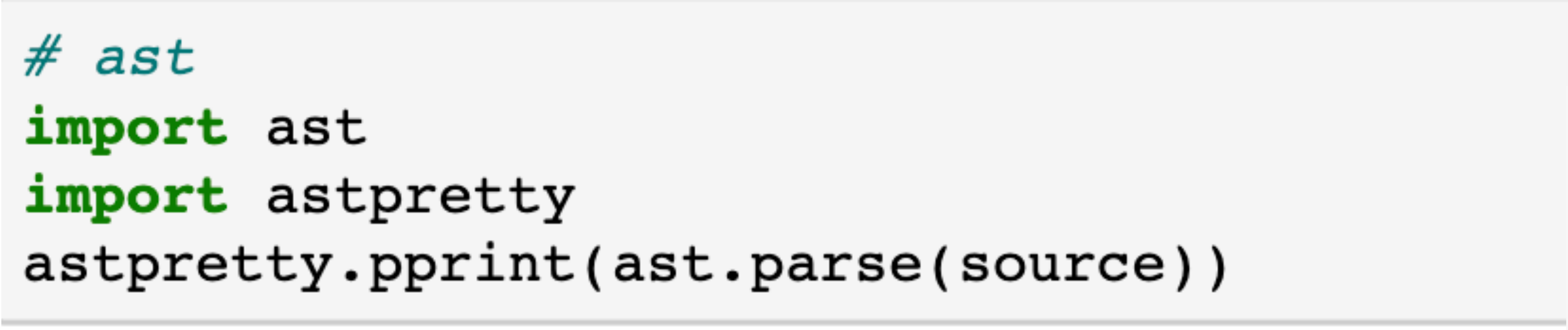
AST

Abstract Syntax Tree

- Missing formatting information:
    - comments
    - comma, space, newline



```
result: int = 0
if condition == True:
  # an important comment
  result = 1
```



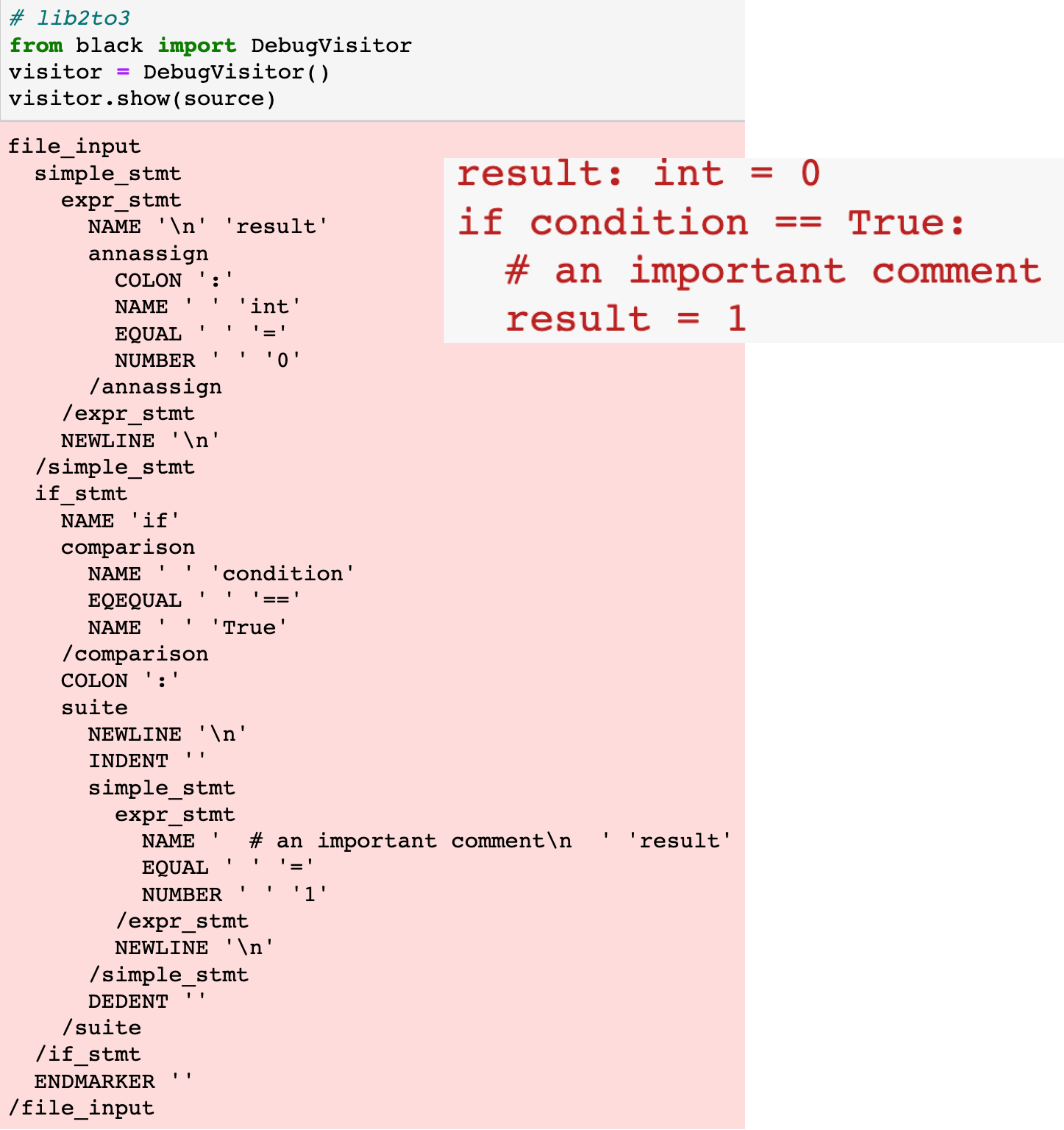
```
# ast
import ast
import astpretty
astpretty.pprint(ast.parse(source))
```

```
Module(
    body=[
        AnnAssign(
            lineno=2,
            col_offset=0,
            target=Name(lineno=2, col_offset=0, id='result', ctx=Store()),
            annotation=Name(lineno=2, col_offset=8, id='int', ctx=Load()),
            value=Num(lineno=2, col_offset=14, n=0),
            simple=1,
        ),
        If(
            lineno=3,
            col_offset=0,
            test=Compare(
                lineno=3,
                col_offset=3,
                left=Name(lineno=3, col_offset=3, id='condition', ctx=Load()),
                ops=[Eq()],
                comparators=[NameConstant(lineno=3, col_offset=16, value=True)],
            ),
            body=[
                Assign(
                    lineno=5,
                    col_offset=2,
                    targets=[Name(lineno=5, col_offset=2, id='result', ctx=Store())],
                    value=Num(lineno=5, col_offset=11, n=1),
                ),
            ],
            orelse=[],
        ),
    ],
)
```

lib2to3

Concrete Syntax Tree

- Not very easy to traverse



```
# lib2to3
from black import DebugVisitor
visitor = DebugVisitor()
visitor.show(source)
```

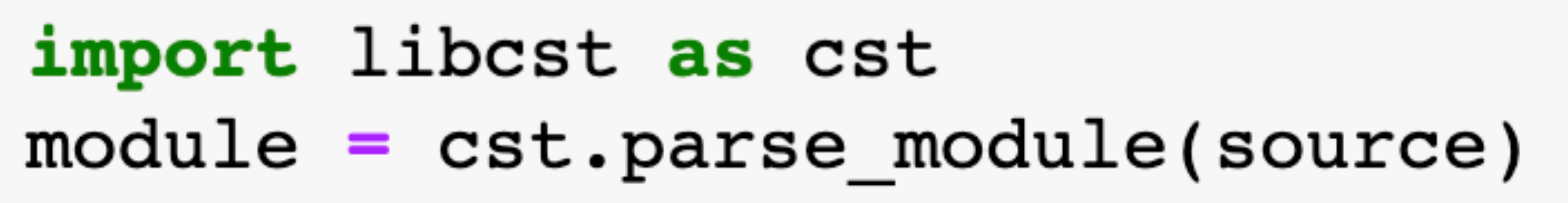
```
file_input
  simple_stmt
    expr_stmt
      NAME '\n' 'result'
      annassign
        COLON ':'
        NAME ' ' 'int'
        EQUAL ' ' '='
        NUMBER ' ' '0'
      /annassign
    /expr_stmt
    NEWLINE '\n'
  /simple_stmt
  if_stmt
    NAME 'if'
    comparison
      NAME ' ' 'condition'
      EQEQUAL ' ' '=='
      NAME ' ' 'True'
    /comparison
    COLON ':'
    suite
      NEWLINE '\n'
      INDENT ''
      simple_stmt
        expr_stmt
          NAME '  # an important comment\n  ' 'result'
          EQUAL ' ' '='
          NUMBER ' ' '1'
        /expr_stmt
        NEWLINE '\n'
      /simple_stmt
      DEDENT ''
    /suite
  /if_stmt
  ENDMARKER ''
/file_input
```

```
result: int = 0
if condition == True:
  # an important comment
  result = 1
```

LibCST

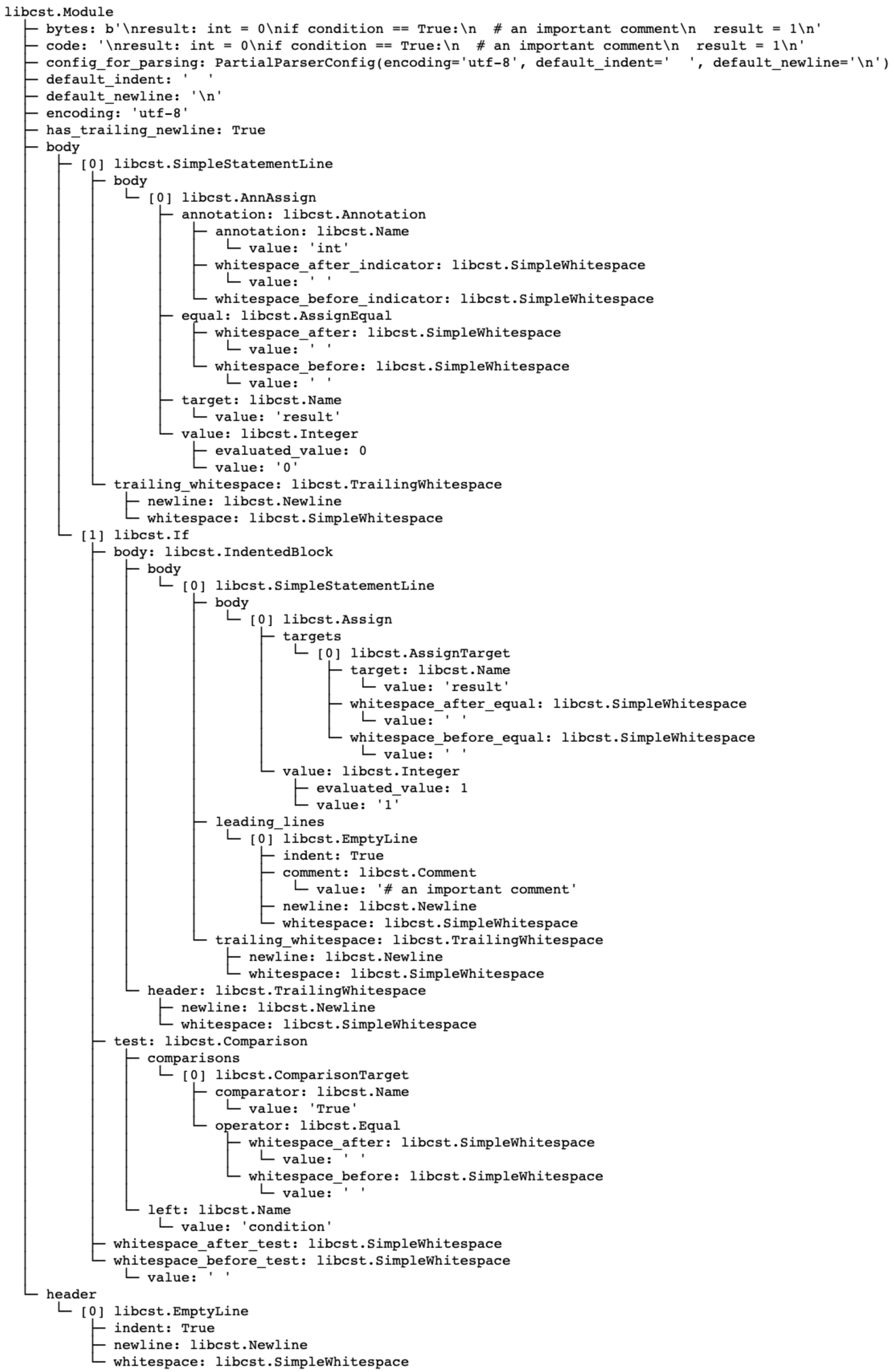
https://github.com/Instagram/LibCST/

- provides a concrete syntax tree (CST) looks like and feels like AST



```python
import libcst as cst
module = cst.parse_module(source)
```

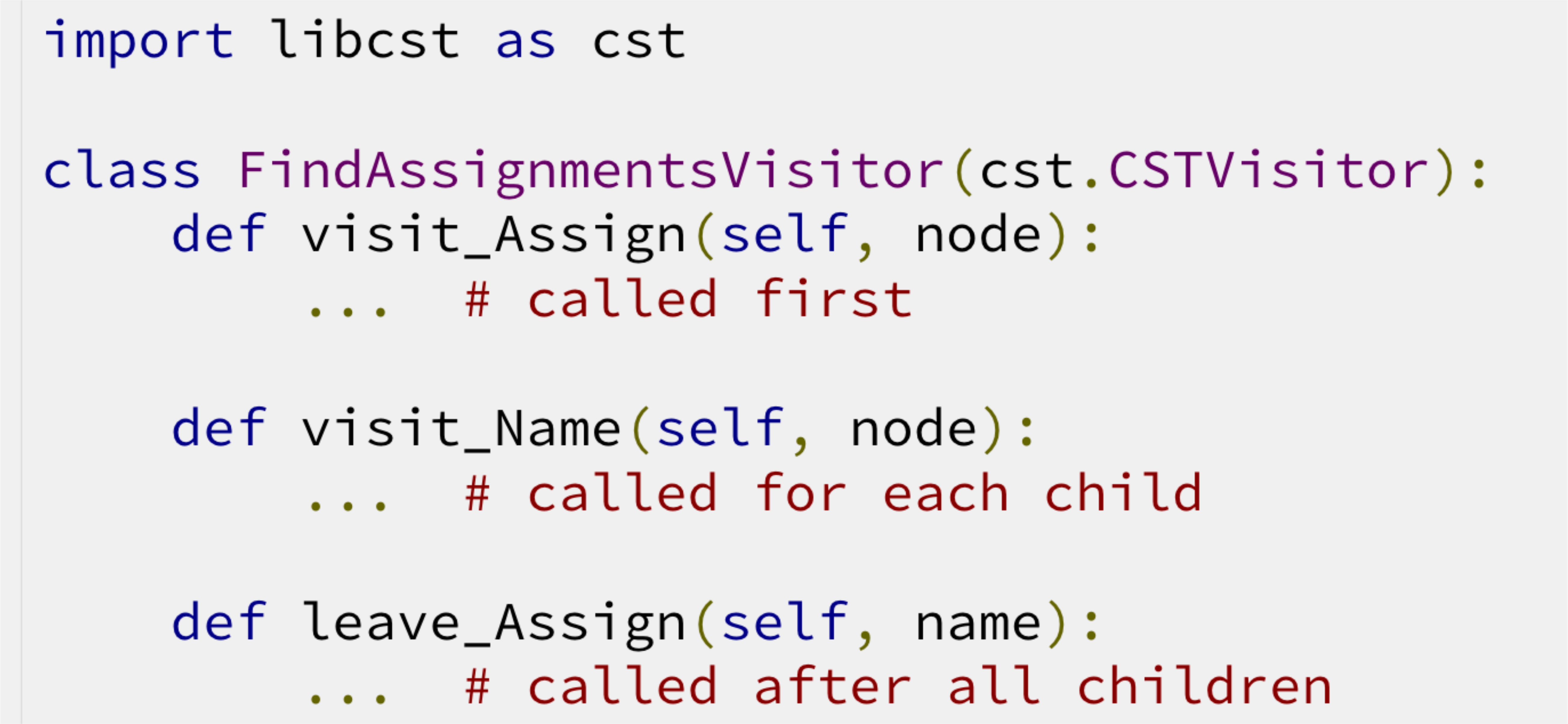
- Thanks to astboom provides pretty print for LibCST



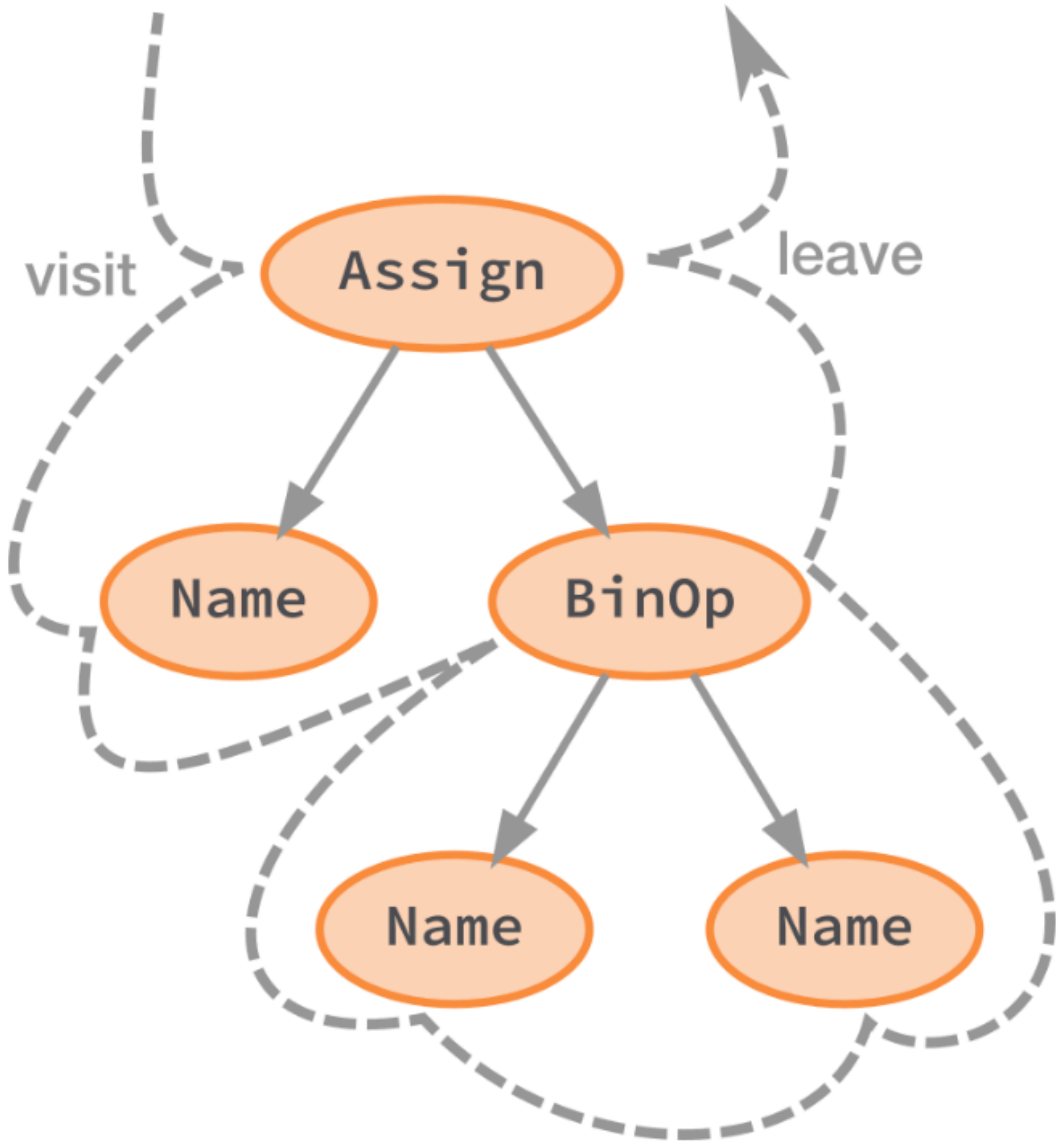
```
libcst.Module
 ├─ bytes: b'\nresult: int = 0\nif condition == True:\n  # an important comment\n  result = 1\n'
 ├─ code: '\nresult: int = 0\nif condition == True:\n  # an important comment\n  result = 1\n'
 ├─ config_for_parsing: PartialParserConfig(encoding='utf-8', default_indent='    ', default_newline='\n')
 ├─ default_indent: '    '
 ├─ default_newline: '\n'
 ├─ encoding: 'utf-8'
 ├─ has_trailing_newline: True
 ├─ body
 │   ├─ [0] libcst.SimpleStatementLine
 │   │   ├─ body
 │   │   │   └─ [0] libcst.AnnAssign
 │   │   │       ├─ annotation: libcst.Annotation
 │   │   │       │   ├─ annotation: libcst.Name
 │   │   │       │   │   └─ value: 'int'
 │   │   │       │   ├─ whitespace_after_indicator: libcst.SimpleWhitespace
 │   │   │       │   │   └─ value: ' '
 │   │   │       │   └─ whitespace_before_indicator: libcst.SimpleWhitespace
 │   │   │       ├─ equal: libcst.AssignEqual
 │   │   │       │   ├─ whitespace_after: libcst.SimpleWhitespace
 │   │   │       │   │   └─ value: ' '
 │   │   │       │   └─ whitespace_before: libcst.SimpleWhitespace
 │   │   │       │       └─ value: ' '
 │   │   │       ├─ target: libcst.Name
 │   │   │       │   └─ value: 'result'
 │   │   │       └─ value: libcst.Integer
 │   │   │           ├─ evaluated_value: 0
 │   │   │           └─ value: '0'
 │   │   └─ trailing_whitespace: libcst.TrailingWhitespace
 │   │       ├─ newline: libcst.Newline
 │   │       └─ whitespace: libcst.SimpleWhitespace
 │   └─ [1] libcst.If
 │       ├─ body: libcst.IndentedBlock
 │       │   ├─ body
 │       │   │   └─ [0] libcst.SimpleStatementLine
 │       │   │       ├─ body
 │       │   │       │   └─ [0] libcst.Assign
 │       │   │       │       ├─ targets
 │       │   │       │       │   └─ [0] libcst.AssignTarget
 │       │   │       │       │       ├─ target: libcst.Name
 │       │   │       │       │       │   └─ value: 'result'
 │       │   │       │       │       ├─ whitespace_after_equal: libcst.SimpleWhitespace
 │       │   │       │       │       │   └─ value: ' '
 │       │   │       │       │       └─ whitespace_before_equal: libcst.SimpleWhitespace
 │       │   │       │       │           └─ value: ' '
 │       │   │       │       └─ value: libcst.Integer
 │       │   │       │           ├─ evaluated_value: 1
 │       │   │       │           └─ value: '1'
 │       │   │       ├─ leading_lines
 │       │   │       │   └─ [0] libcst.EmptyLine
 │       │   │       │       ├─ indent: True
 │       │   │       │       ├─ comment: libcst.Comment
 │       │   │       │       │   └─ value: '# an important comment'
 │       │   │       │       ├─ newline: libcst.Newline
 │       │   │       │       └─ whitespace: libcst.SimpleWhitespace
 │       │   │       └─ trailing_whitespace: libcst.TrailingWhitespace
 │       │   │           ├─ newline: libcst.Newline
 │       │   │           └─ whitespace: libcst.SimpleWhitespace
 │       │   └─ header: libcst.TrailingWhitespace
 │       │       ├─ newline: libcst.Newline
 │       │       └─ whitespace: libcst.SimpleWhitespace
 │       ├─ test: libcst.Comparison
 │       │   ├─ comparisons
 │       │   │   └─ [0] libcst.ComparisonTarget
 │       │   │       ├─ comparator: libcst.Name
 │       │   │       │   └─ value: 'True'
 │       │   │       └─ operator: libcst.Equal
 │       │   │           ├─ whitespace_after: libcst.SimpleWhitespace
 │       │   │           │   └─ value: ' '
 │       │   │           └─ whitespace_before: libcst.SimpleWhitespace
 │       │   │               └─ value: ' '
 │       │   └─ left: libcst.Name
 │       │       └─ value: 'condition'
 │       ├─ whitespace_after_test: libcst.SimpleWhitespace
 │       └─ whitespace_before_test: libcst.SimpleWhitespace
 │           └─ value: ' '
 └─ header
     └─ [0] libcst.EmptyLine
         ├─ indent: True
         ├─ newline: libcst.Newline
         └─ whitespace: libcst.SimpleWhitespace
```

Visitor Pattern From AST

Traverse syntax tree and focus on specific type of nodes

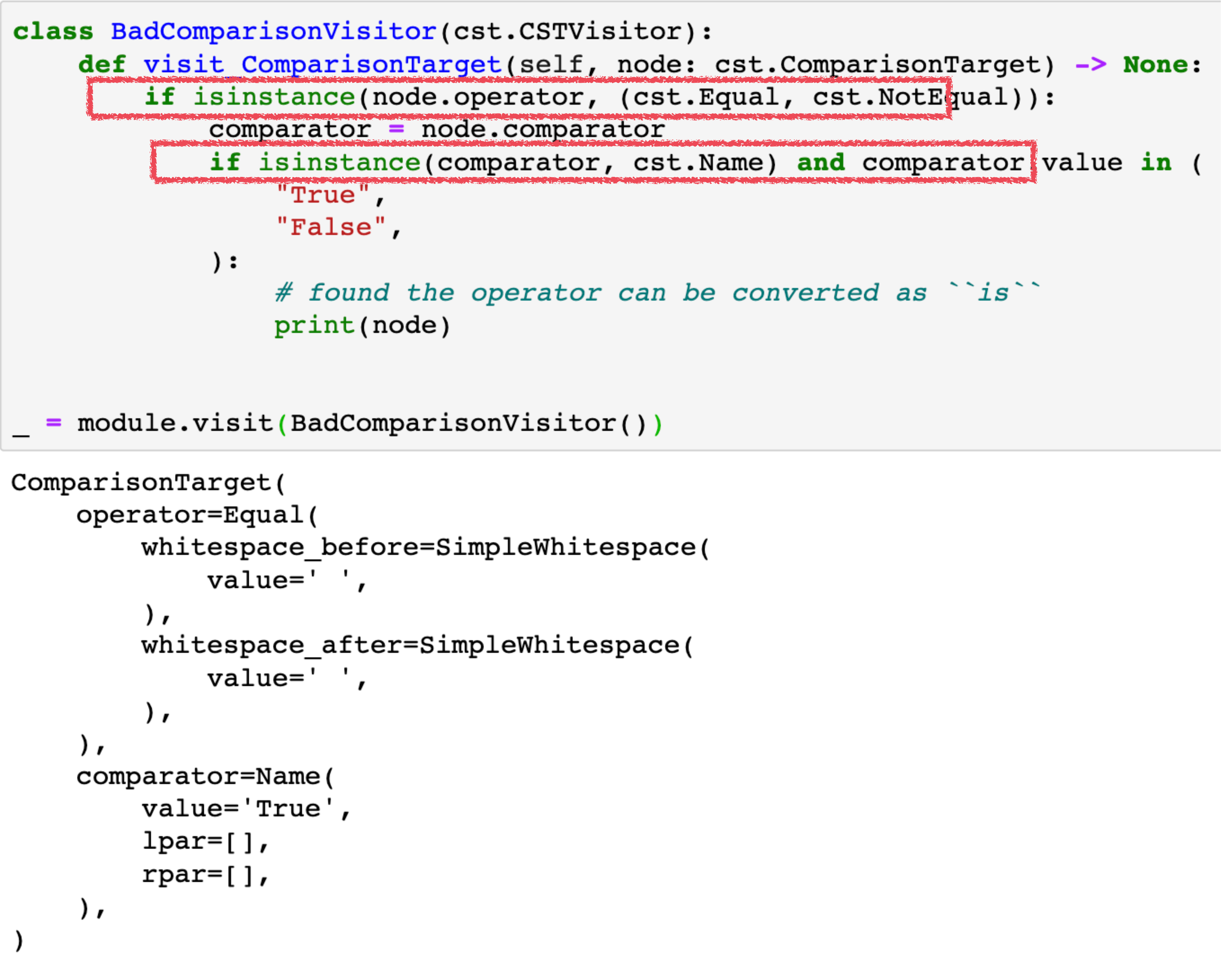


```python
import libcst as cst

class FindAssignmentsVisitor(cst.CSTVisitor):
    def visit_Assign(self, node):
        ...  # called first

    def visit_Name(self, node):
        ...  # called for each child

    def leave_Assign(self, name):
        ...  # called after all children
```



Use Visitor

- Inspect tree manually
  - Repeated pattern:
    - isinstance + check value



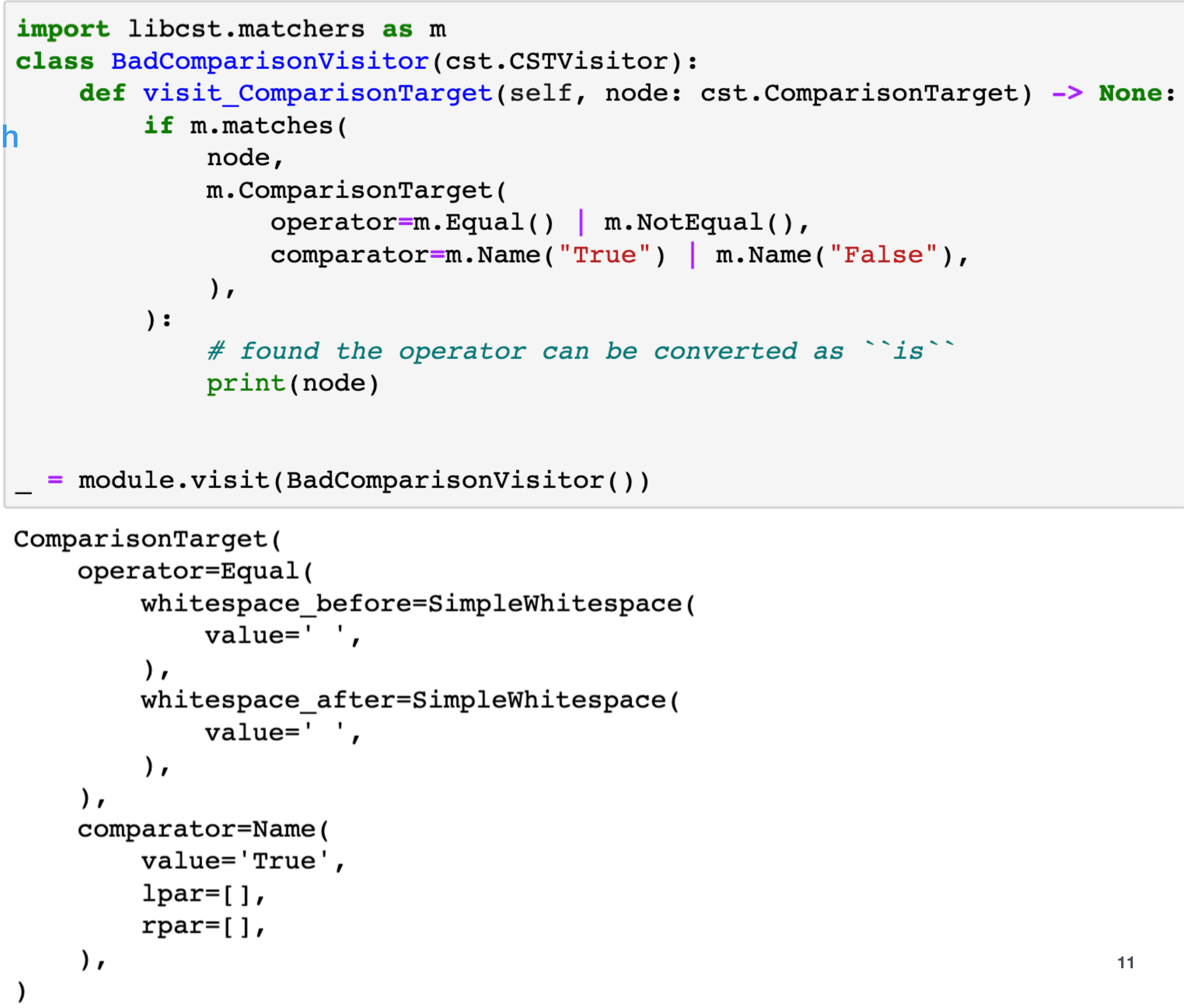
```
class BadComparisonVisitor(cst.CSTVisitor):
    def visit_ComparisonTarget(self, node: cst.ComparisonTarget) -> None:
        if isinstance(node.operator, (cst.Equal, cst.NotEqual)):
            comparator = node.comparator
            if isinstance(comparator, cst.Name) and comparator.value in (
                "True",
                "False",
            ):
                # found the operator can be converted as ``is``
                print(node)


_ = module.visit(BadComparisonVisitor())
```

```
ComparisonTarget(
    operator=Equal(
        whitespace_before=SimpleWhitespace(
            value=' ',
        ),
        whitespace_after=SimpleWhitespace(
            value=' ',
        ),
    ),
    comparator=Name(
        value='True',
        lpar=[],
        rpar=[],
    ),
)
```

Matcher

Describe the shape you'd like to match

- Easier to read and write

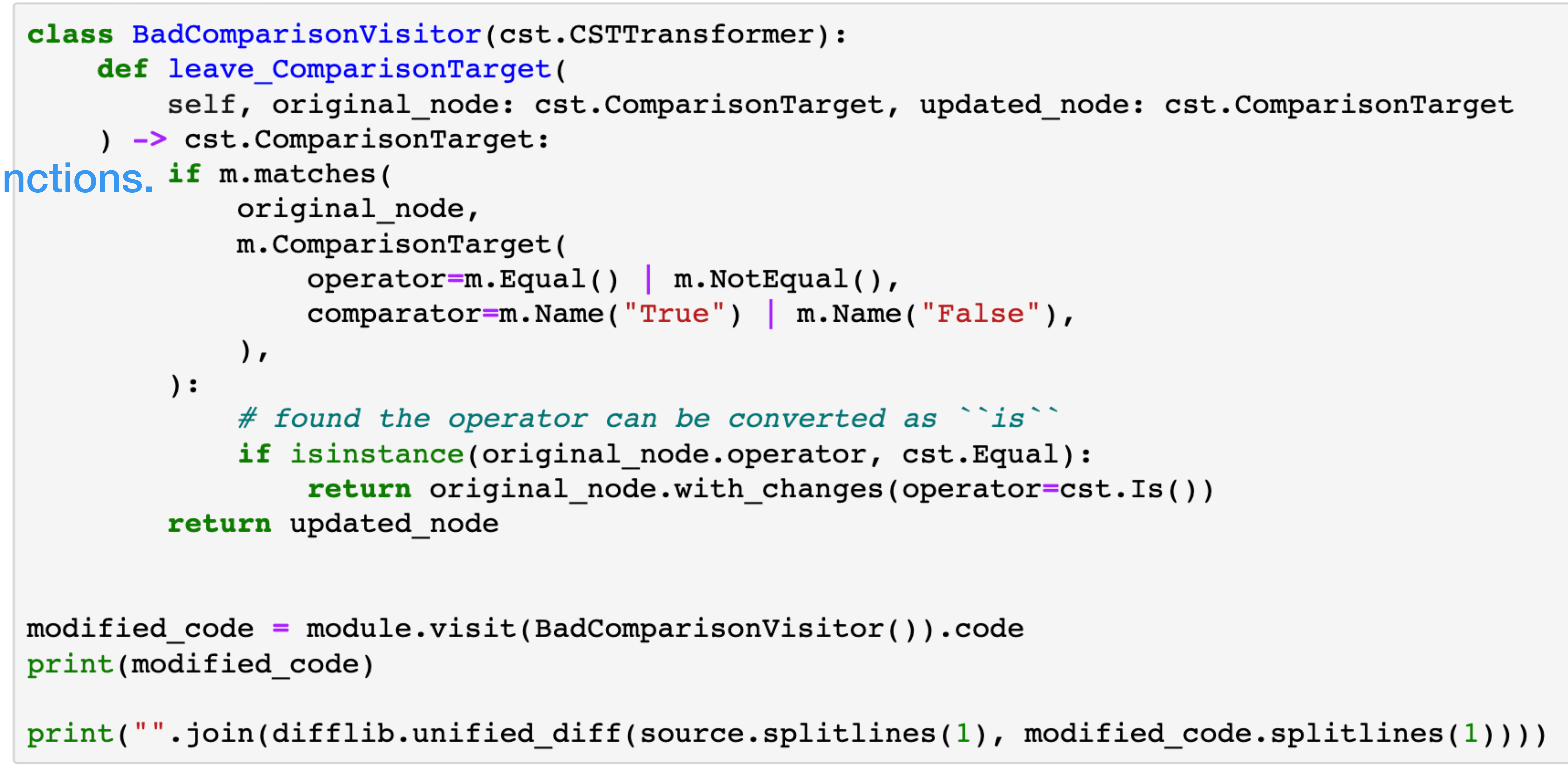


```
import libcst.matchers as m
class BadComparisonVisitor(cst.CSTVisitor):
    def visit_ComparisonTarget(self, node: cst.ComparisonTarget) -> None:
        if m.matches(
            node,
            m.ComparisonTarget(
                operator=m.Equal() | m.NotEqual(),
                comparator=m.Name("True") | m.Name("False"),
            ),
        ):
            # found the operator can be converted as ``is``
            print(node)


_ = module.visit(BadComparisonVisitor())
```

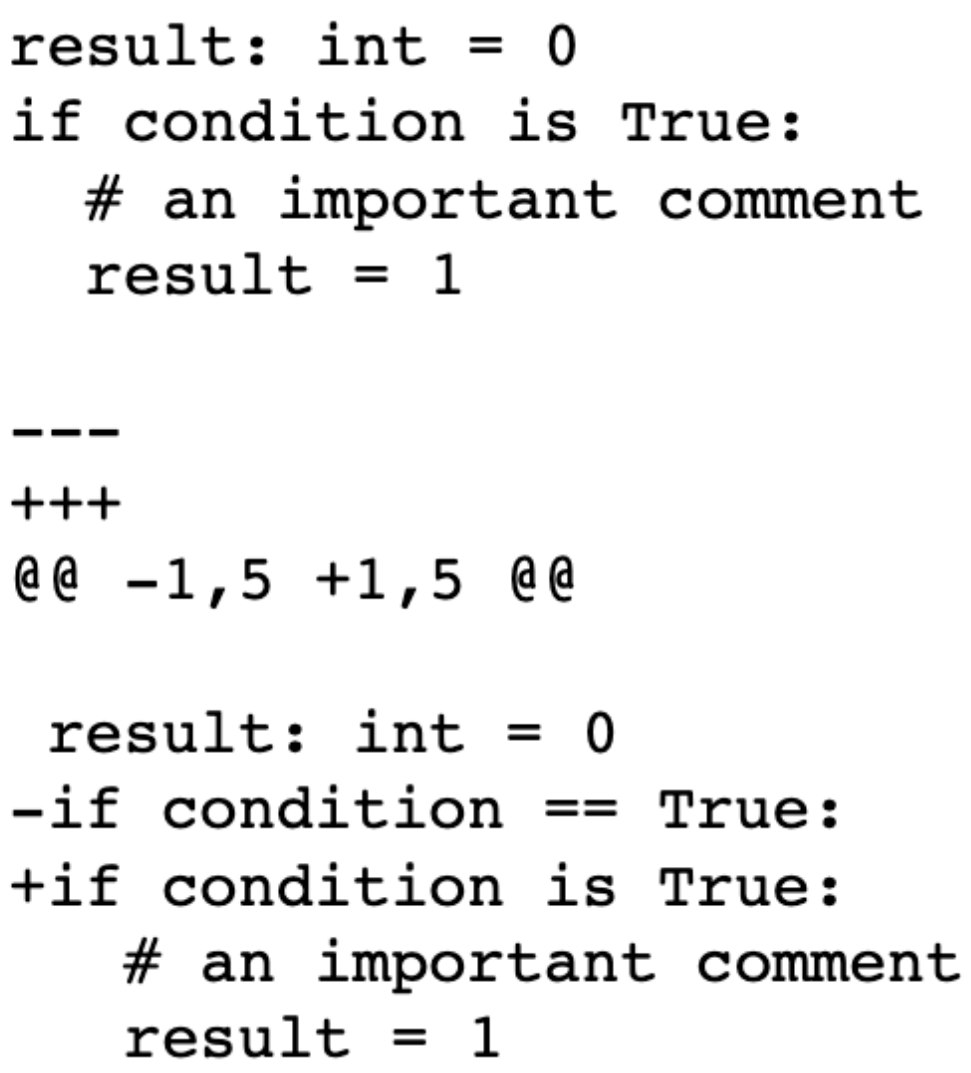
```
ComparisonTarget(
    operator=Equal(
        whitespace_before=SimpleWhitespace(
            value=' ',
        ),
        whitespace_after=SimpleWhitespace(
            value=' ',
        ),
    ),
    comparator=Name(
        value='True',
        lpar=[],
        rpar=[],
    ),
)
```

Transformer Pattern

Tree modification on leave_Node( ) functions.



```python
class BadComparisonVisitor(cst.CSTTransformer):
    def leave_ComparisonTarget(
        self, original_node: cst.ComparisonTarget, updated_node: cst.ComparisonTarget
    ) -> cst.ComparisonTarget:
        if m.matches(
            original_node,
            m.ComparisonTarget(
                operator=m.Equal() | m.NotEqual(),
                comparator=m.Name("True") | m.Name("False"),
            ),
        ):
            # found the operator can be converted as ``is``
            if isinstance(original_node.operator, cst.Equal):
                return original_node.with_changes(operator=cst.Is())
        return updated_node


modified_code = module.visit(BadComparisonVisitor()).code
print(modified_code)

print("".join(difflib.unified_diff(source.splitlines(1), modified_code.splitlines(1))))
```

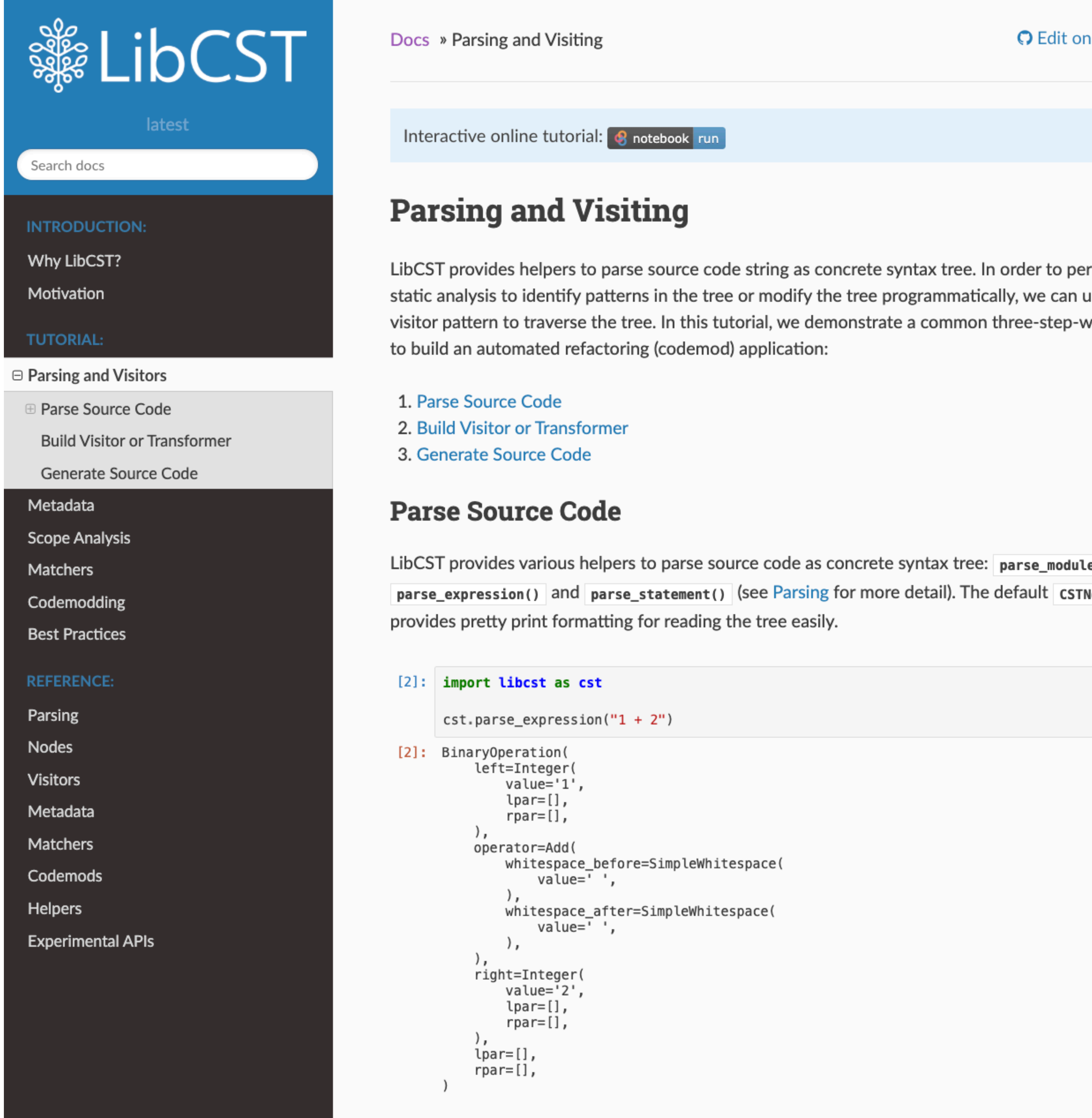


```
result: int = 0
if condition is True:
  # an important comment
  result = 1

---
+++
@@ -1,5 +1,5 @@

 result: int = 0
-if condition == True:
+if condition is True:
   # an important comment
   result = 1
```

More LibCST Components

- Metadata API and providers.
- Codemod: run code transforms on entire codebase.
- Helpers: write less and do more.

- LibCST documentation
  - https://libcst.readthedocs.io/en/latest/index.html



Interactive online tutorial: notebook run

Parsing and Visiting

LibCST provides helpers to parse source code string as concrete syntax tree. In order to per static analysis to identify patterns in the tree or modify the tree programmatically, we can u visitor pattern to traverse the tree. In this tutorial, we demonstrate a common three-step-w to build an automated refactoring (codemod) application:

1. Parse Source Code
2. Build Visitor or Transformer
3. Generate Source Code

Parse Source Code

LibCST provides various helpers to parse source code as concrete syntax tree: `parse_module` `parse_expression()` and `parse_statement()` (see Parsing for more detail). The default `CSTN` provides pretty print formatting for reading the tree easily.

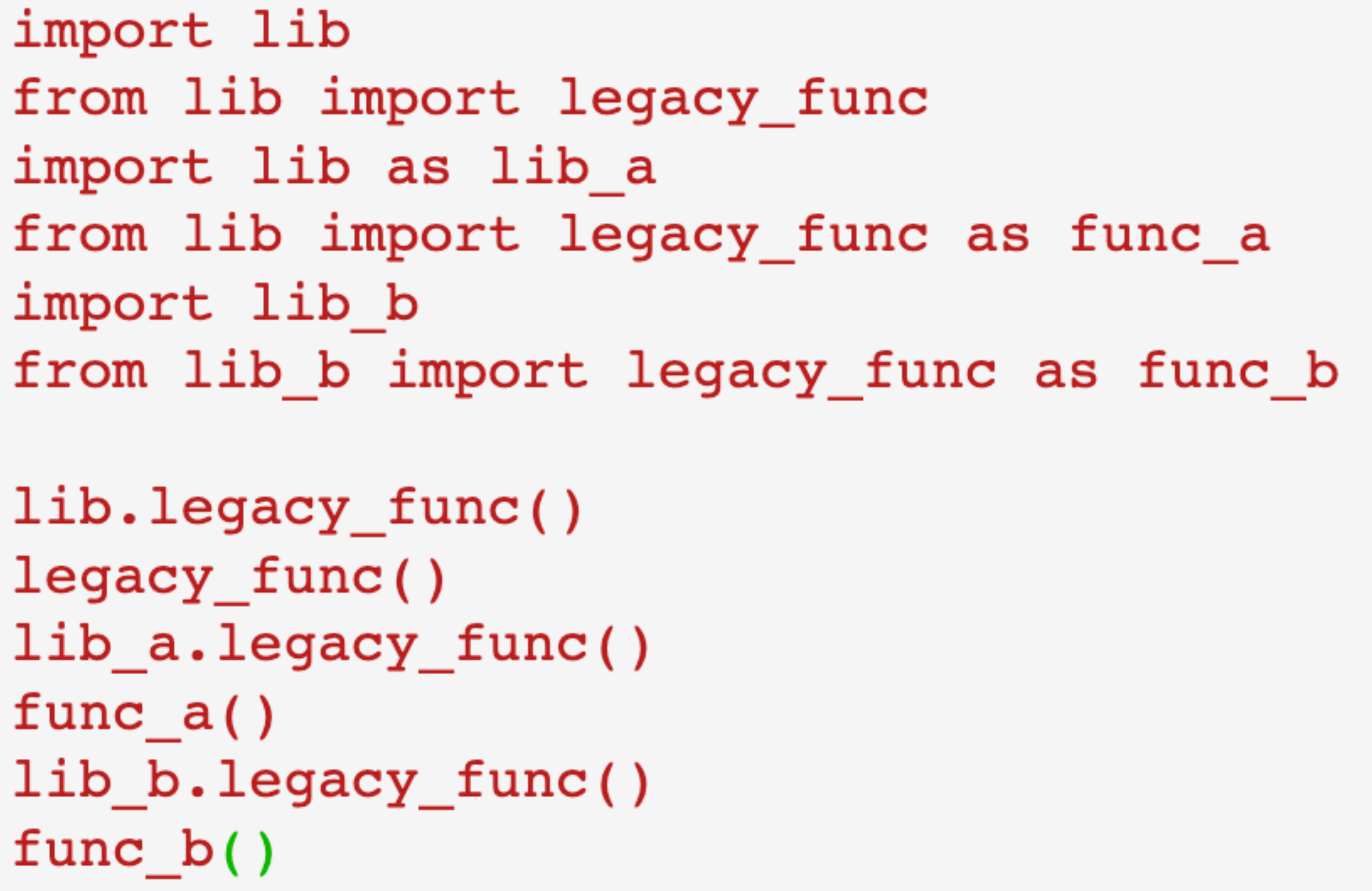
```
[2]: import libcst as cst

     cst.parse_expression("1 + 2")
```

```
[2]: BinaryOperation(
         left=Integer(
             value='1',
             lpar=[],
             rpar=[],
         ),
         operator=Add(
             whitespace_before=SimpleWhitespace(
                 value=' ',
             ),
             whitespace_after=SimpleWhitespace(
                 value=' ',
             ),
         ),
         right=Integer(
             value='2',
             lpar=[],
             rpar=[],
         ),
         lpar=[],
         rpar=[],
     )
```

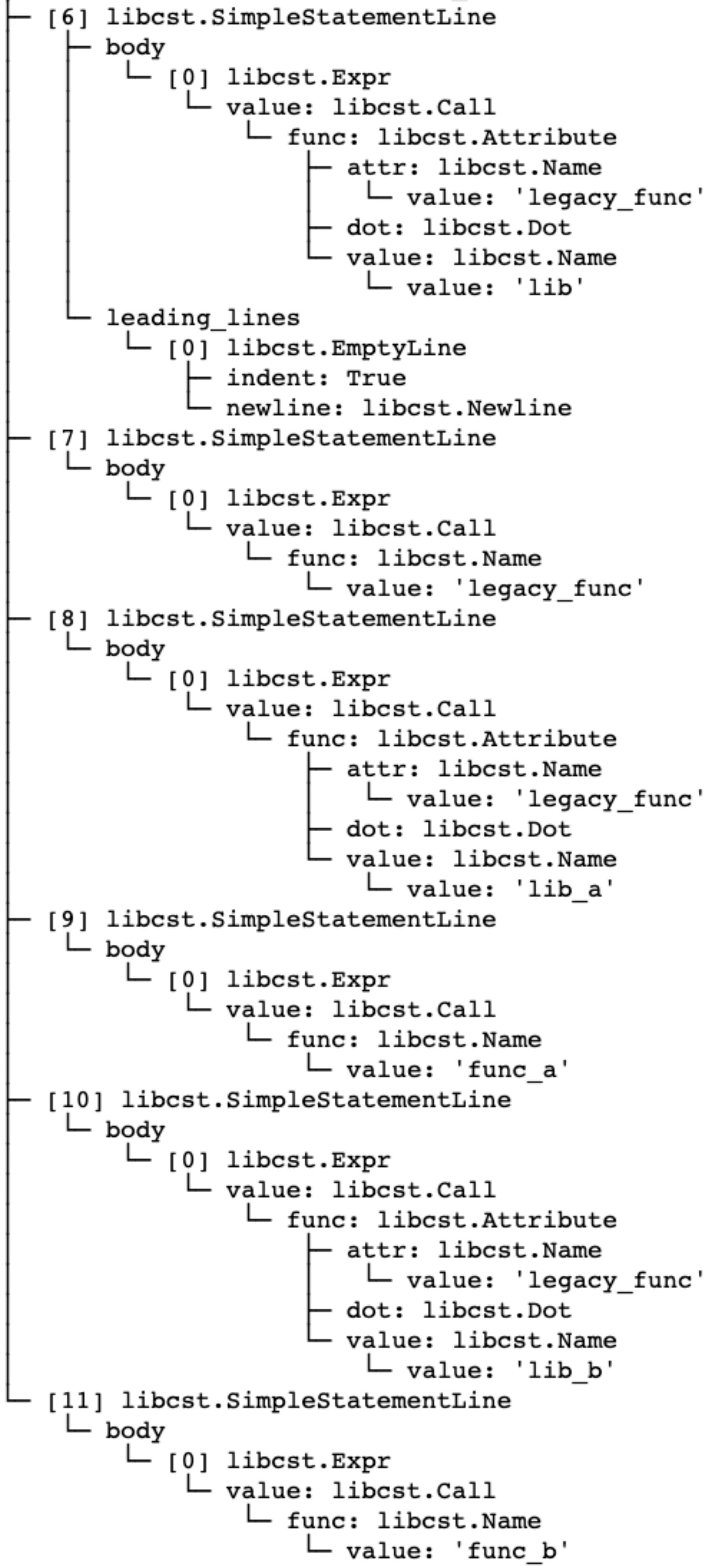
Rename a Legacy API

A common use case.

- Rename lib.legacy_func as lib.new_func
- Not trivial to identify all legacy_func calls.



```
import lib
from lib import legacy_func
import lib as lib_a
from lib import legacy_func as func_a
import lib_b
from lib_b import legacy_func as func_b

lib.legacy_func()
legacy_func()
lib_a.legacy_func()
func_a()
lib_b.legacy_func()
func_b()
```

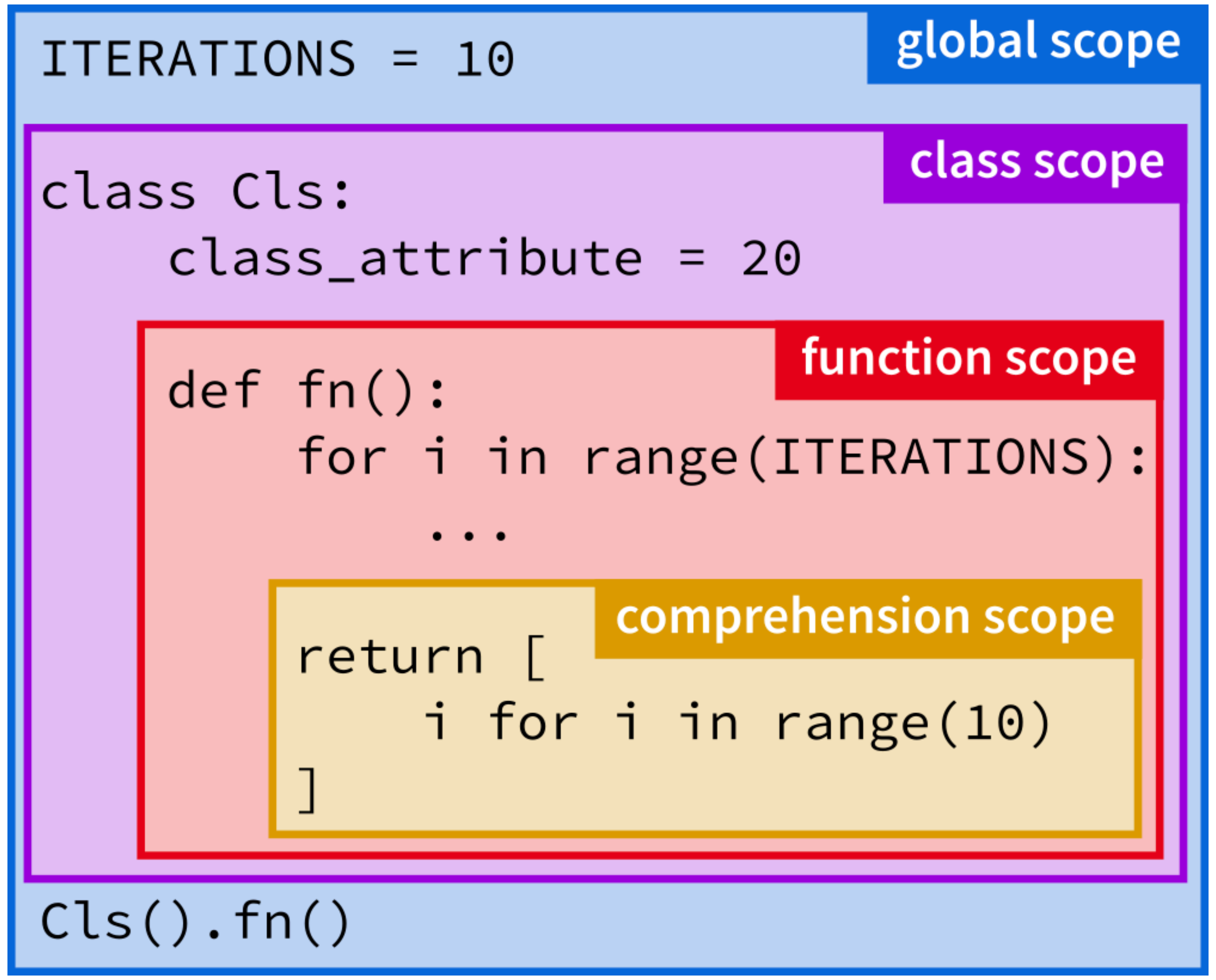


```
├─ [6] libcst.SimpleStatementLine
│   ├─ body
│   │   └─ [0] libcst.Expr
│   │       └─ value: libcst.Call
│   │           └─ func: libcst.Attribute
│   │               ├─ attr: libcst.Name
│   │               │   └─ value: 'legacy_func'
│   │               ├─ dot: libcst.Dot
│   │               └─ value: libcst.Name
│   │                   └─ value: 'lib'
│   └─ leading_lines
│       └─ [0] libcst.EmptyLine
│           ├─ indent: True
│           └─ newline: libcst.Newline
├─ [7] libcst.SimpleStatementLine
│   └─ body
│       └─ [0] libcst.Expr
│           └─ value: libcst.Call
│               └─ func: libcst.Name
│                   └─ value: 'legacy_func'
├─ [8] libcst.SimpleStatementLine
│   └─ body
│       └─ [0] libcst.Expr
│           └─ value: libcst.Call
│               └─ func: libcst.Attribute
│                   ├─ attr: libcst.Name
│                   │   └─ value: 'legacy_func'
│                   ├─ dot: libcst.Dot
│                   └─ value: libcst.Name
│                       └─ value: 'lib_a'
├─ [9] libcst.SimpleStatementLine
│   └─ body
│       └─ [0] libcst.Expr
│           └─ value: libcst.Call
│               └─ func: libcst.Name
│                   └─ value: 'func_a'
├─ [10] libcst.SimpleStatementLine
│   └─ body
│       └─ [0] libcst.Expr
│           └─ value: libcst.Call
│               └─ func: libcst.Attribute
│                   ├─ attr: libcst.Name
│                   │   └─ value: 'legacy_func'
│                   ├─ dot: libcst.Dot
│                   └─ value: libcst.Name
│                       └─ value: 'lib_b'
└─ [11] libcst.SimpleStatementLine
    └─ body
        └─ [0] libcst.Expr
            └─ value: libcst.Call
                └─ func: libcst.Name
                    └─ value: 'func_b'
```

Scope Analysis

Track name definition and accesses across scopes.

- ExpressionContextProvider
- ScopeProvider
- QualifiedNameProvider



```
ITERATIONS = 10

class Cls:
    class_attribute = 20

    def fn():
        for i in range(ITERATIONS):
            ...

        return [
            i for i in range(10)
        ]

Cls().fn()
```

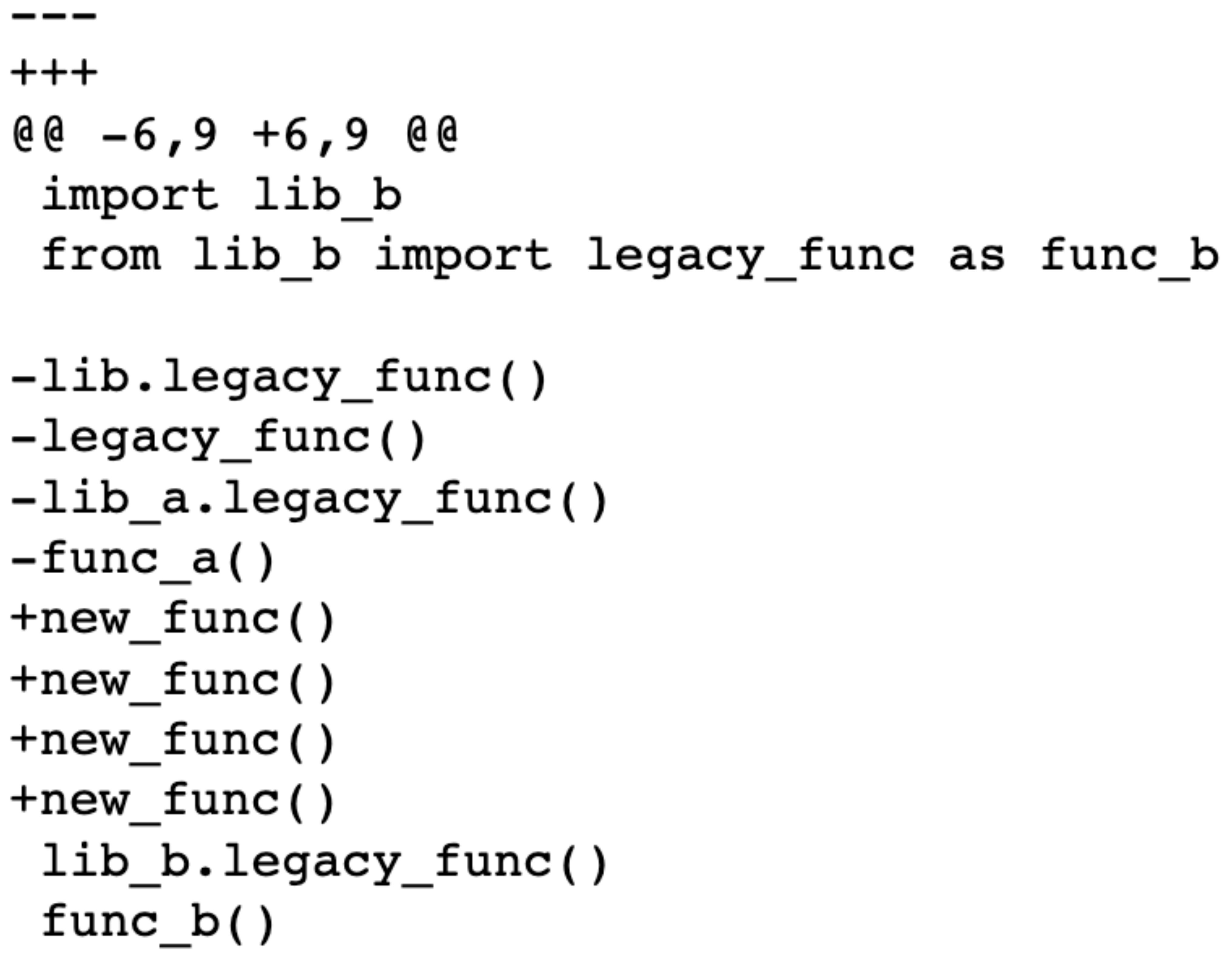
QualifiedName: Cls.fn, Local

QualifiedName: Cls.fn, Local

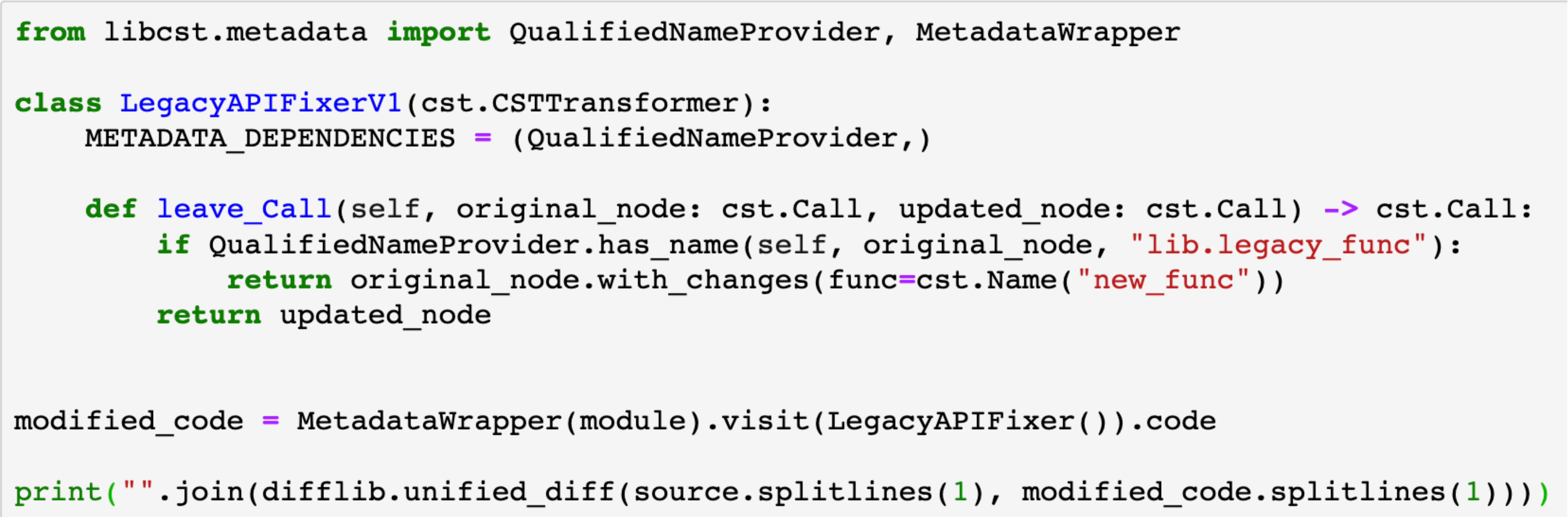
QualifiedNameProvider

A metadata provider provides QualifiedName

- Check identity of Name, Attribute, Call, FunctionDef, ClassDef, ... easily



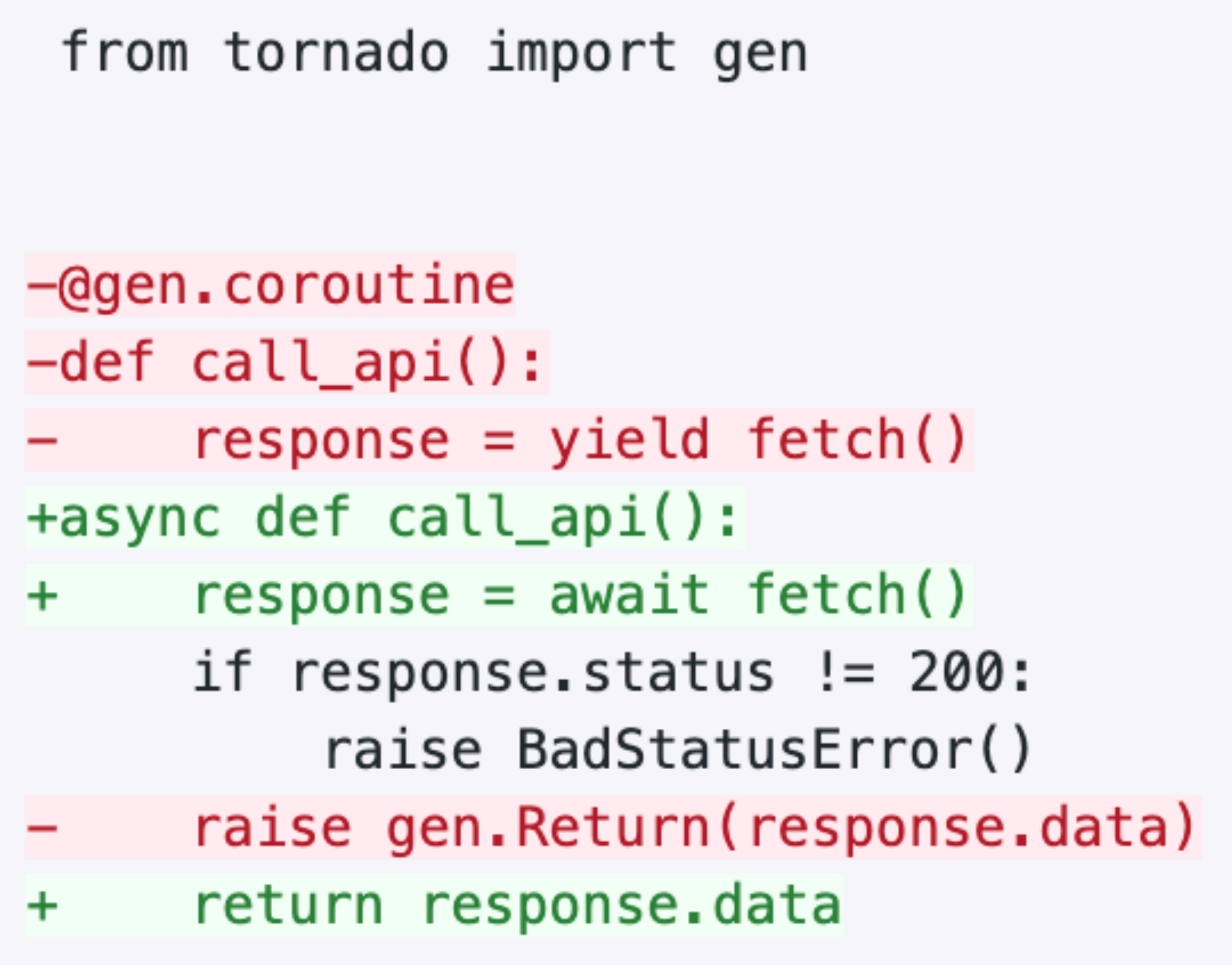
```
---
+++
@@ -6,9 +6,9 @@
 import lib_b
 from lib_b import legacy_func as func_b

-lib.legacy_func()
-legacy_func()
-lib_a.legacy_func()
-func_a()
+new_func()
+new_func()
+new_func()
+new_func()
 lib_b.legacy_func()
 func_b()
```



```python
from libcst.metadata import QualifiedNameProvider, MetadataWrapper

class LegacyAPIFixerV1(cst.CSTTransformer):
    METADATA_DEPENDENCIES = (QualifiedNameProvider,)

    def leave_Call(self, original_node: cst.Call, updated_node: cst.Call) -> cst.Call:
        if QualifiedNameProvider.has_name(self, original_node, "lib.legacy_func"):
            return original_node.with_changes(func=cst.Name("new_func"))
        return updated_node


modified_code = MetadataWrapper(module).visit(LegacyAPIFixer()).code

print("".join(difflib.unified_diff(source.splitlines(1), modified_code.splitlines(1))))
```

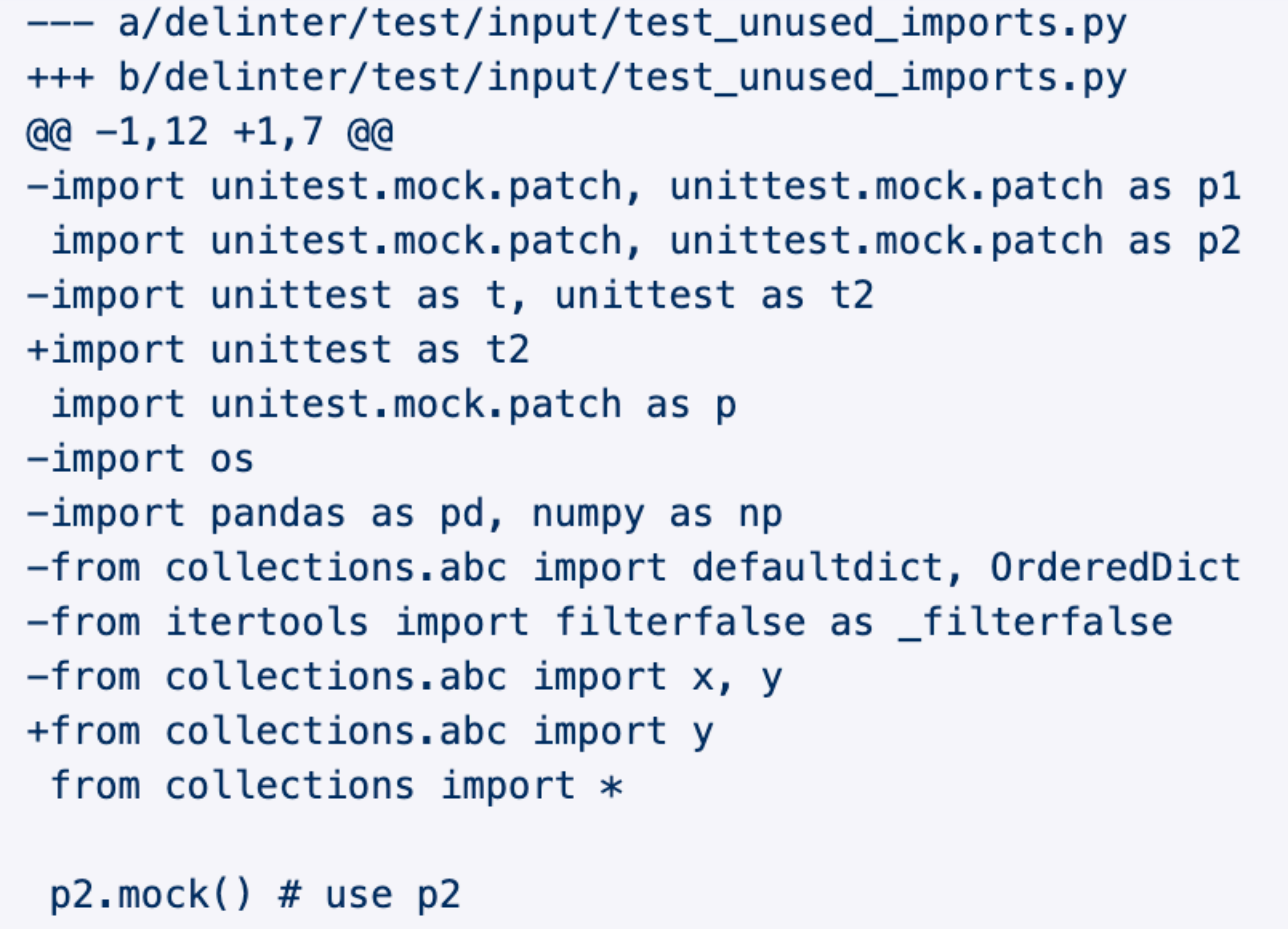
Open Source LibCST Applications

- tornado-async-transformer



```
 from tornado import gen


-@gen.coroutine
-def call_api():
-    response = yield fetch()
+async def call_api():
+    response = await fetch()
     if response.status != 200:
         raise BadStatusError()
-    raise gen.Return(response.data)
+    return response.data
```

- pydelinter



```
--- a/delinter/test/input/test_unused_imports.py
+++ b/delinter/test/input/test_unused_imports.py
@@ -1,12 +1,7 @@
-import unitest.mock.patch, unittest.mock.patch as p1
 import unitest.mock.patch, unittest.mock.patch as p2
-import unittest as t, unittest as t2
+import unittest as t2
 import unitest.mock.patch as p
-import os
-import pandas as pd, numpy as np
-from collections.abc import defaultdict, OrderedDict
-from itertools import filterfalse as _filterfalse
-from collections.abc import x, y
+from collections.abc import y
 from collections import *

 p2.mock() # use p2
```

- Coming up: LibCST-based autofixer lint framework

Github: https://github.com/Instagram/LibCST/

Doc: https://libcst.readthedocs.io/

Blog post: Static Analysis at Scale: An Instagram Story

Your contributions are more than welcome!

Q&A