LibCST Documentation

Benjamin Woodruff, Jennifer Taylor, Carl Meyer, Jimmy Lai, Ray Z

Feb 01, 2022

INTRODUCTION:

LibCST parses Python 3.0 -> 3.11 source code as a CST tree that keeps all formatting details (comments, whitespaces, parentheses, etc). It's useful for building automated refactoring (codemod) applications and linters.

WHY LIBCST?

Python's ast module already provides a syntax tree. Why do we need another?

LibCST creates a compromise between an Abstract Syntax Tree (AST) and a traditional Concrete Syntax Tree (CST). By carefully reorganizing and naming node types and fields, we've created a lossless CST that looks and feels like an AST.

1.1 Abstract Syntax Trees (AST)

Let's look at Python's AST for the following code snippet:

```
fn(1, 2)  # calls fn
```

```
ast.Module(
    body=[
        ast.Expr(
            value=ast.Call(
                func=ast.Name("fn", ctx=ast.Load()),
                args=[ast.Num(n=1), ast.Num(n=2)],
                keywords=[],
            ),
        ),
    ],
)
```

This syntax tree does a great job of preserving the semantics of the original code, and the structure of the tree is relatively simple.

However, given only the AST, it wouldn't be possible to reprint the original source code. Like a JPEG, the Abstract Syntax Tree is lossy.

- The comment we left at the line is gone.

- There's a newline at the end of the file, but the AST doesn't tell us that. It also doesn't tell us if it's \n, \r, or \r\n.

- We've lost some information about the whitespace between the first and second argument.

Abstract Syntax Trees are good for tools like compilers and type checkers where the semantics of code is important, but the exact syntax isn't.

1.2 Concrete Syntax Trees (CST)

A popular CST library for Python is lib2to3, which powers tools like 2to3 and Black. Let's look at the syntax tree it generates for the same piece of code:

```python
fn(1, 2)  # calls fn
```

```python
Node(
    file_input,
    children=[
        Node(
            simple_stmt,
            children=[
                Node(
                    power,
                    children=[
                        Leaf(NAME, "fn", prefix=""),
                        Node(
                            trailer,
                            children=[
                                Leaf(LPAR, "(", prefix=""),
                                Node(
                                    arglist,
                                    children=[
                                        Leaf(NUMBER, "1", prefix=""),
                                        Leaf(COMMA, ",", prefix=""),
                                        Leaf(NUMBER, "2", prefix=" "),
                                    ],
                                ),
                                Leaf(RPAR, ")", prefix=""),
                            ],
                        ),
                    ],
                ),
                Leaf(
                    NEWLINE,
                    "\n",
                    prefix="  # calls fn",
                ),
            ],
            prefix=""
        ),
        Leaf(ENDMARKER, "", prefix=""),
    ],
    prefix="",
)
```

This tree is lossless. It retains enough information to reprint the exact input code by storing whitespace information in `prefix` properties. This makes it a "Concrete" Syntax Tree, or CST.

However, much of the semantics of the code is now difficult to understand and extract. lib2to3 presents a tree that closely matches Python's grammar which can be hard to manipulate for complex operations.

- Adding or removing a parameter from `fn` requires careful preservation of `COMMA` nodes.

- Whitespace and comment ownership is unclear. Deleting nodes could result in invalid generated code.

Concrete Syntax Trees are good for operations that don't significantly change the tree and tools that do not wish to change the semantics of the code itself, such as Black.

1.3 LibCST

LibCST takes a compromise between the two formats outlined above. Like a CST, LibCST preserves all whitespace and can be reprinted exactly. Like an AST, LibCST parses source into nodes that represent the semantics of the code.

```
fn(1, 2)  # calls fn
```

```
Module(
    body=[
        SimpleStatementLine(
            body=[
                Expr(
                    value=Call(
                        func=Name(
                            value='fn',
                            lpar=[],
                            rpar=[],
                        ),
                        args=[
                            Arg(
                                value=Integer(
                                    value='1',
                                    lpar=[],
                                    rpar=[],
                                ),
                                keyword=None,
                                equal=MaybeSentinel.DEFAULT,
                                comma=Comma(
                                    whitespace_before=SimpleWhitespace(
                                        value='',
                                    ),
                                    whitespace_after=SimpleWhitespace(
                                        value=' ',
                                    ),
                                ),
                                star='',
                                whitespace_after_star=SimpleWhitespace(
                                    value='',
                                ),
                                whitespace_after_arg=SimpleWhitespace(
                                    value='',
                                ),
                            ),
                            Arg(
                                value=Integer(
                                    value='2',
                                    lpar=[],
                                    rpar=[],
                                ),
                                keyword=None,
                                equal=MaybeSentinel.DEFAULT,
                                comma=MaybeSentinel.DEFAULT,
```

(continues on next page)

```
                        star='',
                        whitespace_after_star=SimpleWhitespace(
                            value='',
                        ),
                        whitespace_after_arg=SimpleWhitespace(
                            value='',
                        ),
                    ),
                ],
                lpar=[],
                rpar=[],
                whitespace_after_func=SimpleWhitespace(
                    value='',
                ),
                whitespace_before_args=SimpleWhitespace(
                    value='',
                ),
            ),
            semicolon=MaybeSentinel.DEFAULT,
        ),
    ],
    leading_lines=[],
    trailing_whitespace=TrailingWhitespace(
        whitespace=SimpleWhitespace(
            value='  ',
        ),
        comment=Comment(
            value='# calls fn',
        ),
        newline=Newline(
            value=None,
        ),
    ),
    ),
],
header=[],
footer=[],
encoding='utf-8',
default_indent='    ',
default_newline='\n',
has_trailing_newline=True,
)
```

LibCST preserves whitespace by parsing it using an internal whitespace parser and assigning it to relevant nodes. This allows for much more granular whitespace ownership and greatly reduces the amount of work necessary to perform complex manipulations. Additionally, it is fully typed. A node's children are well-defined and match the semantics of Python.

However, this does come with some downsides.

- It is more difficult to implement tools that focus almost exclusively on whitespace on top of LibCST instead of lib2to3. For example, Black would need to modify whitespace nodes instead of prefix strings, making its implementation much more complex.

- The equivalent AST for a Python module will usually be simpler. We must preserve whitespace ownership by assigning it to nodes that make the most sense which requires us to introduce nodes such as *Comma*.

- Parsing with LibCST will always be slower than Python's AST due to the extra work needed to assign whitespace correctly.

Nevertheless, we think that the trade-offs made in LibCST are worthwhile and offer a great deal of flexibility and power.

MOTIVATION

When designing LibCST, we used the following list of motivations.

2.1 Exact Representation

- **Trees should be rewritable.** It should always be possible to take a valid python file, parse it to a CST using LibCST and then write that tree back out exactly, byte for byte. When changing nodes in the tree, changes to the original source file should be localized to the area represented by the changed portion of the tree. Effectively, for all valid python inputs, the following equation should be true:

```
parse_module(some_input).code == some_input
```

- **Nodes should be constructed exactly as written in code.** No magic should happen on initialization and all construction should be explicit. Nodes should directly correlate to the code they represent and vice versa.

2.2 Ease of Traversal

- **As flat as possible.** There shouldn't be an AsyncFunction wrapper containing a FunctionDef just because the grammar specifies it that way. Instead, we should make a FunctionDef node and give it an async attribute. Instead of representing parenthesis as wrapper nodes, they should be attached to the expressions that they operate on. In any scenario where we could achieve deduplication of LibCST code through extra layers in the resulting tree, we will opt for more code in order to make traversal simpler.

- **As regular as possible.** A module should always have a list of statements, even if that list is empty or only has one item. Irregularity makes tree inspection more difficult.

- **As high-level as possible.** The tree should be as close to the Python AST as possible. It should not be necessary to understand Python syntax in order to traverse the tree correctly. You should not have to know to ignore commas when traversing a list of parameters for a function. You should not have to use helper functions to traverse or recognize expressions wrapped in parenthesis. A LibCST node will represent its semantic operation in python with as little syntactic trivia exposed as possible.

2.3 Ease of Modification

- **All nodes should be fully typed.** A module is a list of statements, not a list of untyped nodes. A function has a name, parameters and an optional return. It should be clear where to access various attributes of each node and what are the valid node types that can be used for that attribute.

- **Additional runtime (in addition to static types) constraints.** It shouldn't be possible to construct a node that can't be serialized correctly or that would result in invalid code. You shouldn't be able to construct a Name node with a string that isn't a valid python identifier. Strong constraints here should allow us to perform multiple passes safely without serializing and re-parsing the tree after each pass.

- **Sane defaults.** If I construct a node, I shouldn't have to supply whitespace, commas or other required syntax unless I want to. I should be able to treat the node in abstract, specifying only the semantics of the resulting code.

- **Reasonably intelligent ownership of whitespace.** A statement should own the comments directly above it, and any trailing comments on the same line. If we delete that statement, the whitespace should disappear with it.

- **It should be easy to change a single field** in an existing node without needing to modify or fix up adjacent nodes. Syntactic trivia such as commas or proper spacing between nodes should be children of the node they logically belong to so that inserting or removing a node does not require modifications to adjacent nodes.

- **Reparentable.** It should be possible to move or copy a node from one part of the tree easily.

2.4 Well Tested

- **All nodes should be fully tested.** It should not be possible to break upstream parsing or rendering code with a change to LibCST. Parsing, rendering and verifying functionality are all tested as completely as possible for all defined nodes.

PARSING AND VISITING

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

1. *Parse Source Code*

2. *Build Visitor or Transformer*

3. *Generate Source Code*

3.1 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 *CSTNode* repr provides pretty print formatting for reading the tree easily.

```python
import libcst as cst

cst.parse_expression("1 + 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=[],
)
```

3.1.1 Example: add typing annotation from pyi stub file to Python source

Python typing annotation was added in Python 3.5. Some Python applications add typing annotations in separate `pyi` stub files in order to support old Python versions. When applications decide to stop supporting old Python versions, they'll want to automatically copy the type annotation from a pyi file to a source file. Here we demonstrate how to do that easliy using LibCST. The first step is to parse the pyi stub and source files as trees.

```python
[3]: py_source = '''
class PythonToken(Token):
    def __repr__(self):
        return ('TokenInfo(type=%s, string=%r, start_pos=%r, prefix=%r)' %
                self._replace(type=self.type.name))

def tokenize(code, version_info, start_pos=(1, 0)):
    """Generate tokens from a the source code (string)."""
    lines = split_lines(code, keepends=True)
    return tokenize_lines(lines, version_info, start_pos=start_pos)
'''

pyi_source = '''
class PythonToken(Token):
    def __repr__(self) -> str: ...

def tokenize(
    code: str, version_info: PythonVersionInfo, start_pos: Tuple[int, int] = (1, 0)
) -> Generator[PythonToken, None, None]: ...
'''

source_tree = cst.parse_module(py_source)
stub_tree = cst.parse_module(pyi_source)
```

3.2 Build Visitor or Transformer

For traversing and modifying the tree, LibCST provides Visitor and Transformer classes similar to the ast module. To implement a visitor (read only) or transformer (read/write), simply implement a subclass of *CSTVisitor* or *CSTTransformer* (see *Visitors* for more detail). In the typing example, we need to implement a visitor to collect typing annotation from the stub tree and a transformer to copy the annotation to the function signature. In the visitor, we implement `visit_FunctionDef` to collect annotations. Later in the transformer, we implement `leave_FunctionDef` to add the collected annotations.

```python
[4]: from typing import List, Tuple, Dict, Optional


class TypingCollector(cst.CSTVisitor):
    def __init__(self):
        # stack for storing the canonical name of the current function
        self.stack: List[Tuple[str, ...]] = []
        # store the annotations
        self.annotations: Dict[
            Tuple[str, ...],  # key: tuple of canonical class/function name
            Tuple[cst.Parameters, Optional[cst.Annotation]],  # value: (params,
→returns)
        ] = {}
```

(continues on next page)

```python
    def visit_ClassDef(self, node: cst.ClassDef) -> Optional[bool]:
        self.stack.append(node.name.value)

    def leave_ClassDef(self, node: cst.ClassDef) -> None:
        self.stack.pop()

    def visit_FunctionDef(self, node: cst.FunctionDef) -> Optional[bool]:
        self.stack.append(node.name.value)
        self.annotations[tuple(self.stack)] = (node.params, node.returns)
        return (
            False
        )  # pyi files don't support inner functions, return False to stop the
→traversal.

    def leave_FunctionDef(self, node: cst.FunctionDef) -> None:
        self.stack.pop()


class TypingTransformer(cst.CSTTransformer):
    def __init__(self, annotations):
        # stack for storing the canonical name of the current function
        self.stack: List[Tuple[str, ...]] = []
        # store the annotations
        self.annotations: Dict[
            Tuple[str, ...],  # key: tuple of canonical class/function name
            Tuple[cst.Parameters, Optional[cst.Annotation]],  # value: (params,
→returns)
        ] = annotations

    def visit_ClassDef(self, node: cst.ClassDef) -> Optional[bool]:
        self.stack.append(node.name.value)

    def leave_ClassDef(
        self, original_node: cst.ClassDef, updated_node: cst.ClassDef
    ) -> cst.CSTNode:
        self.stack.pop()
        return updated_node

    def visit_FunctionDef(self, node: cst.FunctionDef) -> Optional[bool]:
        self.stack.append(node.name.value)
        return (
            False
        )  # pyi files don't support inner functions, return False to stop the
→traversal.

    def leave_FunctionDef(
        self, original_node: cst.FunctionDef, updated_node: cst.FunctionDef
    ) -> cst.CSTNode:
        key = tuple(self.stack)
        self.stack.pop()
        if key in self.annotations:
            annotations = self.annotations[key]
            return updated_node.with_changes(
                params=annotations[0], returns=annotations[1]
            )
        return updated_node
```

```
visitor = TypingCollector()
stub_tree.visit(visitor)
transformer = TypingTransformer(visitor.annotations)
modified_tree = source_tree.visit(transformer)
```

3.3 Generate Source Code

Generating the source code from a cst tree is as easy as accessing the *code* attribute on *Module*. After the code generation, we often use ufmt to reformate the code to keep a consistent coding style.

```
[5]: print(modified_tree.code)
```

```
class PythonToken(Token):
    def __repr__(self) -> str:
        return ('TokenInfo(type=%s, string=%r, start_pos=%r, prefix=%r)' %
                self._replace(type=self.type.name))

def tokenize(code: str, version_info: PythonVersionInfo, start_pos: Tuple[int, int] =
→(1, 0)
) -> Generator[PythonToken, None, None]:
    """Generate tokens from a the source code (string)."""
    lines = split_lines(code, keepends=True)
    return tokenize_lines(lines, version_info, start_pos=start_pos)
```

```
[6]: # Use difflib to show the changes to verify type annotations were added as expected.
import difflib

print(
    "".join(
        difflib.unified_diff(py_source.splitlines(1), modified_tree.code.
→splitlines(1))
    )
)
```

```
---
+++
@@ -1,10 +1,11 @@

 class PythonToken(Token):
-    def __repr__(self):
+    def __repr__(self) -> str:
        return ('TokenInfo(type=%s, string=%r, start_pos=%r, prefix=%r)' %
                self._replace(type=self.type.name))

-def tokenize(code, version_info, start_pos=(1, 0)):
+def tokenize(code: str, version_info: PythonVersionInfo, start_pos: Tuple[int, int]
→= (1, 0)
+) -> Generator[PythonToken, None, None]:
    """Generate tokens from a the source code (string)."""
    lines = split_lines(code, keepends=True)
    return tokenize_lines(lines, version_info, start_pos=start_pos)
```

For the sake of efficiency, we don't want to re-write the file when the transformer doesn't change the source code. We can use *deep_equals()* to check whether two trees have the same source code. Note that == checks the identity of tree object instead of representation.

```
[7]: if not modified_tree.deep_equals(source_tree):
         ...  # write to file
```

FOUR

WORKING WITH METADATA

LibCST handles node metadata in a somewhat unusual manner in order to maintain the immutability of the tree. See *Metadata* for the complete documentation.

4.1 Providing Metadata

While it's possible to write visitors that gather metadata from a tree ad hoc, using the provider interface gives you the advantage of being able to use dependency declaration to automatically run your providers in other visitors and type safety. For most cases, you'll want to extend `BatchableMetadataProvider` as providers that extend from that class can be resolved more efficiently in batches.

Here's an example of a simple metadata provider that marks *Name* nodes that are function parameters:

```python
import libcst as cst


class IsParamProvider(cst.BatchableMetadataProvider[bool]):
    """
    Marks Name nodes found as a parameter to a function.
    """
    def __init__(self) -> None:
        super().__init__()
        self.is_param = False

    def visit_Param(self, node: cst.Param) -> None:
        # Mark the child Name node as a parameter
        self.set_metadata(node.name, True)

    def visit_Name(self, node: cst.Name) -> None:
        # Mark all other Name nodes as not parameters
        if not self.get_metadata(type(self), node, False):
            self.set_metadata(node, False)
```

4.1.1 Line and Column Metadata

LibCST ships with two built-in providers for line and column metadata. See *Position Metadata* for more information.

4.2 Accessing Metadata

Once you have a provider, the metadata interface gives you two primary ways of working with your providers. The first is using the resolve methods provided by `MetadataWrapper` and the second is through declaring metadata dependencies on a *CSTTransformer* or *CSTVisitor*.

4.2.1 Using the `MetadataWrapper`

The metadata wrapper class provides a way to associate metadata with a module as well as a simple interface to run providers. Here's an example of using a wrapper with the provider we just wrote:

```
[3]: module = cst.parse_module("x")
     wrapper = cst.MetadataWrapper(module)

     isparam = wrapper.resolve(IsParamProvider)
     x_name_node = wrapper.module.body[0].body[0].value

     print(isparam[x_name_node])  # should print False
```

```
False
```

4.2.2 Using Dependency Declaration

The visitors that ship with LibCST can declare metadata providers as dependencies that will be run automatically when visited by a wrapper. Here is a visitor that prints all names that are function parameters.

```
[4]: from libcst.metadata import PositionProvider

     class ParamPrinter(cst.CSTVisitor):
         METADATA_DEPENDENCIES = (IsParamProvider, PositionProvider,)

         def visit_Name(self, node: cst.Name) -> None:
             # Only print out names that are parameters
             if self.get_metadata(IsParamProvider, node):
                 pos = self.get_metadata(PositionProvider, node).start
                 print(f"{node.value} found at line {pos.line}, column {pos.column}")


     module = cst.parse_module("def foo(x):\n    y = 1\n    return x + y")
     wrapper = cst.MetadataWrapper(module)
     result = wrapper.visit(ParamPrinter())  # NB: wrapper.visit not module.visit
```

```
x found at line 1, column 8
```

SCOPE ANALYSIS

Scope analysis keeps track of assignments and accesses which could be useful for code automatic refactoring. If you're not familiar with scope analysis, see *Scope Metadata* for more detail about scope metadata. This tutorial demonstrates some use cases of scope analysis. If you're new to metadata, see *Metadata Tutorial* to get started. Given source codes, scope analysis parses all variable *Assignment* (or a *BuiltinAssignment* if it's a builtin) and *Access* to store in *Scope* containers.

Note: The scope analysis only handles local variable name access and cannot handle simple string type annotation forward references. See *Access*

Given the following example source code contains a couple of unused imports (`f`, `i`, `m` and `n`) and undefined variable references (`func_undefined` and `var_undefined`). Scope analysis helps us identifying those unused imports and undefined variables to automatically provide warnings to developers to prevent bugs while they're developing.

```
[2]: source = """\
     import a, b, c as d, e as f  # expect to keep: a, c as d
     from g import h, i, j as k, l as m  # expect to keep: h, j as k
     from n import o  # expect to be removed entirely

     a()

     def fun():
         d()

     class Cls:
         att = h.something

         def __new__(self) -> "Cls":
             var = k.method()
             func_undefined(var_undefined)
     """
```

With a parsed *Module*, we construct a *MetadataWrapper* object and it provides a *resolve()* function to resolve metadata given a metadata provider. *ScopeProvider* is used here for analysing scope and there are three types of scopes (*GlobalScope*, *FunctionScope* and *ClassScope*) in this example.

```
[3]: import libcst as cst


     wrapper = cst.metadata.MetadataWrapper(cst.parse_module(source))
     scopes = set(wrapper.resolve(cst.metadata.ScopeProvider).values())
     for scope in scopes:
         print(scope)
```

```
<libcst.metadata.scope_provider.GlobalScope object at 0x7f7cf03d0100>
<libcst.metadata.scope_provider.ClassScope object at 0x7f7cf03d0a00>
<libcst.metadata.scope_provider.FunctionScope object at 0x7f7cf03d1660>
<libcst.metadata.scope_provider.FunctionScope object at 0x7f7cf03d0fa0>
```

5.1 Warn on unused imports and undefined references

To find all unused imports, we iterate through *assignments* and an assignment is unused when its *references* is empty. To find all undefined references, we iterate through *accesses* (we focus on *Import*/*ImportFrom* assignments) and an access is undefined reference when its *referents* is empty. When reporting the warning to developer, we'll want to report the line number and column offset along with the suggestion to make it more clear. We can get position information from *PositionProvider* and print the warnings as follows.

```python
[4]: from collections import defaultdict
     from typing import Dict, Union, Set

     unused_imports: Dict[Union[cst.Import, cst.ImportFrom], Set[str]] = defaultdict(set)
     undefined_references: Dict[cst.CSTNode, Set[str]] = defaultdict(set)
     ranges = wrapper.resolve(cst.metadata.PositionProvider)
     for scope in scopes:
         for assignment in scope.assignments:
             node = assignment.node
             if isinstance(assignment, cst.metadata.Assignment) and isinstance(
                 node, (cst.Import, cst.ImportFrom)
             ):
                 if len(assignment.references) == 0:
                     unused_imports[node].add(assignment.name)
                     location = ranges[node].start
                     print(
                         f"Warning on line {location.line:2d}, column {location.column:2d}:
     ↪ Imported name `{assignment.name}` is unused."
                     )

         for access in scope.accesses:
             if len(access.referents) == 0:
                 node = access.node
                 location = ranges[node].start
                 print(
                     f"Warning on line {location.line:2d}, column {location.column:2d}:␣
     ↪Name reference `{node.value}` is not defined."
                 )
```

```
Warning on line  1, column  0: Imported name `b` is unused.
Warning on line  1, column  0: Imported name `f` is unused.
Warning on line  2, column  0: Imported name `i` is unused.
Warning on line  2, column  0: Imported name `m` is unused.
Warning on line  3, column  0: Imported name `o` is unused.
Warning on line 15, column  8: Name reference `func_undefined` is not defined.
Warning on line 15, column 23: Name reference `var_undefined` is not defined.
```

5.2 Automatically Remove Unused Import

Unused import is a commmon code suggestion provided by lint tool like flake8 F401 `imported but unused`. Even though reporting unused import is already useful, with LibCST we can provide automatic fix to remove unused import. That can make the suggestion more actionable and save developer's time.

An import statement may import multiple names, we want to remove those unused names from the import statement. If all the names in the import statement are not used, we remove the entire import. To remove the unused name, we implement `RemoveUnusedImportTransformer` by subclassing *CSTTransformer*. We overwrite `leave_Import` and `leave_ImportFrom` to modify the import statements. When we find the import node in lookup table, we iterate through all `names` and keep used names in `names_to_keep`. If `names_to_keep` is empty, all names are unused and we remove the entire import node. Otherwise, we update the import node and just removing partial names.

```python
[5]: class RemoveUnusedImportTransformer(cst.CSTTransformer):
    def __init__(
        self, unused_imports: Dict[Union[cst.Import, cst.ImportFrom], Set[str]]
    ) -> None:
        self.unused_imports = unused_imports

    def leave_import_alike(
        self,
        original_node: Union[cst.Import, cst.ImportFrom],
        updated_node: Union[cst.Import, cst.ImportFrom],
    ) -> Union[cst.Import, cst.ImportFrom, cst.RemovalSentinel]:
        if original_node not in self.unused_imports:
            return updated_node
        names_to_keep = []
        for name in updated_node.names:
            asname = name.asname
            if asname is not None:
                name_value = asname.name.value
            else:
                name_value = name.name.value
            if name_value not in self.unused_imports[original_node]:
                names_to_keep.append(name.with_changes(comma=cst.MaybeSentinel.
→DEFAULT))
        if len(names_to_keep) == 0:
            return cst.RemoveFromParent()
        else:
            return updated_node.with_changes(names=names_to_keep)

    def leave_Import(
        self, original_node: cst.Import, updated_node: cst.Import
    ) -> cst.Import:
        return self.leave_import_alike(original_node, updated_node)

    def leave_ImportFrom(
        self, original_node: cst.ImportFrom, updated_node: cst.ImportFrom
    ) -> cst.ImportFrom:
        return self.leave_import_alike(original_node, updated_node)
```

After the transform, we use `.code` to generate fixed code and all unused names are fixed as expected! The difflib is used to show only changed part and only import lines are updated as expected.

```
[6]: import difflib
     fixed_module = wrapper.module.visit(RemoveUnusedImportTransformer(unused_imports))

     # Use difflib to show the changes to verify unused imports are removed as expected.
     print(
         "".join(
             difflib.unified_diff(source.splitlines(1), fixed_module.code.splitlines(1))
         )
     )
```

```
---
+++
@@ -1,6 +1,5 @@
-import a, b, c as d, e as f  # expect to keep: a, c as d
-from g import h, i, j as k, l as m  # expect to keep: h, j as k
-from n import o  # expect to be removed entirely
+import a, c as d  # expect to keep: a, c as d
+from g import h, j as k  # expect to keep: h, j as k

 a()
```

WORKING WITH MATCHERS

Matchers provide a flexible way of comparing LibCST nodes in order to build more complex transforms. See *Matchers* for the complete documentation.

6.1 Basic Matcher Usage

Let's say you are visiting a LibCST `Call` node and you want to know if all arguments provided are the literal `True` or `False`. You look at the documentation and see that `Call.args` is a sequence of `Arg`, and each `Arg.value` is a `BaseExpression`. In order to verify that each argument is either `True` or `False` you would have to first loop over `node.args`, and then check `isinstance(arg.value, cst.Name)` for each `arg` in the loop before finally checking `arg.value.value in ("True", "False")`.

Here's a short example of that in action:

```python
[2]:  import libcst as cst

      def is_call_with_booleans(node: cst.Call) -> bool:
          for arg in node.args:
              if not isinstance(arg.value, cst.Name):
                  # This can't be the literal True/False, so bail early.
                  return False
              if cst.ensure_type(arg.value, cst.Name).value not in ("True", "False"):
                  # This is a Name node, but not the literal True/False, so bail.
                  return False
          # We got here, so all arguments are literal boolean values.
          return True
```

We can see from a few examples that this does work as intended. However, it is an awful lot of boilerplate that was fairly cumbersome to write.

```python
[3]:  call_1 = cst.Call(
          func=cst.Name("foo"),
          args=(
              cst.Arg(cst.Name("True")),
          ),
      )
      is_call_with_booleans(call_1)
```

```python
[3]:  True
```

```
[4]: call_2 = cst.Call(
         func=cst.Name("foo"),
         args=(
             cst.Arg(cst.Name("None")),
         ),
     )
     is_call_with_booleans(call_2)
```

```
[4]: False
```

Let's try to do a bit better with matchers. We can make a better function that takes advantage of matchers to get rid of both the instance check and the `ensure_type` call, like so:

```
[5]: import libcst.matchers as m

     def better_is_call_with_booleans(node: cst.Call) -> bool:
         for arg in node.args:
             if not m.matches(arg.value, m.Name("True") | m.Name("False")):
                 # Oops, this isn't a True/False literal!
                 return False
         # We got here, so all arguments are literal boolean values.
         return True
```

This is a lot shorter and is easier to read as well! We made use of the fact that matchers handles instance checking for us in a safe way. We also made use of the fact that matchers allows us to concisely express multiple match options with the use of Python's or operator. We can also see that it still works on our previous examples:

```
[6]: better_is_call_with_booleans(call_1)
```

```
[6]: True
```

```
[7]: better_is_call_with_booleans(call_2)
```

```
[7]: False
```

We still have one more trick up our sleeve though. Matchers don't just allow us to specify which attributes we want to match on exactly. It also allows us to specify rules for matching sequences of nodes, like the list of *Arg* nodes that appears in *Call*. Let's make use of that, turning our original is_call_with_booleans function into a call to *matches()*:

```
[8]: def best_is_call_with_booleans(node: cst.Call) -> bool:
         return m.matches(
             node,
             m.Call(
                 args=(
                     m.ZeroOrMore(m.Arg(m.Name("True") | m.Name("False"))),
                 ),
             ),
         )
```

We've turned our original function into a single call to *matches()*. As an added benefit, the match node can be read from left to right in a way that makes sense in english: "match any call with zero or more arguments that are the literal `True` or `False`". As we can see, it works as intended:

```
[9]: best_is_call_with_booleans(call_1)
```

```
[9]: True
```

```
[10]: best_is_call_with_booleans(call_2)
```

```
[10]: False
```

6.2 Matcher Decorators

You can already do a lot with just `matches()`. It lets you define the shape of nodes you want to match and LibCST takes care of the rest. However, you still need to include a lot of boilerplate into your *Visitors* in order to identify which nodes you care about. Matcher *Decorators* help reduce that boilerplate.

Say you wanted to invert the boolean literals in functions which match the above `best_is_call_with_booleans`. You could build something that looks like the following:

```python
[11]: class BoolInverter(cst.CSTTransformer):
          def __init__(self) -> None:
              self.in_call: int = 0

          def visit_Call(self, node: cst.Call) -> None:
              if m.matches(node, m.Call(args=(
                  m.ZeroOrMore(m.Arg(m.Name("True") | m.Name("False"))),
              ))):
                  self.in_call += 1

          def leave_Call(self, original_node: cst.Call, updated_node: cst.Call) -> cst.Call:
              if m.matches(original_node, m.Call(args=(
                  m.ZeroOrMore(m.Arg(m.Name("True") | m.Name("False"))),
              ))):
                  self.in_call -= 1
              return updated_node

          def leave_Name(self, original_node: cst.Name, updated_node: cst.Name) -> cst.Name:
              if self.in_call > 0:
                  if updated_node.value == "True":
                      return updated_node.with_changes(value="False")
                  if updated_node.value == "False":
                      return updated_node.with_changes(value="True")
              return updated_node
```

We can try it out on a source file to see that it works:

```python
[12]: source = "def some_func(*params: object) -> None:\n    pass\n\nsome_func(True,␣
      ↪False)\nsome_func(1, 2, 3)\nsome_func()\n"
      module = cst.parse_module(source)
      print(source)
```

```
def some_func(*params: object) -> None:
    pass
```

(continues on next page)

```
some_func(True, False)
some_func(1, 2, 3)
some_func()
```

[13]:
```
new_module = module.visit(BoolInverter())
print(new_module.code)
```

```
def some_func(*params: object) -> None:
    pass

some_func(False, True)
some_func(1, 2, 3)
some_func()
```

While this works its not super elegant. We have to track where we are in the tree so we know when its safe to invert boolean literals which means we have to create a constructor and we have to duplicate matching logic. We could refactor that into a helper like the `best_is_call_with_booleans` above, but it only makes things so much better.

So, let's try rewriting it with matcher decorators instead. Note that this includes changing the class we inherit from to *MatcherDecoratableTransformer* in order to enable the matcher decorator feature:

[14]:
```
class BetterBoolInverter(m.MatcherDecoratableTransformer):
    @m.call_if_inside(m.Call(args=(
        m.ZeroOrMore(m.Arg(m.Name("True") | m.Name("False"))),
    )))
    def leave_Name(self, original_node: cst.Name, updated_node: cst.Name) -> cst.Name:
        if updated_node.value == "True":
            return updated_node.with_changes(value="False")
        if updated_node.value == "False":
            return updated_node.with_changes(value="True")
        return updated_node
```

[15]:
```
new_module = module.visit(BetterBoolInverter())
print(new_module.code)
```

```
def some_func(*params: object) -> None:
    pass

some_func(False, True)
some_func(1, 2, 3)
some_func()
```

Using matcher decorators we successfully removed all of the boilerplate around state tracking! The only thing that `leave_Name` needs to concern itself with is the actual business logic of the transform. However, it still needs to check to see if the value of the node should be inverted. This is because the `Call.func` is a *Name* in this case. Let's use another matcher decorator to make that problem go away:

[16]:
```
class BestBoolInverter(m.MatcherDecoratableTransformer):
    @m.call_if_inside(m.Call(args=(
```

```
        m.ZeroOrMore(m.Arg(m.Name("True") | m.Name("False"))),
    )))
    @m.leave(m.Name("True") | m.Name("False"))
    def invert_bool_literal(self, original_node: cst.Name, updated_node: cst.Name) ->
→cst.Name:
        return updated_node.with_changes(value="False" if updated_node.value == "True
→" else "True")
```

[17]:
```
new_module = module.visit(BestBoolInverter())
print(new_module.code)
```

```
def some_func(*params: object) -> None:
    pass

some_func(False, True)
some_func(1, 2, 3)
some_func()
```

That's it! Instead of using a leave_Name which modifies all *Name* nodes we instead created a matcher visitor that only modifies *Name* nodes with the value of True or False. We decorate *that* with *call_if_inside()* to ensure we run this on *Name* nodes found inside of function calls that only take boolean literals. Using two matcher decorators we got rid of all of the state management as well as all of the cases where we needed to handle nodes we weren't interested in.

WORKING WITH CODEMODS

Codemods are an abstraction on top of LibCST for performing large-scale changes to an entire codebase. See *Codemods* for the complete documentation.

7.1 Setting up and Running Codemods

Let's say you were interested in converting legacy `.format()` calls to shiny new Python 3.6 f-strings. LibCST ships with a command-line interface known as `libcst.tool`. This includes a few provisions for working with codemods at the command-line. It also includes a library of pre-defined codemods, one of which is a transform that can convert most `.format()` calls to f-strings. So, let's use this to give Python 3.6 f-strings a try.

You might be lucky enough that the defaults for LibCST perfectly match your coding style, but chances are you want to customize LibCST to your repository. Initialize your repository by running the following command in the root of your repository and then edit the produced `.libcst.codemod.yaml` file:

```
python3 -m libcst.tool initialize .
```

The file includes provisions for customizing any generated code marker, calling an external code formatter such as black, blackisting patterns of files you never wish to touch and a list of modules that contain valid codemods that can be executed. If you want to write and run codemods specific to your repository or organization, you can add an in-repo module location to the list of modules and LibCST will discover codemods in all locations.

Now that your repository is initialized, let's have a quick look at what's currently available for running. Run the following command from the root of your repository:

```
python3 -m libcst.tool list
```

You'll see several codemods available to you, one of which is `convert_format_to_fstring.ConvertFormatStringCommand`. The description to the right of this codemod indicates that it converts `.format()` calls to f-strings, so let's give it a whirl! Execute the codemod from the root of your repository like so:

```
python3 -m libcst.tool codemod convert_format_to_fstring.ConvertFormatStringCommand .
```

If you want to try it out on only one file or a specific subdirectory, you can replace the `.` in the above command with a relative directory, file, list of directories or list of files. While LibCST is walking through your repository and codemodding files you will see a progress indicator. If there's anything the codemod can't do or any unexpected syntax errors, you will also see them on your console as it progresses.

If everything works out, you'll notice that your `.format()` calls have been converted to f-strings!

7.2 Writing a Codemod

Codemods use the same principles as the rest of LibCST. They take LibCST's core, metadata and matchers and package them up as a simple command-line interface. So, anything you can do with LibCST in isolation you can also do with a codemod.

Let's say you need to clean up some legacy code which used magic values instead of constants. You've already got a constants module called `utils.constants` and you want to assume that every reference to a raw string matching a particular constant should be converted to that constant. For the simplest version of this codemod, you'll need a command-line tool that takes as arguments the string to replace and the constant to replace it with. You'll also need to ensure that modified modules import the constant itself.

So, you can write something similar to the following:

```python
import argparse
from ast import literal_eval
from typing import Union

import libcst as cst
from libcst.codemod import CodemodContext, VisitorBasedCodemodCommand
from libcst.codemod.visitors import AddImportsVisitor


class ConvertConstantCommand(VisitorBasedCodemodCommand):

    # Add a description so that future codemodders can see what this does.
    DESCRIPTION: str = "Converts raw strings to constant accesses."

    @staticmethod
    def add_args(arg_parser: argparse.ArgumentParser) -> None:
        # Add command-line args that a user can specify for running this
        # codemod.
        arg_parser.add_argument(
            "--string",
            dest="string",
            metavar="STRING",
            help="String contents that we should look for.",
            type=str,
            required=True,
        )
        arg_parser.add_argument(
            "--constant",
            dest="constant",
            metavar="CONSTANT",
            help="Constant identifier we should replace strings with.",
            type=str,
            required=True,
        )

    def __init__(self, context: CodemodContext, string: str, constant: str) -> None:
        # Initialize the base class with context, and save our args. Remember, the
        # "dest" for each argument we added above must match a parameter name in
        # this init.
        super().__init__(context)
        self.string = string
        self.constant = constant
```

(continues on next page)

```python
    def leave_SimpleString(
        self, original_node: cst.SimpleString, updated_node: cst.SimpleString
    ) -> Union[cst.SimpleString, cst.Name]:
        if literal_eval(updated_node.value) == self.string:
            # Check to see if the string matches what we want to replace. If so,
            # then we do the replacement. We also know at this point that we need
            # to import the constant itself.
            AddImportsVisitor.add_needed_import(
                self.context, "utils.constants", self.constant,
            )
            return cst.Name(self.constant)
        # This isn't a string we're concerned with, so leave it unchanged.
        return updated_node
```

This codemod is pretty simple. It defines a command-line description, sets up to parse a few required command-line args, initializes its own member variables with the command-line args that were parsed for it by `libcst.tool codemod` and finally replaces any string which matches our string command-line argument with a constant. It also takes care of adding the import required for the constant to be defined properly.

Cool! Let's look at the command-line help for this codemod. Let's assume you saved it as `constant_folding.py` inside `libcst.codemod.commands`. You can get help for the codemod by running the following command:

```
python3 -m libcst.tool codemod constant_folding.ConvertConstantCommand --help
```

Notice that along with the default arguments, the `--string` and `--constant` arguments are present in the help, and the command-line description has been updated with the codemod's description string. You'll notice that the codemod also shows up on `libcst.tool list`.

7.3 Testing Codemods

Instead of iterating on a codemod by running it repeatedly on a codebase and seeing what happens, we can write a series of unit tests that assert on desired transformations. Given the above constant folding codemod that we wrote, we can test it with some code similar to the following:

```python
from libcst.codemod import CodemodTest
from libcst.codemod.commands.constant_folding import ConvertConstantCommand


class TestConvertConstantCommand(CodemodTest):

    # The codemod that will be instantiated for us in assertCodemod.
    TRANSFORM = ConvertConstantCommand

    def test_noop(self) -> None:
        before = """
            foo = "bar"
        """
        after = """
            foo = "bar"
        """

        # Verify that if we don't have a valid string match, we don't make
        # any substitutions.
        self.assertCodemod(before, after, string="baz", constant="BAZ")
```

```python
    def test_substitution(self) -> None:
        before = """
            foo = "bar"
        """
        after = """
            from utils.constants import BAR

            foo = BAR
        """

        # Verify that if we do have a valid string match, we make a substitution
        # as well as import the constant.
        self.assertCodemod(before, after, string="bar", constant="BAR")
```

If we save this as `test_constant_folding.py` inside `libcst.codemod.commands.tests` then we can execute the tests with the following line:

```
python3 -m unittest libcst.codemod.commands.tests.test_constant_folding
```

That's all there is to it!

BEST PRACTICES

While there are plenty of ways to interact with LibCST, we recommend some patterns over others. Various best practices are laid out here along with their justifications.

8.1 Avoid `isinstance` when traversing

Excessive use of `isinstance` implies that you should rewrite your check as a matcher or unroll it into a set of visitor methods. Often, you should make use of ensure_type() to make your type checker aware of a node's type.

Often it is far easier to use *Matchers* over explicit instance checks in a transform. Matching against some pattern and then extracting a value from a node's child is often easier and far more readable. Unfortunately this clashes with various type-checkers which do not understand that *matches()* guarantees a particular set of children. Instead of instance checks, you should use ensure_type() which can be inlined and nested.

For example, if you have written the following:

```python
def get_identifier_name(node: cst.CSTNode) -> Optional[str]:
    if m.matches(node, m.Name()):
        assert isinstance(node, cst.Name)
        return node.value
    return None
```

You could instead write something like:

```python
def get_identifier_name(node: cst.CSTNode) -> Optional[str]:
    return (
        cst.ensure_type(node, cst.Name).value
        if m.matches(node, m.Name())
        else None
    )
```

If you find yourself attempting to manually traverse a tree using `isinstance`, you can often rewrite your code using visitor methods instead. Nested instance checks can often be unrolled into visitors methods along with matcher decorators. This may entail adding additional state to your visitor, but the resulting code is far more likely to work after changes to LibCST itself. For example, if you have written the following:

```python
class CountBazFoobarArgs(cst.CSTVisitor):
    """
    Given a set of function names, count how many arguments to those function
    calls are the identifiers "baz" or "foobar".
    """

    def __init__(self, functions: Set[str]) -> None:
```

(continues on next page)

```python
        super().__init__()
        self.functions: Set[str] = functions
        self.arg_count: int = 0

    def visit_Call(self, node: cst.Call) -> None:
        # See if the call itself is one of our functions we care about
        if isinstance(node.func, cst.Name) and node.func.value in self.functions:
            # Loop through each argument
            for arg in node.args:
                # See if the argument is an identifier matching what we want to count
                if isinstance(arg.value, cst.Name) and arg.value.value in {"baz",
↪"foobar"}:
                    self.arg_count += 1
```

You could instead write something like:

```python
class CountBazFoobarArgs(m.MatcherDecoratableVisitor):
    """
    Given a set of function names, count how many arguments to those function
    calls are the identifiers "baz" or "foobar".
    """

    def __init__(self, functions: Set[str]) -> None:
        super().__init__()
        self.functions: Set[str] = functions
        self.arg_count: int = 0
        self.call_stack: List[str] = []

    def visit_Call(self, node: cst.Call) -> None:
        # Store all calls in a stack
        if m.matches(node.func, m.Name()):
            self.call_stack.append(cst.ensure_type(node.func, cst.Name).value)

    def leave_Call(self, original_node: cst.Call) -> None:
        # Pop the latest call off the stack
        if m.matches(node.func, m.Name()):
            self.call_stack.pop()

    @m.visit(m.Arg(m.Name("baz") | m.Name("foobar")))
    def _count_args(self, node: cst.Arg) -> None:
        # See if the most shallow call is one we're interested in, so we can
        # count the args we care about only in calls we care about.
        if self.call_stack[-1] in self.functions:
            self.arg_count += 1
```

While there is more code than the previous example, it is arguably easier to understand and maintain each part of the code. It is also immune to any future changes to LibCST which change's the tree shape. Note that LibCST is traversing the tree completely in both cases, so while the first appears to be faster, it is actually doing the same amount of work as the second.

8.2 Prefer `updated_node` when modifying trees

When you are using *CSTTransformer* to modify a LibCST tree, only return modifications to `updated_node`. The `original_node` parameter on any `leave_<Node>` method is provided for book-keeping and is guaranteed to be equal via == and is checks to the `node` parameter in the corresponding `visit_<Node>` method. Remember

that LibCST trees are immutable, so the only way to make a modification is to return a new tree. Hence, by the time we get to calling `leave_<Node>` methods, we have an updated tree whose children have been modified. Therefore, you should only return `original_node` when you want to explicitly discard changes performed on the node's children.

Say you wanted to rename all function calls which were calling global functions. So, you might write the following:

```python
class FunctionRenamer(cst.CSTTransformer):
    def leave_Call(self, original_node: cst.Call, updated_node: cst.Call) -> cst.Call:
        if m.matches(original_node.func, m.Name()):
            return original_node.with_changes(
                func=cst.Name(
                    "renamed_" + cst.ensure_type(original_node.func, cst.Name).value
                )
            )
        return original_node
```

Consider writing instead:

```python
class FunctionRenamer(cst.CSTTransformer):
    def leave_Call(self, original_node: cst.Call, updated_node: cst.Call) -> cst.Call:
        if m.matches(updated_node.func, m.Name()):
            return updated_node.with_changes(
                func=cst.Name(
                    "renamed_" + cst.ensure_type(updated_node.func, cst.Name).value
                )
            )
        return updated_node
```

The version that returns modifications to `original_node` has a subtle bug. Consider the following code snippet:

```python
some_func(1, 2, other_func(3))
```

Running the recommended transform will return us a new code snippet that looks like this:

```python
renamed_some_func(1, 2, renamed_other_func(3))
```

However, running the version which modifies `original_node` will instead return:

```python
renamed_some_func(1, 2, other_func(3))
```

That's because the `updated_node` tree contains the modification to `other_func`. By returning modifications to `original_node` instead of `updated_node`, we accidentally discarded all the work done deeper in the tree.

8.3 Provide a `config` when generating code from templates

When generating complex trees it is often far easier to pass a string to *parse_statement()* or *parse_expression()* than it is to manually construct the tree. When using these functions to generate code, you should always use the `config` parameter in order to generate code that matches the defaults of the module you are modifying. The *Module* class even has a helper attribute *config_for_parsing* to make it easy to use. This ensures that line endings and indentation are consistent with the defaults in the module you are adding the code to.

For example, to add a print statement to the end of a module:

```python
module = cst.parse_module(some_code_string)
new_module = module.with_changes(
    body=(
```

```
        *module.body,
        cst.parse_statement(
            "print('Hello, world!')",
            config=module.config_for_parsing,
        ),
    ),
)
new_code_string = new_module.code
```

Leaving out the `config` parameter means that LibCST will assume some defaults and could result in added code which is formatted differently than the rest of the module it was added to. In the above example, because we used the config from the already-parsed example, the print statement will be added with line endings matching the rest of the module. If we neglect the `config` parameter, we might accidentally insert a windows line ending into a unix file or vice versa, depending on what system we ran the code under.

PARSING

The parser functions accept source code and an optional configuration object, and will generate *CSTNode* objects.

parse_module() is the most useful function here, since it accepts the entire contents of a file and returns a new tree, but *parse_expression()* and *parse_statement()* are useful when inserting new nodes into the tree, because they're easier to use than the equivalent node constructors.

```
>>> import libcst as cst
>>> cst.parse_expression("1 + 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=[],
)
```

libcst.**parse_module**(*source: Union[str, bytes], config: libcst._parser.types.config.PartialParserConfig = PartialParserConfig())* → libcst._nodes.module.Module
    Accepts an entire python module, including all leading and trailing whitespace.

    If source is bytes, the encoding will be inferred and preserved. If the source is a string, we will default to assuming UTF-8 encoding if the module is rendered back out to source as bytes. It is recommended that when calling *parse_module()* with a string you access the serialized code using *Module*'s code attribute, and when calling it with bytes you access the serialized code using *Module*'s bytes attribute.

libcst.**parse_expression**(*source: str, config: libcst._parser.types.config.PartialParserConfig = PartialParserConfig())* → libcst._nodes.expression.BaseExpression
    Accepts an expression on a single line. Leading and trailing whitespace is not valid (there's nowhere to store it on the expression node). *parse_expression()* is provided mainly as a convenience function to generate semi-complex trees from code snippets. If you need to represent an expression exactly, including all leading/trailing

comments, you should instead use *parse_module()*.

libcst.**parse_statement**(*source: str, config: libcst._parser.types.config.PartialParserConfig = PartialParserConfig()*) → Union[libcst._nodes.statement.SimpleStatementLine, libcst._nodes.statement.BaseCompoundStatement]

Accepts a statement followed by a trailing newline. If a trailing newline is not provided, one will be added. *parse_statement()* is provided mainly as a convenience function to generate semi-complex trees from code snippetes. If you need to represent a statement exactly, including all leading/trailing comments, you should instead use *parse_module()*.

Leading comments and trailing comments (on the same line) are accepted, but whitespace (or anything else) after the statement's trailing newline is not valid (there's nowhere to store it on the statement node). Note that since there is nowhere to store leading and trailing comments/empty lines, code rendered out from a parsed statement using cst.Module([]).code_for_node(statement) will not include leading/trailing comments.

class libcst.**PartialParserConfig**

An optional object that can be supplied to the parser entrypoints (e.g. *parse_module()*) to configure the parser.

Unspecified fields will be inferred from the input source code or from the execution environment.

```
>>> import libcst as cst
>>> tree = cst.parse_module("abc")
>>> tree.bytes
b'abc'
>>> # override the default utf-8 encoding
... tree = cst.parse_module("abc", cst.PartialParserConfig(encoding="utf-32"))
>>> tree.bytes
b'\xff\xfe\x00\x00a\x00\x00\x00b\x00\x00\x00c\x00\x00\x00'
```

python_version : typing.Union[str, types.config.AutoConfig] = AutoConfig.token

The version of Python that the input source code is expected to be syntactically compatible with. This may be different from the Python interpreter being used to run LibCST. For example, you can parse code as 3.7 with a CPython 3.6 interpreter.

If unspecified, it will default to the syntax of the running interpreter (rounding down from among the following list).

Currently, only Python 3.0, 3.1, 3.3, 3.5, 3.6, 3.7 and 3.8 syntax is supported. The gaps did not have any syntax changes from the version prior.

parsed_python_version : parso.utils.PythonVersionInfo

A named tuple with the major and minor Python version numbers. This is derived from *python_version* and should not be supplied to the *PartialParserConfig* constructor.

encoding : typing.Union[str, types.config.AutoConfig] = AutoConfig.token

The file's encoding format. When parsing a bytes object, this value may be inferred from the contents of the parsed source code. When parsing a str, this value defaults to "utf-8".

future_imports : typing.Union[typing.FrozenSet[str], types.config.AutoConfig] = AutoCo

Detected __future__ import names

default_indent : typing.Union[str, types.config.AutoConfig] = AutoConfig.token

The indentation of the file, expressed as a series of tabs and/or spaces. This value is inferred from the contents of the parsed source code by default.

default_newline : typing.Union[str, types.config.AutoConfig] = AutoConfig.token

The newline of the file, expressed as \n, \r\n, or \r. This value is inferred from the contents of the parsed source code by default.

9.1 Syntax Errors

class libcst.**ParserSyntaxError**

> Contains an error encountered while trying to parse a piece of source code. This exception shouldn't be constructed directly by the user, but instead may be raised by calls to *parse_module()*, *parse_expression()*, or *parse_statement()*.
>
> This does not inherit from SyntaxError because Python's may raise a SyntaxError for any number of reasons, potentially leading to unintended behavior.
>
> **message**
>
> > A human-readable explanation of the syntax error without information about where the error occurred.
> >
> > For a human-readable explanation of the error alongside information about where it occurred, use *__str__()* (via str(ex)) instead.
>
> **raw_line**
>
> > The one-indexed line where the error occured.
>
> **raw_column**
>
> > The zero-indexed column as a number of characters from the start of the line where the error occured.
>
> **__str__**() → str
>
> > A multi-line human-readable error message of where the syntax error is in their code. For example:
> >
> > ```
> > Syntax Error @ 2:1.
> > Incomplete input. Encountered end of file (EOF), but expected 'except', or
> > ↪'finally'.
> >
> > try: pass
> >   ^
> > ```
>
> **property context**
>
> > A formatted string containing the line of code with the syntax error (or a non-empty line above it) along with a caret indicating the exact column where the error occurred.
> >
> > Return None if there's no relevant non-empty line to show. (e.g. the file consists of only blank lines)
>
> **property editor_line**
>
> > The expected one-indexed line in the user's editor. This is the same as *raw_line*.
>
> **property editor_column**
>
> > The expected one-indexed column that's likely to match the behavior of the user's editor, assuming tabs expand to 1-8 spaces. This is the column number shown when the syntax error is printed out with *str*.
> >
> > This assumes single-width characters. However, because python doesn't ship with a wcwidth function, it's hard to handle this properly without a third-party dependency.
> >
> > For a raw zero-indexed character offset without tab expansion, see *raw_column*.

TEN

NODES

CSTNode and its subclasses cover Python's full grammar in a whitespace-sensitive fashion, forming LibCST's concrete syntax tree.

Many of these nodes are designed to behave similarly to Python's abstract syntax tree.

10.1 CSTNode

The base node type which all other nodes derive from.

class libcst.**CSTNode**

> **validate_types_shallow**() → None
> Compares the type annotations on a node's fields with those field's actual values at runtime. Raises a TypeError is a mismatch is found.
>
> Only validates the current node, not any of it's children. For a recursive version, see *validate_types_deep()*.
>
> If you're using a static type checker (highly recommended), this is useless. However, if your code doesn't use a static type checker, or if you're unable to statically type your code for some reason, you can use this method to help validate your tree.
>
> Some (non-typing) validation is done unconditionally during the construction of a node. That validation does not overlap with the work that *validate_types_deep()* does.
>
> **validate_types_deep**() → None
> Like *validate_types_shallow()*, but recursively validates the whole tree.
>
> **property children**
> The immediate (not transitive) child CSTNodes of the current node. Various properties on the nodes, such as string values, will not be visited if they are not a subclass of CSTNode.
>
> Iterable properties of the node (e.g. an IndentedBlock's body) will be flattened into the children's sequence.
>
> The children will always be returned in the same order that they appear lexically in the code.
>
> **visit**()
> Visits the current node, its children, and all transitive children using the given visitor's callbacks.
>
> **with_changes**(****changes*) → _CSTNodeSelfT
> A convenience method for performing mutation-like operations on immutable nodes. Creates a new object of the same type, replacing fields with values from the supplied keyword arguments.
>
> For example, to update the test of an if conditional, you could do:

```
def leave_If(self, original_node: cst.If, updated_node: cst.If) -> cst.If:
    new_node = updated_node.with_changes(test=new_conditional)
    return new_node
```

`new_node` will have the same `body`, `orelse`, and whitespace fields as `updated_node`, but with the updated `test` field.

The accepted arguments match the arguments given to `__init__`, however there are no required or positional arguments.

TODO: This API is untyped. There's probably no sane way to type it using pyre's current feature-set, but we should still think about ways to type this or a similar API in the future.

deep_clone() → _CSTNodeSelfT
    Recursively clone the entire tree. The created tree is a new tree has the same representation but different identity.

```
>>> tree = cst.parse_expression("1+2")
```

```
>>> tree.deep_clone() == tree
False
```

```
>>> tree == tree
True
```

```
>>> tree.deep_equals(tree.deep_clone())
True
```

deep_equals(*other: libcst._nodes.base.CSTNode*) → bool
    Recursively inspects the entire tree under `self` and `other` to determine if the two trees are equal by representation instead of identity (==).

deep_replace(*old_node: libcst._nodes.base.CSTNode*,   *new_node: CSTNodeT*) → Union[_CSTNodeSelfT, CSTNodeT]
    Recursively replaces any instance of `old_node` with `new_node` by identity. Use this to avoid nested `with_changes` blocks when you are replacing one of a node's deep children with a new node. Note that if you have previously modified the tree in a way that `old_node` appears more than once as a deep child, all instances will be replaced.

deep_remove(*old_node: libcst._nodes.base.CSTNode*)   →   Union[_CSTNodeSelfT, libcst._removal_sentinel.RemovalSentinel]
    Recursively removes any instance of `old_node` by identity. Note that if you have previously modified the tree in a way that `old_node` appears more than once as a deep child, all instances will be removed.

with_deep_changes(*old_node: libcst._nodes.base.CSTNode*, ***changes*) → _CSTNodeSelfT
    A convenience method for applying *with_changes* to a child node. Use this to avoid chains of *with_changes* or combinations of *deep_replace* and *with_changes*.

The accepted arguments match the arguments given to the child node's `__init__`.

TODO: This API is untyped. There's probably no sane way to type it using pyre's current feature-set, but we should still think about ways to type this or a similar API in the future.

classmethod field(**args*, ***kwargs*) → Any
    A helper that allows us to easily use CSTNodes in dataclass constructor defaults without accidentally aliasing nodes by identity across multiple instances.

10.2 Module

A node that represents an entire python module.

class libcst.**Module**

Contains some top-level information inferred from the file letting us set correct defaults when printing the tree about global formatting rules. All code parsed with *parse_module()* will be encapsulated in a module.

body : typing.Sequence[typing.Union[statement.SimpleStatementLine, statement.BaseCompo...

A list of zero or more statements that make up this module.

header : typing.Sequence[whitespace.EmptyLine]

Normally any whitespace/comments are assigned to the next node visited, but *Module* is a special case, and comments at the top of the file tend to refer to the module itself, so we assign them to the *Module* instead of the first statement in the body.

footer : typing.Sequence[whitespace.EmptyLine]

Any trailing whitespace/comments found after the last statement.

encoding : str = "utf-8"

The file's encoding format. When parsing a bytes object, this value may be inferred from the contents of the parsed source code. When parsing a str, this value defaults to "utf-8".

This value affects how *bytes* encodes the source code.

default_indent : str = " "

The indentation of the file, expressed as a series of tabs and/or spaces. This value is inferred from the contents of the parsed source code by default.

default_newline : str = " "

The newline of the file, expressed as \n, \r\n, or \r. This value is inferred from the contents of the parsed source code by default.

has_trailing_newline : bool = True

Whether the module has a trailing newline or not.

visit(*visitor: Union[CSTTransformer, CSTVisitor]*) → _ModuleSelfT

Returns the result of running a visitor over this module.

Module overrides the default visitor entry point to resolve metadata dependencies declared by 'visitor'.

property code

The string representation of this module, respecting the inferred indentation and newline type.

property bytes

The bytes representation of this module, respecting the inferred indentation and newline type, using the current encoding.

code_for_node(*node: libcst._nodes.base.CSTNode*) → str

Generates the code for the given node in the context of this module. This is a method of Module, not CSTNode, because we need to know the module's default indentation and newline formats.

property config_for_parsing

Generates a parser config appropriate for passing to a *parse_expression()* or *parse_statement()* call. This is useful when using either parser function to generate code from a string template. By using a generated parser config instead of the default, you can guarantee that trees generated from both statement and expression strings have the same inferred defaults for things like newlines, indents and similar:

```
module = cst.parse_module("pass\n")
expression = cst.parse_expression("1 + 2", config=module.config_for_parsing)
```

get_docstring()
>   Returns a `inspect.cleandoc()` cleaned docstring if the docstring is available, `None` otherwise.

10.3 Expressions

An expression is anything that represents a value (e.g. it could be returned from a function). All expressions subclass from *BaseExpression*.

Expression can be parsed with *parse_expression()* or as part of a statement or module using *parse_statement()* or *parse_module()*.

class libcst.**BaseExpression**
>   An base class for all expressions. *BaseExpression* contains no fields.

10.3.1 Names and Object Attributes

class libcst.**Name**
>   A simple variable name. Names are typically used in the context of a variable access, an assignment, or a deletion.
>
>   Dotted variable names (`a.b.c`) are represented with *Attribute* nodes, and subscripted variable names (`a[b]`) are represented with *Subscript* nodes.
>
>   **value : str**
>   >   The variable's name (or "identifier") as a string.
>
>   **lpar : typing.Sequence[expression.LeftParen]**
>
>   **rpar : typing.Sequence[expression.RightParen]**
>   >   Sequence of parenthesis for precedence dictation.

class libcst.**Attribute**
>   An attribute reference, such as `x.y`.
>
>   Note that in the case of `x.y.z`, the outer attribute will have an attr of `z` and the value will be another *Attribute* referencing the `y` attribute on `x`:

```
Attribute(
    value=Attribute(
        value=Name("x")
        attr=Name("y")
    ),
    attr=Name("z"),
)
```

>   **value : expression.BaseExpression**
>   >   An expression which, when evaluated, will produce an object with `attr` as an attribute.
>
>   **attr : expression.Name**
>   >   The name of the attribute being accessed on the `value` object.
>
>   **dot : op.Dot = Dot(whitespace_before=SimpleWhitespace(value='',), whitespace_after=**
>   >   A separating dot. If there's whitespace between the `value` and `attr`, this dot owns it.
>
>   **lpar : typing.Sequence[expression.LeftParen]**
>
>   **rpar : typing.Sequence[expression.RightParen]**
>   >   Sequence of parenthesis for precedence dictation.

10.3.2 Operations and Comparisons

Operation and Comparison nodes combine one or more expressions with an *operator*.

class libcst.**UnaryOperation**

> Any generic unary expression, such as not x or -x. *UnaryOperation* nodes apply a *BaseUnaryOp* to an expression.
>
> **operator : op.BaseUnaryOp**
>> The unary operator that applies some operation (e.g. negation) to the expression.
>
> **expression : expression.BaseExpression**
>> The expression that should be transformed (e.g. negated) by the operator to create a new value.
>
> **lpar : typing.Sequence[expression.LeftParen]**
>
> **rpar : typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

class libcst.**BinaryOperation**

> An operation that combines two expression such as x << y or y + z. *BinaryOperation* nodes apply a *BaseBinaryOp* to an expression.
>
> Binary operations do not include operations performed with *BaseBooleanOp* nodes, such as and or or. Instead, those operations are provided by *BooleanOperation*.
>
> It also does not include support for comparision operators performed with *BaseCompOp*, such as <, >=, ==, is, or in. Instead, those operations are provided by *Comparison*.
>
> **left : expression.BaseExpression**
>> The left hand side of the operation.
>
> **operator : op.BaseBinaryOp**
>> The actual operator such as << or + that combines the left and right expressions.
>
> **right : expression.BaseExpression**
>> The right hand side of the operation.
>
> **lpar : typing.Sequence[expression.LeftParen]**
>
> **rpar : typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

class libcst.**BooleanOperation**

> An operation that combines two booleans such as x or y or z and w *BooleanOperation* nodes apply a *BaseBooleanOp* to an expression.
>
> Boolean operations do not include operations performed with *BaseBinaryOp* nodes, such as + or <<. Instead, those operations are provided by *BinaryOperation*.
>
> It also does not include support for comparision operators performed with *BaseCompOp*, such as <, >=, ==, is, or in. Instead, those operations are provided by *Comparison*.
>
> **left : expression.BaseExpression**
>> The left hand side of the operation.
>
> **operator : op.BaseBooleanOp**
>> The actual operator such as and or or that combines the left and right expressions.
>
> **right : expression.BaseExpression**
>> The right hand side of the operation.
>
> **lpar : typing.Sequence[expression.LeftParen]**

> **rpar : typing.Sequence[expression.RightParen]**
> Sequence of parenthesis for precedence dictation.

class libcst.**Comparison**

A comparison between multiple values such as x < y, x < y < z, or x in [y, z]. These comparisions typically result in boolean values.

Unlike *BinaryOperation* and *BooleanOperation*, comparisons are not restricted to a left and right child. Instead they can contain an arbitrary number of *ComparisonTarget* children.

x < y < z is not equivalent to (x < y) < z or x < (y < z). Instead, it's roughly equivalent to x < y and y < z.

For more details, see Python's documentation on comparisons.

```
# x < y < z

Comparison(
    Name("x"),
    [
        ComparisonTarget(LessThan(), Name("y")),
        ComparisonTarget(LessThan(), Name("z")),
    ],
)
```

> **left : expression.BaseExpression**
> The first value in the full sequence of values to compare. This value will be compared against the first value in comparisions.

> **comparisons : typing.Sequence[expression.ComparisonTarget]**
> Pairs of *BaseCompOp* operators and expression values to compare. These come after left. Each value is compared against the value before and after itself in the sequence.

> **lpar : typing.Sequence[expression.LeftParen]**

> **rpar : typing.Sequence[expression.RightParen]**
> Sequence of parenthesis for precedence dictation.

class libcst.**ComparisonTarget**

A target for a *Comparison*. Owns the comparison operator and the value to the right of the operator.

> **operator : op.BaseCompOp**
> A comparison operator such as <, >=, ==, is, or in.

> **comparator : expression.BaseExpression**
> The right hand side of the comparison operation.

10.3.3 Control Flow

class libcst.**Asynchronous**

Used by asynchronous function definitions, as well as async for and async with.

> **whitespace_after : whitespace.SimpleWhitespace**
> Any space that appears directly after this async keyword.

class libcst.**Await**

An await expression. Await expressions are only valid inside the body of an asynchronous *FunctionDef* or (as of Python 3.7) inside of an asynchronous *GeneratorExp* nodes.

> **expression : expression.BaseExpression**
> The actual expression we need to wait for.

       lpar : typing.Sequence[expression.LeftParen]

       rpar : typing.Sequence[expression.RightParen]
          Sequence of parenthesis for precedence dictation.

       whitespace_after_await : whitespace.BaseParenthesizableWhitespace
          Whitespace that appears after the `async` keyword, but before the inner `expression`.

class `libcst`.**Yield**
    A yield expression similar to `yield x` or `yield from fun()`.

    To learn more about the ways that yield can be used in generators, refer to Python's language reference.

       value : typing.Union[expression.BaseExpression, expression.From, NoneType]
          The value yielded from the generator, in the case of a *From* clause, a sub-generator to iterate over.

       lpar : typing.Sequence[expression.LeftParen]

       rpar : typing.Sequence[expression.RightParen]
          Sequence of parenthesis for precedence dictation.

       whitespace_after_yield : typing.Union[whitespace.BaseParenthesizableWhitespace, MaybeS
          Whitespace after the `yield` keyword, but before the `value`.

class `libcst`.**From**
    A `from x` stanza in a *Yield* or *Raise*.

       item : expression.BaseExpression
          The expression that we are yielding/raising from.

       whitespace_before_from : typing.Union[whitespace.BaseParenthesizableWhitespace, MaybeS
          The whitespace at the very start of this node.

       whitespace_after_from : whitespace.BaseParenthesizableWhitespace
          The whitespace after the `from` keyword, but before the `item`.

class `libcst`.**IfExp**
    An if expression of the form `body if test else orelse`.

    If statements are provided by *If* and *Else* nodes.

       test : expression.BaseExpression
          The test to perform.

       body : expression.BaseExpression
          The expression to evaluate when the test is true.

       orelse : expression.BaseExpression
          The expression to evaluate when the test is false.

       lpar : typing.Sequence[expression.LeftParen]

       rpar : typing.Sequence[expression.RightParen]
          Sequence of parenthesis for precedence dictation.

       whitespace_before_if : whitespace.BaseParenthesizableWhitespace
          Whitespace after the `body` expression, but before the `if` keyword.

       whitespace_after_if : whitespace.BaseParenthesizableWhitespace
          Whitespace after the `if` keyword, but before the `test` clause.

       whitespace_before_else : whitespace.BaseParenthesizableWhitespace
          Whitespace after the `test` expression, but before the `else` keyword.

> **whitespace_after_else** : **whitespace.BaseParenthesizableWhitespace**
> Whitespace after the else keyword, but before the orelse expression.

10.3.4 Lambdas and Function Calls

class libcst.**Lambda**
> A lambda expression that creates an anonymous function.

```
Lambda(
    params=Parameters([Param(Name("arg"))]),
    body=Ellipsis(),
)
```

> Represents the following code:

```
lambda arg: ...
```

> Named functions statements are provided by *FunctionDef*.
>
> **params** : **expression.Parameters**
> The arguments to the lambda. This is similar to the arguments on a *FunctionDef*, however lambda arguments are not allowed to have an *Annotation*.
>
> **body** : **expression.BaseExpression**
> The value that the lambda computes and returns when called.
>
> **colon** : **op.Colon**
> The colon separating the parameters from the body.
>
> **lpar** : **typing.Sequence[expression.LeftParen]**
>
> **rpar** : **typing.Sequence[expression.RightParen]**
> Sequence of parenthesis for precedence dictation.
>
> **whitespace_after_lambda** : **typing.Union[whitespace.BaseParenthesizableWhitespace, Maybe**
> Whitespace after the lambda keyword, but before any argument or the colon.

class libcst.**Call**
> An expression representing a function call, such as do_math(1, 2) or picture. post_on_instagram().
>
> Function calls consist of a function name and a sequence of arguments wrapped in *Arg* nodes.
>
> **func** : **expression.BaseExpression**
> The expression resulting in a callable that we are to call. Often a *Name* or *Attribute*.
>
> **args** : **typing.Sequence[expression.Arg]**
> The arguments to pass to the resulting callable. These may be a mix of positional arguments, keyword arguments, or "starred" arguments.
>
> **lpar** : **typing.Sequence[expression.LeftParen]**
>
> **rpar** : **typing.Sequence[expression.RightParen]**
> Sequence of parenthesis for precedence dictation. These are not the parenthesis before and after the list of args, but rather arguments around the entire call expression, such as ((do_math(1, 2))).
>
> **whitespace_after_func** : **whitespace.BaseParenthesizableWhitespace**
> Whitespace after the func name, but before the opening parenthesis.

whitespace_before_args : **whitespace.BaseParenthesizableWhitespace**
> Whitespace after the opening parenthesis but before the first argument (if there are any). Whitespace after the last argument but before the closing parenthesis is owned by the last *Arg* if it exists.

class libcst.**Arg**
> A single argument to a *Call*.

> This supports named keyword arguments in the form of keyword=value and variable argument expansion using *args or **kwargs syntax.

> **value** : **expression.BaseExpression**
>> The argument expression itself, not including a preceding keyword, or any of the surrounding the value, like a comma or asterisks.

> **keyword** : **typing.Optional[expression.Name]**
>> Optional keyword for the argument.

> **equal** : **typing.Union[op.AssignEqual, MaybeSentinel] = MaybeSentinel.DEFAULT**
>> The equal sign used to denote assignment if there is a keyword.

> **comma** : **typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT**
>> Any trailing comma.

> **star** : **typing.Literal['', '*', '**']**
>> A string with zero, one, or two asterisks appearing before the name. These are expanded into variable number of positional or keyword arguments.

> **whitespace_after_star** : **whitespace.BaseParenthesizableWhitespace**
>> Whitespace after the star (if it exists), but before the keyword or value (if no keyword is provided).

> **whitespace_after_arg** : **whitespace.BaseParenthesizableWhitespace**
>> Whitespace after this entire node. The *Comma* node (if it exists) may also store some trailing whitespace.

10.3.5 Literal Values

class libcst.**Ellipsis**
> An ellipsis .... When used as an expression, it evaluates to the Ellipsis constant. Ellipsis are often used as placeholders in code or in conjunction with *SubscriptElement*.

> **lpar** : **typing.Sequence[expression.LeftParen]**

> **rpar** : **typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

Numbers

class libcst.**BaseNumber**
> A type such as *Integer*, *Float*, or *Imaginary* that can be used anywhere that you need to explicitly take any number type.

class libcst.**Integer**

> **value** : **str**
>> A string representation of the integer, such as "100000" or 100_000.

>> To convert this string representation to an int, use the calculated property *evaluated_value*.

> **lpar** : **typing.Sequence[expression.LeftParen]**

> **rpar** : **typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

> **property evaluated_value**
>> Return an `ast.literal_eval()` evaluated int of *value*.

class libcst.**Float**

> **value** : **str**
>> A string representation of the floating point number, such as `"0.05"`, `".050"`, or `"5e-2"`.

>> To convert this string representation to an `float`, use the calculated property *evaluated_value*.

> **lpar** : **typing.Sequence[expression.LeftParen]**

> **rpar** : **typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

> **property evaluated_value**
>> Return an `ast.literal_eval()` evaluated float of *value*.

class libcst.**Imaginary**

> **value** : **str**
>> A string representation of the imaginary (complex) number, such as `"2j"`.

>> To convert this string representation to an `complex`, use the calculated property *evaluated_value*.

> **lpar** : **typing.Sequence[expression.LeftParen]**

> **rpar** : **typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precedence dictation.

> **property evaluated_value**
>> Return an `ast.literal_eval()` evaluated complex of *value*.

Strings

class libcst.**BaseString**
> A type that can be used anywhere that you need to take any string. This includes *SimpleString*, *ConcatenatedString*, and *FormattedString*.

class libcst.**SimpleString**
> Any sort of literal string expression that is not a *FormattedString* (f-string), including triple-quoted multi-line strings.

> **value** : **str**
>> The texual representation of the string, including quotes, prefix characters, and any escape characters present in the original source code , such as `r"my string\n"`. To remove the quotes and interpret any escape characters, use the calculated property *evaluated_value*.

> **lpar** : **typing.Sequence[expression.LeftParen]**

> **rpar** : **typing.Sequence[expression.RightParen]**
>> Sequence of parenthesis for precidence dictation.

> **property prefix**
>> Returns the string's prefix, if any exists. The prefix can be `r`, `u`, `b`, `br` or `rb`.

> **property quote**
>> Returns the quotation used to denote the string. Can be either `'`, `"`, `'''` or `"""`.

property raw_value

Returns the raw value of the string as it appears in source, without the beginning or end quotes and without the prefix. This is often useful when constructing transforms which need to manipulate strings in source code.

property evaluated_value

Return an `ast.literal_eval()` evaluated str of *value*.

class libcst.**ConcatenatedString**

Represents an implicitly concatenated string, such as:

```
"abc" "def" == "abcdef"
```

> **Warning:** This is different from two strings joined in a *BinaryOperation* with an *Add* operator, and is sometimes viewed as an antifeature of Python.

left : typing.Union[expression.SimpleString, expression.FormattedString]

String on the left of the concatenation.

right : typing.Union[expression.SimpleString, expression.FormattedString, ForwardRef('C

String on the right of the concatenation.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]

Sequence of parenthesis for precedence dictation.

whitespace_between : whitespace.BaseParenthesizableWhitespace

Whitespace between the `left` and `right` substrings.

property evaluated_value

Return an `ast.literal_eval()` evaluated str of recursively concatenated *left* and *right* if and only if both *left* and *right* are composed by *SimpleString* or *ConcatenatedString* (*FormattedString* cannot be evaluated).

Formatted Strings (f-strings)

class libcst.**FormattedString**

An "f-string". These formatted strings are string literals prefixed by the letter "f". An f-string may contain interpolated expressions inside curly braces (`{` and `}`).

F-strings are defined in PEP 498 and documented in Python's language reference.

```
>>> import libcst as cst
>>> cst.parse_expression('f"ab{cd}ef"')
FormattedString(
    parts=[
        FormattedStringText(
            value='ab',
        ),
        FormattedStringExpression(
            expression=Name(
                value='cd',
                lpar=[],
                rpar=[],
            ),
```

(continues on next page)

```
            conversion=None,
            format_spec=None,
            whitespace_before_expression=SimpleWhitespace(
                value='',
            ),
            whitespace_after_expression=SimpleWhitespace(
                value='',
            ),
        ),
        FormattedStringText(
            value='ef',
        ),
    ],
    start='f"',
    end='"',
    lpar=[],
    rpar=[],
)
```

parts : typing.Sequence[expression.BaseFormattedStringContent]
A formatted string is composed as a series of *FormattedStringText* and *FormattedStringExpression* parts.

start : str = "f""
The string prefix and the leading quote, such as f", F', fr", or f""".

end : typing.Literal['"', "'", '"""', "'''"] = """
The trailing quote. This must match the type of quote used in start.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
Sequence of parenthesis for precidence dictation.

property prefix
Returns the string's prefix, if any exists. The prefix can be f, fr, or rf.

property quote
Returns the quotation used to denote the string. Can be either ', ", ''' or """.

class libcst.**BaseFormattedStringContent**
The base type for *FormattedStringText* and *FormattedStringExpression*. A *FormattedString* is composed of a sequence of *BaseFormattedStringContent* parts.

class libcst.**FormattedStringText**
Part of a *FormattedString* that is not inside curly braces ({ or }). For example, in:

```
f"ab{cd}ef"
```

ab and ef are *FormattedStringText* nodes, but {cd} is a *FormattedStringExpression*.

value : str
The raw string value, including any escape characters present in the source code, not including any enclosing quotes.

class libcst.**FormattedStringExpression**
Part of a *FormattedString* that is inside curly braces ({ or }), including the surrounding curly braces. For example, in:

```
f"ab{cd}ef"
```

`{cd}` is a *FormattedStringExpression*, but `ab` and `ef` are *FormattedStringText* nodes.

An f-string expression may contain `conversion` and `format_spec` suffixes that control how the expression is converted to a string. See Python's language reference for details.

expression : expression.BaseExpression
    The expression we will evaluate and render when generating the string.

conversion : typing.Optional[str]
    An optional conversion specifier, such as `!s`, `!r` or `!a`.

format_spec : typing.Optional[typing.Sequence[expression.BaseFormattedStringContent]]
    An optional format specifier following the format specification mini-language.

whitespace_before_expression : whitespace.BaseParenthesizableWhitespace
    Whitespace after the opening curly brace (`{`), but before the `expression`.

whitespace_after_expression : whitespace.BaseParenthesizableWhitespace
    Whitespace after the `expression`, but before the `conversion`, `format_spec` and the closing curly brace (`}`). Python does not allow whitespace inside or after a `conversion` or `format_spec`.

equal : typing.Optional[op.AssignEqual]
    Equal sign for formatted string expression uses self-documenting expressions, such as `f"{x=}"`. See the Python 3.8 release notes.

10.3.6 Collections

Simple Collections

class `libcst.`**Tuple**
    An immutable literal tuple. Tuples are often (but not always) parenthesized.

```
Tuple([
    Element(Integer("1")),
    Element(Integer("2")),
    StarredElement(Name("others")),
])
```

generates the following code:

```
(1, 2, *others)
```

elements : typing.Sequence[expression.BaseElement]
    A sequence containing all the *Element* and *StarredElement* nodes in the tuple.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
    Sequence of parenthesis for precedence dictation.

class `libcst.`**BaseList**
    A base class for *List* and *ListComp*, which both result in a list object when evaluated.

    lbracket = Field(name=None,type=None,default=<dataclasses._MISSING_TYPE object>,default

    rbracket = Field(name=None,type=None,default=<dataclasses._MISSING_TYPE object>,default
        Brackets surrounding the list.

lpar = ()

rpar = ()
> Sequence of parenthesis for precedence dictation.

class libcst.**List**
> A mutable literal list.

```
List([
    Element(Integer("1")),
    Element(Integer("2")),
    StarredElement(Name("others")),
])
```

generates the following code:

```
[1, 2, *others]
```

List comprehensions are represented with a *ListComp* node.

elements : typing.Sequence[expression.BaseElement]
> A sequence containing all the *Element* and *StarredElement* nodes in the list.

lbracket : expression.LeftSquareBracket

rbracket : expression.RightSquareBracket
> Brackets surrounding the list.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
> Sequence of parenthesis for precedence dictation.

class libcst.**BaseSet**
> An abstract base class for *Set* and *SetComp*, which both result in a set object when evaluated.

class libcst.**Set**
> A mutable literal set.

```
Set([
    Element(Integer("1")),
    Element(Integer("2")),
    StarredElement(Name("others")),
])
```

generates the following code:

```
{1, 2, *others}
```

Set comprehensions are represented with a *SetComp* node.

elements : typing.Sequence[expression.BaseElement]
> A sequence containing all the *Element* and *StarredElement* nodes in the set.

lbrace : expression.LeftCurlyBrace

rbrace : expression.RightCurlyBrace
> Braces surrounding the set.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
> Sequence of parenthesis for precedence dictation.

Simple Collection Elements

class libcst.**BaseElement**
> An element of a literal list, tuple, or set. For elements of a literal dict, see BaseDictElement.

class libcst.**Element**
> A simple value in a literal *List*, *Tuple*, or *Set*. These a literal collection may also contain a *StarredElement*.
>
> If you're using a literal *Dict*, see *DictElement* instead.
>
> **value : expression.BaseExpression**
>
> **comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT**
> > A trailing comma. By default, we'll only insert a comma if one is required.

class libcst.**StarredElement**
> A starred *value element that expands to represent multiple values in a literal *List*, *Tuple*, or *Set*.
>
> If you're using a literal *Dict*, see *StarredDictElement* instead.
>
> If this node owns parenthesis, those parenthesis wrap the leading asterisk, but not the trailing comma. For example:

```
StarredElement(
    cst.Name("el"),
    comma=cst.Comma(),
    lpar=[cst.LeftParen()],
    rpar=[cst.RightParen()],
)
```

> will generate:

```
(*el),
```

> **value : expression.BaseExpression**
>
> **comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT**
> > A trailing comma. By default, we'll only insert a comma if one is required.
>
> **lpar : typing.Sequence[expression.LeftParen]**
> > Parenthesis at the beginning of the node, before the leading asterisk.
>
> **rpar : typing.Sequence[expression.RightParen]**
> > Parentheses after the value, but before a comma (if there is one).
>
> **whitespace_before_value : whitespace.BaseParenthesizableWhitespace**
> > Whitespace between the leading asterisk and the value expression.

Dictionaries

class libcst.**BaseDict**
> An abstract base class for *Dict* and *DictComp*, which both result in a dict object when evaluated.

class libcst.**Dict**
> A literal dictionary. Key-value pairs are stored in elements using *DictElement* nodes.
>
> It's possible to expand one dictionary into another, as in {k: v, **expanded}. Expanded elements are stored as *StarredDictElement* nodes.

```
Dict([
    DictElement(Name("k1"), Name("v1")),
    DictElement(Name("k2"), Name("v2")),
    StarredDictElement(Name("expanded")),
])
```

generates the following code:

```
{k1: v1, k2: v2, **expanded}
```

elements : typing.Sequence[expression.BaseDictElement]

lbrace : expression.LeftCurlyBrace

rbrace : expression.RightCurlyBrace

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]

Dictionary Elements

class libcst.**BaseDictElement**
    An element of a literal dict. For elements of a list, tuple, or set, see BaseElement.

class libcst.**DictElement**
    A simple key: value pair that represents a single entry in a literal *Dict*. *Dict* nodes may also contain a *StarredDictElement*.

    If you're using a literal *List*, *Tuple*, or *Set*, see *Element* instead.

    key : expression.BaseExpression

    value : expression.BaseExpression

    comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
        A trailing comma. By default, we'll only insert a comma if one is required.

    whitespace_before_colon : whitespace.BaseParenthesizableWhitespace
        Whitespace after the key, but before the colon in key : value.

    whitespace_after_colon : whitespace.BaseParenthesizableWhitespace
        Whitespace after the colon, but before the value in key : value.

class libcst.**StarredDictElement**
    A starred **value element that expands to represent multiple values in a literal *Dict*.

    If you're using a literal *List*, *Tuple*, or *Set*, see *StarredElement* instead.

    Unlike *StarredElement*, this node does not own left or right parenthesis, but the value field may still contain parenthesis. This is due to some asymmetry in Python's grammar.

    value : expression.BaseExpression

    comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
        A trailing comma. By default, we'll only insert a comma if one is required.

    whitespace_before_value : whitespace.BaseParenthesizableWhitespace
        Whitespace between the leading asterisks and the value expression.

10.3.7 Comprehensions

class libcst.**BaseComp**

A base class for all comprehension and generator expressions, including *GeneratorExp*, *ListComp*, *SetComp*, and *DictComp*.

class libcst.**BaseSimpleComp**

The base class for *ListComp*, *SetComp*, and *GeneratorExp*. *DictComp* is not a *BaseSimpleComp*, because it uses key and value.

elt

The expression evaluated during each iteration of the comprehension. This lexically comes before the for_in clause, but it is semantically the inner-most element, evaluated inside the for_in clause.

for_in

The for ... in ... if ... clause that lexically comes after elt. This may be a nested structure for nested comprehensions. See *CompFor* for details.

class libcst.**GeneratorExp**

A generator expression. elt represents the value yielded for each item in *CompFor.iter*.

All for ... in ... and if ... clauses are stored as a recursive *CompFor* data structure inside for_in.

elt : expression.BaseAssignTargetExpression

The expression evaluated and yielded during each iteration of the generator.

for_in : expression.CompFor

The for ... in ... if ... clause that comes after elt. This may be a nested structure for nested comprehensions. See *CompFor* for details.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]

Sequence of parentheses for precedence dictation. Generator expressions must always be parenthesized. However, if a generator expression is the only argument inside a function call, the enclosing *Call* node may own the parentheses instead.

class libcst.**ListComp**

A list comprehension. elt represents the value stored for each item in *CompFor.iter*.

All for ... in ... and if ... clauses are stored as a recursive *CompFor* data structure inside for_in.

elt : expression.BaseAssignTargetExpression

The expression evaluated and stored during each iteration of the comprehension.

for_in : expression.CompFor

The for ... in ... if ... clause that comes after elt. This may be a nested structure for nested comprehensions. See *CompFor* for details.

lbracket : expression.LeftSquareBracket

rbracket : expression.RightSquareBracket

Brackets surrounding the list comprehension.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]

Sequence of parenthesis for precedence dictation.

class libcst.**SetComp**

A set comprehension. elt represents the value stored for each item in *CompFor.iter*.

All `for ... in ...` and `if ...` clauses are stored as a recursive *CompFor* data structure inside `for_in`.

elt : expression.BaseAssignTargetExpression
The expression evaluated and stored during each iteration of the comprehension.

for_in : expression.CompFor
The `for ... in ... if ...` clause that comes after `elt`. This may be a nested structure for nested comprehensions. See *CompFor* for details.

lbrace : expression.LeftCurlyBrace

rbrace : expression.RightCurlyBrace
Braces surrounding the set comprehension.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
Sequence of parenthesis for precedence dictation.

class `libcst.`**DictComp**
A dictionary comprehension. `key` and `value` represent the dictionary entry evaluated for each item.

All `for ... in ...` and `if ...` clauses are stored as a recursive *CompFor* data structure inside `for_in`.

key : expression.BaseAssignTargetExpression
The key inserted into the dictionary during each iteration of the comprehension.

value : expression.BaseAssignTargetExpression
The value associated with the `key` inserted into the dictionary during each iteration of the comprehension.

for_in : expression.CompFor
The `for ... in ... if ...` clause that lexically comes after `key` and `value`. This may be a nested structure for nested comprehensions. See *CompFor* for details.

lbrace : expression.LeftCurlyBrace

rbrace : expression.RightCurlyBrace
Braces surrounding the dict comprehension.

lpar : typing.Sequence[expression.LeftParen]

rpar : typing.Sequence[expression.RightParen]
Sequence of parenthesis for precedence dictation.

whitespace_before_colon : whitespace.BaseParenthesizableWhitespace
Whitespace after the key, but before the colon in `key : value`.

whitespace_after_colon : whitespace.BaseParenthesizableWhitespace
Whitespace after the colon, but before the value in `key : value`.

class `libcst.`**CompFor**
One `for` clause in a *BaseComp*, or a nested hierarchy of `for` clauses.

Nested loops in comprehensions are difficult to get right, but they can be thought of as a flat representation of nested clauses.

`elt for a in b for c in d if e` can be thought of as:

```python
for a in b:
    for c in d:
        if e:
            yield elt
```

And that would form the following CST:

```
ListComp(
    elt=Name("elt"),
    for_in=CompFor(
        target=Name("a"),
        iter=Name("b"),
        ifs=[],
        inner_comp_for=CompFor(
            target=Name("c"),
            iter=Name("d"),
            ifs=[
                CompIf(
                    test=Name("e"),
                ),
            ],
        ),
    ),
)
```

Normal `for` statements are provided by *For*.

target : expression.BaseAssignTargetExpression
> The target to assign a value to in each iteration of the loop. This is different from *GeneratorExp.elt*, *ListComp.elt*, *SetComp.elt*, and `key` and `value` in *DictComp*, because it doesn't directly effect the value of resulting generator, list, set, or dict.

iter : expression.BaseExpression
> The value to iterate over. Every value in `iter` is stored in `target`.

ifs : typing.Sequence[ForwardRef('CompIf')]
> Zero or more conditional clauses that control this loop. If any of these tests fail, the `target` item is skipped.

```
if a if b if c
```

> has similar semantics to:

```
if a and b and c
```

inner_for_in : typing.Optional[ForwardRef('CompFor')]
> Another *CompFor* node used to form nested loops. Nested comprehensions can be useful, but they tend to be difficult to read and write. As a result they are uncommon.

asynchronous : typing.Optional[expression.Asynchronous]
> An optional async modifier that appears before the `for` keyword.

whitespace_before : whitespace.BaseParenthesizableWhitespace
> Whitespace that appears at the beginning of this node, before the `for` and `async` keywords.

whitespace_after_for : whitespace.BaseParenthesizableWhitespace
> Whitespace appearing after the `for` keyword, but before the `target`.

whitespace_before_in : whitespace.BaseParenthesizableWhitespace
> Whitespace appearing after the `target`, but before the `in` keyword.

whitespace_after_in : whitespace.BaseParenthesizableWhitespace
> Whitespace appearing after the `in` keyword, but before the `iter`.

class libcst.**CompIf**
> A conditional clause in a *CompFor*, used as part of a generator or comprehension expression.

If the `test` fails, the current element in the [*CompFor*](#) will be skipped.

test : expression.BaseExpression
> An expression to evaluate. When interpreted, Python will coerce it to a boolean.

whitespace_before : whitespace.BaseParenthesizableWhitespace
> Whitespace before the `if` keyword.

whitespace_before_test : whitespace.BaseParenthesizableWhitespace
> Whitespace after the `if` keyword, but before the `test` expression.

10.3.8 Subscripts and Slices

class `libcst.`**Subscript**
> A indexed subscript reference ([*Index*](#)) such as `x[2]`, a [*Slice*](#) such as `x[1:-1]`, or an extended slice ([*SubscriptElement*](#)) such as `x[1:2, 3]`.

> **value : expression.BaseExpression**
> > The left-hand expression which, when evaluated, will be subscripted, such as `x` in `x[2]`.

> **slice : typing.Sequence[expression.SubscriptElement]**
> > The [*SubscriptElement*](#) to extract from the `value`.

> **lbracket : expression.LeftSquareBracket**

> **rbracket : expression.RightSquareBracket**
> > Brackets after the `value` surrounding the `slice`.

> **lpar : typing.Sequence[expression.LeftParen]**

> **rpar : typing.Sequence[expression.RightParen]**
> > Sequence of parenthesis for precedence dictation.

> **whitespace_after_value : whitespace.BaseParenthesizableWhitespace**
> > Whitespace after the `value`, but before the `lbracket`.

class `libcst.`**BaseSlice**
> Any slice type that can slot into a [*SubscriptElement*](#). This node is purely for typing.

class `libcst.`**Index**
> Any index as passed to a [*Subscript*](#). In `x[2]`, this would be the `2` value.

> **value : expression.BaseExpression**
> > The index value itself.

class `libcst.`**Slice**
> Any slice operation in a [*Subscript*](#), such as `1:`, `2:3:4`, etc.

> Note that the grammar does NOT allow parenthesis around a slice so they are not supported here.

> **lower : typing.Optional[expression.BaseExpression]**
> > The lower bound in the slice, if present

> **upper : typing.Optional[expression.BaseExpression]**
> > The upper bound in the slice, if present

> **step : typing.Optional[expression.BaseExpression]**
> > The step in the slice, if present

> **first_colon : op.Colon**
> > The first slice operator

second_colon : typing.Union[op.Colon, MaybeSentinel] = MaybeSentinel.DEFAULT
> The second slice operator, usually omitted

class libcst.**SubscriptElement**
> Part of a sequence of slices in a *Subscript*, such as 1:2, 3. This is not used in Python's standard library, but it is used in some third-party libraries. For example, NumPy uses it to select values and ranges from multi-dimensional arrays.
>
> **slice : expression.BaseSlice**
> > A slice or index that is part of a subscript.
>
> **comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT**
> > A separating comma, with any whitespace it owns.

10.3.9 Parenthesis, Brackets, and Braces

class libcst.**LeftParen**
> Used by various nodes to denote a parenthesized section. This doesn't own the whitespace to the left of it since this is owned by the parent node.
>
> **whitespace_after : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly after this left parenthesis.

class libcst.**RightParen**
> Used by various nodes to denote a parenthesized section. This doesn't own the whitespace to the right of it since this is owned by the parent node.
>
> **whitespace_before : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly after this left parenthesis.

class libcst.**LeftSquareBracket**
> Used by various nodes to denote a subscript or list section. This doesn't own the whitespace to the left of it since this is owned by the parent node.
>
> **whitespace_after : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly after this left square bracket.

class libcst.**RightSquareBracket**
> Used by various nodes to denote a subscript or list section. This doesn't own the whitespace to the right of it since this is owned by the parent node.
>
> **whitespace_before : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly before this right square bracket.

class libcst.**LeftCurlyBrace**
> Used by various nodes to denote a dict or set. This doesn't own the whitespace to the left of it since this is owned by the parent node.
>
> **whitespace_after : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly after this left curly brace.

class libcst.**RightCurlyBrace**
> Used by various nodes to denote a dict or set. This doesn't own the whitespace to the right of it since this is owned by the parent node.
>
> **whitespace_before : whitespace.BaseParenthesizableWhitespace**
> > Any space that appears directly before this right curly brace.

10.4 Statements

Statements represent a "line of code" or a control structure with other lines of code, such as an *If* block.

All statements subclass from *BaseSmallStatement* or *BaseCompoundStatement*.

Statements can be parsed with *parse_statement()* or as part of a module using *parse_module()*.

10.4.1 Simple Statements

Statements which at most have expressions as child attributes.

class libcst.**BaseSmallStatement**

Encapsulates a small statement, like del or pass, and optionally adds a trailing semicolon. A small statement is always contained inside a *SimpleStatementLine* or *SimpleStatementSuite*. This exists to simplify type definitions and isinstance checks.

> **semicolon = 1**
>
> An optional semicolon that appears after a small statement. This is optional for the last small statement in a *SimpleStatementLine* or *SimpleStatementSuite*, but all other small statements inside a simple statement must contain a semicolon to disambiguate multiple small statements on the same line.

class libcst.**AnnAssign**

An assignment statement such as x: int = 5 or x: int. This only allows for one assignment target unlike *Assign* but it includes a variable annotation. Also unlike *Assign*, the assignment target is optional, as it is possible to annotate an expression without assigning to it.

> **target : expression.BaseAssignTargetExpression**
>
> The target that is being annotated and possibly assigned to.
>
> **annotation : expression.Annotation**
>
> The annotation for the target.
>
> **value : typing.Optional[expression.BaseExpression]**
>
> The optional expression being assigned to the target.
>
> **equal : typing.Union[op.AssignEqual, MaybeSentinel] = MaybeSentinel.DEFAULT**
>
> The equals sign used to denote assignment if there is a value.
>
> **semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT**
>
> Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Assert**

An assert statement such as assert x > 5 or assert x > 5, 'Uh oh!'

> **test : expression.BaseExpression**
>
> The test we are going to assert on.
>
> **msg : typing.Optional[expression.BaseExpression]**
>
> The optional message to display if the test evaluates to a falsey value.
>
> **comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT**
>
> A comma separating test and message, if there is a message.
>
> **whitespace_after_assert : whitespace.SimpleWhitespace**
>
> Whitespace appearing after the assert keyword and before the test.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Assign**
An assignment statement such as x = y or x = y = z. Unlike *AnnAssign*, this does not allow type annotations but does allow for multiple targets.

targets : typing.Sequence[statement.AssignTarget]
One or more targets that are being assigned to.

value : expression.BaseExpression
The expression being assigned to the targets.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**AugAssign**
An augmented assignment statement, such as x += 5.

target : expression.BaseAssignTargetExpression
Target that is being operated on and assigned to.

operator : op.BaseAugOp
The augmented assignment operation being performed.

value : expression.BaseExpression
The value used with the above operator to calculate the new assignment.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Break**
Represents a break statement, which is used to break out of a *For* or *While* loop early.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Continue**
Represents a continue statement, which is used to skip to the next iteration in a *For* or *While* loop.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Del**
Represents a del statement. del is always followed by a target.

target : expression.BaseDelTargetExpression
The target expression will be deleted. This can be a name, a tuple, an item of a list, an item of a dictionary, or an attribute.

whitespace_after_del : whitespace.SimpleWhitespace
The whitespace after the del keyword.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Expr**

    An expression used as a statement, where the result is unused and unassigned. The most common place you will find this is in function calls where the return value is unneeded.

    value : expression.BaseExpression

        The expression itself. Python will evaluate the expression but not assign the result anywhere.

    semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT

        Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Global**

    A global statement.

    names : typing.Sequence[statement.NameItem]

        A list of one or more names.

    whitespace_after_global : whitespace.SimpleWhitespace

        Whitespace appearing after the global keyword and before the first name.

    semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT

        Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Import**

    An import statement.

    names : typing.Sequence[statement.ImportAlias]

        One or more names that are being imported, with optional local aliases.

    semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT

        Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

    whitespace_after_import : whitespace.SimpleWhitespace

        The whitespace that appears after the import keyword but before the first import alias.

class libcst.**ImportFrom**

    A from x import y statement.

    module : typing.Union[expression.Attribute, expression.Name, NoneType]

        Name or Attribute node representing the module we're importing from. This is optional as *ImportFrom* allows purely relative imports.

    names : typing.Union[typing.Sequence[statement.ImportAlias], op.ImportStar]

        One or more names that are being imported from the specified module, with optional local aliases.

    relative : typing.Sequence[op.Dot]

        Sequence of *Dot* nodes indicating relative import level.

    lpar : typing.Optional[expression.LeftParen]

        Optional open parenthesis for multi-line import continuation.

    rpar : typing.Optional[expression.RightParen]

        Optional close parenthesis for multi-line import continuation.

    semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT

        Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

    whitespace_after_from : whitespace.SimpleWhitespace

        The whitespace that appears after the from keyword but before the module and any relative import dots.

whitespace_before_import : whitespace.SimpleWhitespace
> The whitespace that appears after the module but before the import keyword.

whitespace_after_import : whitespace.SimpleWhitespace
> The whitespace that appears after the import keyword but before the first import name or optional left paren.

class libcst.**Nonlocal**
> A nonlocal statement.

names : typing.Sequence[statement.NameItem]
> A list of one or more names.

whitespace_after_nonlocal : whitespace.SimpleWhitespace
> Whitespace appearing after the global keyword and before the first name.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
> Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Pass**
> Represents a pass statement.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
> Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Raise**
> A raise exc or raise exc from cause statement.

exc : typing.Optional[expression.BaseExpression]
> The exception that we should raise.

cause : typing.Optional[expression.From]
> Optionally, a from cause clause to allow us to raise an exception out of another exception's context.

whitespace_after_raise : typing.Union[whitespace.SimpleWhitespace, MaybeSentinel] = Ma
> Any whitespace appearing between the raise keyword and the exception.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
> Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

class libcst.**Return**
> Represents a return or a return x statement.

value : typing.Optional[expression.BaseExpression]
> The optional expression that will be evaluated and returned.

whitespace_after_return : typing.Union[whitespace.SimpleWhitespace, MaybeSentinel] = Ma
> Optional whitespace after the return keyword before the optional value expression.

semicolon : typing.Union[op.Semicolon, MaybeSentinel] = MaybeSentinel.DEFAULT
> Optional semicolon when this is used in a statement line. This semicolon owns the whitespace on both sides of it when it is used.

10.4.2 Compound Statements

Statements that have one or more statement blocks as a child attribute.

class libcst.**BaseCompoundStatement**

Encapsulates a compound statement, like if True: pass or while True: pass. This exists to simplify type definitions and isinstance checks.

> Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.
>
> —https://docs.python.org/3/reference/compound_stmts.html

body

The body of this compound statement.

leading_lines

Any empty lines or comments appearing before this statement.

class libcst.**ClassDef**

A class definition.

name : expression.Name

The class name.

body : statement.BaseSuite

The class body.

bases : typing.Sequence[expression.Arg]

Sequence of base classes this class inherits from.

keywords : typing.Sequence[expression.Arg]

Sequence of keywords, such as "metaclass".

decorators : typing.Sequence[statement.Decorator]

Sequence of decorators applied to this class.

lpar : typing.Union[expression.LeftParen, MaybeSentinel] = MaybeSentinel.DEFAULT

Optional open parenthesis used when there are bases or keywords.

rpar : typing.Union[expression.RightParen, MaybeSentinel] = MaybeSentinel.DEFAULT

Optional close parenthesis used when there are bases or keywords.

leading_lines : typing.Sequence[whitespace.EmptyLine]

Leading empty lines and comments before the first decorator. We assume any comments before the first decorator are owned by the class definition itself. If there are no decorators, this will still contain all of the empty lines and comments before the class definition.

lines_after_decorators : typing.Sequence[whitespace.EmptyLine]

Empty lines and comments between the final decorator and the *ClassDef* node. In the case of no decorators, this will be empty.

whitespace_after_class : whitespace.SimpleWhitespace

Whitespace after the class keyword and before the class name.

whitespace_after_name : whitespace.SimpleWhitespace

Whitespace after the class name and before the opening parenthesis for the bases and keywords.

whitespace_before_colon : whitespace.SimpleWhitespace

Whitespace after the closing parenthesis or class name and before the colon.

get_docstring()

Returns a inspect.cleandoc() cleaned docstring if the docstring is available, None otherwise.

class libcst.**For**
A for target in iter statement.

> **target : expression.BaseAssignTargetExpression**
>   The target of the iterator in the for statement.
>
> **iter : expression.BaseExpression**
>   The iterable expression we will loop over.
>
> **body : statement.BaseSuite**
>   The suite that is wrapped with this statement.
>
> **orelse : typing.Optional[statement.Else]**
>   An optional else case which will be executed if there is no *Break* statement encountered while looping.
>
> **asynchronous : typing.Optional[expression.Asynchronous]**
>   Optional async modifier, if this is an *async for* statement.
>
> **leading_lines : typing.Sequence[whitespace.EmptyLine]**
>   Sequence of empty lines appearing before this for statement.
>
> **whitespace_after_for : whitespace.SimpleWhitespace**
>   Whitespace after the for keyword and before the target.
>
> **whitespace_before_in : whitespace.SimpleWhitespace**
>   Whitespace after the target and before the in keyword.
>
> **whitespace_after_in : whitespace.SimpleWhitespace**
>   Whitespace after the in keyword and before the iter.
>
> **whitespace_before_colon : whitespace.SimpleWhitespace**
>   Whitespace after the iter and before the colon.

class libcst.**FunctionDef**
A function definition.

> **name : expression.Name**
>   The function name.
>
> **params : expression.Parameters**
>   The function parameters. Present even if there are no params.
>
> **body : statement.BaseSuite**
>   The function body.
>
> **decorators : typing.Sequence[statement.Decorator]**
>   Sequence of decorators applied to this function. Decorators are listed in order that they appear in source (top to bottom) as apposed to the order that they are applied to the function at runtime.
>
> **returns : typing.Optional[expression.Annotation]**
>   An optional return annotation, if the function is annotated.
>
> **asynchronous : typing.Optional[expression.Asynchronous]**
>   Optional async modifier, if this is an async function.
>
> **leading_lines : typing.Sequence[whitespace.EmptyLine]**
>   Leading empty lines and comments before the first decorator. We assume any comments before the first decorator are owned by the function definition itself. If there are no decorators, this will still contain all of the empty lines and comments before the function definition.
>
> **lines_after_decorators : typing.Sequence[whitespace.EmptyLine]**
>   Empty lines and comments between the final decorator and the *FunctionDef* node. In the case of no decorators, this will be empty.

> **whitespace_after_def** : **whitespace.SimpleWhitespace**
> Whitespace after the `def` keyword and before the function name.

> **whitespace_after_name** : **whitespace.SimpleWhitespace**
> Whitespace after the function name and before the opening parenthesis for the parameters.

> **whitespace_before_params** : **whitespace.BaseParenthesizableWhitespace**
> Whitespace after the opening parenthesis for the parameters but before the first param itself.

> **whitespace_before_colon** : **whitespace.SimpleWhitespace**
> Whitespace after the closing parenthesis or return annotation and before the colon.

> **get_docstring**()
> When docstring is available, returns a [`inspect.cleandoc()`](#) cleaned docstring. Otherwise, returns `None`.

class `libcst.`**If**

> An `if` statement. `test` holds a single test expression.

> `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.

> **test** : **expression.BaseExpression**
> The expression that, when evaluated, should give us a truthy/falsey value.

> **body** : **statement.BaseSuite**
> The body of this compound statement.

> **orelse** : **typing.Union[ForwardRef('If'), statement.Else, NoneType]**
> An optional `elif` or `else` clause. *`If`* signifies an `elif` block. *`Else`* signifies an `else` block. `None` signifies no `else` or `elif` block.

> **leading_lines** : **typing.Sequence[whitespace.EmptyLine]**
> Sequence of empty lines appearing before this compound statement line.

> **whitespace_before_test** : **whitespace.SimpleWhitespace**
> The whitespace appearing after the `if` keyword but before the test expression.

> **whitespace_after_test** : **whitespace.SimpleWhitespace**
> The whitespace appearing after the test expression but before the colon.

class `libcst.`**Try**

> A regular `try` statement that cannot contain `ExceptStar` blocks. For `try` statements that can contain `ExceptStar` blocks, see `TryStar`.

> **body** : **statement.BaseSuite**
> The suite that is wrapped with a try statement.

> **handlers** : **typing.Sequence[statement.ExceptHandler]**
> A list of zero or more exception handlers.

> **orelse** : **typing.Optional[statement.Else]**
> An optional else case.

> **finalbody** : **typing.Optional[statement.Finally]**
> An optional finally case.

> **leading_lines** : **typing.Sequence[whitespace.EmptyLine]**
> Sequence of empty lines appearing before this compound statement line.

> **whitespace_before_colon** : **whitespace.SimpleWhitespace**
> The whitespace that appears after the `try` keyword but before the colon.

class libcst.**While**

A while statement.

test : expression.BaseExpression
The test we will loop against.

body : statement.BaseSuite
The suite that is wrapped with this statement.

orelse : typing.Optional[statement.Else]
An optional else case which will be executed if there is no *Break* statement encountered while looping.

leading_lines : typing.Sequence[whitespace.EmptyLine]
Sequence of empty lines appearing before this while statement.

whitespace_after_while : whitespace.SimpleWhitespace
Whitespace after the while keyword and before the test.

whitespace_before_colon : whitespace.SimpleWhitespace
Whitespace after the test and before the colon.

class libcst.**With**

A with statement.

items : typing.Sequence[statement.WithItem]
A sequence of one or more items that evaluate to context managers.

body : statement.BaseSuite
The suite that is wrapped with this statement.

asynchronous : typing.Optional[expression.Asynchronous]
Optional async modifier if this is an async with statement.

leading_lines : typing.Sequence[whitespace.EmptyLine]
Sequence of empty lines appearing before this with statement.

lpar : typing.Union[expression.LeftParen, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional open parenthesis for multi-line with bindings

rpar : typing.Union[expression.RightParen, MaybeSentinel] = MaybeSentinel.DEFAULT
Optional close parenthesis for multi-line with bindings

whitespace_after_with : whitespace.SimpleWhitespace
Whitespace after the with keyword and before the first item.

whitespace_before_colon : whitespace.SimpleWhitespace
Whitespace after the last item and before the colon.

10.4.3 Helper Nodes

Nodes that are used by various statements to represent some syntax, but are not statements on their own and cannot be used outside of the statements they belong with.

class libcst.**Annotation**

An annotation for a function (PEP 3107) or on a variable (PEP 526). Typically these are used in the context of type hints (PEP 484), such as:

```python
# a variable with a type
good_ideas: List[str] = []

# a function with type annotations
```

(continues on next page)

```
def concat(substrings: Sequence[str]) -> str:
    ...
```

> **annotation : expression.BaseExpression**
> > The annotation's value itself. This is the part of the annotation after the colon or arrow.
>
> **whitespace_before_indicator : typing.Union[whitespace.BaseParenthesizableWhitespace, M**
>
> **whitespace_after_indicator : whitespace.BaseParenthesizableWhitespace**

class libcst.**AsName**
> An as name clause inside an *ExceptHandler*, *ImportAlias* or *WithItem* node.
>
> **name : typing.Union[expression.Name, expression.Tuple, expression.List]**
> > Identifier that the parent node will be aliased to.
>
> **whitespace_before_as : whitespace.BaseParenthesizableWhitespace**
> > Whitespace between the parent node and the as keyword.
>
> **whitespace_after_as : whitespace.BaseParenthesizableWhitespace**
> > Whitespace between the as keyword and the name.

class libcst.**AssignTarget**
> A target for an *Assign*. Owns the equals sign and the whitespace around it.
>
> **target : expression.BaseAssignTargetExpression**
> > The target expression being assigned to.
>
> **whitespace_before_equal : whitespace.SimpleWhitespace**
> > The whitespace appearing before the equals sign.
>
> **whitespace_after_equal : whitespace.SimpleWhitespace**
> > The whitespace appearing after the equals sign.

class libcst.**BaseAssignTargetExpression**
> An expression that's valid on the left side of an assignment. That assignment may be part an *Assign* node, or it may be part of a number of other control structures that perform an assignment, such as a *For* loop.
>
> Python's grammar defines all expression as valid in this position, but the AST compiler further restricts the allowed types, which is what this type attempts to express.
>
> This is similar to a *BaseDelTargetExpression*, but it also includes *StarredElement* as a valid node.
>
> The set of valid nodes are defined as part of CPython's AST context computation.

class libcst.**BaseDelTargetExpression**
> An expression that's valid on the right side of a *Del* statement.
>
> Python's grammar defines all expression as valid in this position, but the AST compiler further restricts the allowed types, which is what this type attempts to express.
>
> This is similar to a *BaseAssignTargetExpression*, but it excludes *StarredElement*.
>
> The set of valid nodes are defined as part of CPython's AST context computation and as part of CPython's bytecode compiler.

class libcst.**Decorator**
> A single decorator that decorates a *FunctionDef* or a *ClassDef*.
>
> **decorator : typing.Union[expression.Name, expression.Attribute, expression.Call]**
> > The decorator that will return a new function wrapping the parent of this decorator.

leading_lines : **typing.Sequence[whitespace.EmptyLine]**
Line comments and empty lines before this decorator. The parent *[FunctionDef](#)* or *[ClassDef](#)* node owns leading lines before the first decorator so that if the first decorator is removed, spacing is preserved.

whitespace_after_at : **whitespace.SimpleWhitespace**
Whitespace after the @ and before the decorator expression itself.

trailing_whitespace : **whitespace.TrailingWhitespace**
Optional trailing comment and newline following the decorator before the next line.

class libcst.**Else**
An else clause that appears optionally after an *[If](#)*, *[While](#)*, *[Try](#)*, or *[For](#)* statement.

This node does not match elif clauses in *[If](#)* statements. It also does not match the required else clause in an *[IfExp](#)* expression (a = if b else c).

body : **statement.BaseSuite**
The body of else clause.

leading_lines : **typing.Sequence[whitespace.EmptyLine]**
Sequence of empty lines appearing before this compound statement line.

whitespace_before_colon : **whitespace.SimpleWhitespace**
The whitespace appearing after the else keyword but before the colon.

class libcst.**ExceptHandler**
An except clause that appears optionally after a *[Try](#)* statement.

body : **statement.BaseSuite**
The body of the except.

type : **typing.Optional[expression.BaseExpression]**
The type of exception this catches. Can be a tuple in some cases, or None if the code is catching all exceptions.

name : **typing.Optional[statement.AsName]**
The optional name that a caught exception is assigned to.

leading_lines : **typing.Sequence[whitespace.EmptyLine]**
Sequence of empty lines appearing before this compound statement line.

whitespace_after_except : **whitespace.SimpleWhitespace**
The whitespace between the except keyword and the type attribute.

whitespace_before_colon : **whitespace.SimpleWhitespace**
The whitespace after any type or name node (whichever comes last) and the colon.

class libcst.**Finally**
A finally clause that appears optionally after a *[Try](#)* statement.

body : **statement.BaseSuite**
The body of the except.

leading_lines : **typing.Sequence[whitespace.EmptyLine]**
Sequence of empty lines appearing before this compound statement line.

whitespace_before_colon : **whitespace.SimpleWhitespace**
The whitespace that appears after the finally keyword but before the colon.

class libcst.**ImportAlias**
An import, with an optional *[AsName](#)*. Used in both *[Import](#)* and *[ImportFrom](#)* to specify a single import out of another module.

---

name : typing.Union[expression.Attribute, expression.Name]
    Name or Attribute node representing the object we are importing.

asname : typing.Optional[statement.AsName]
    Local alias we will import the above object as.

comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
    Any trailing comma that appears after this import. This is optional for the last *ImportAlias* in a
    Import or *ImportFrom*, but all other import aliases inside an import must contain a comma to disam-
    biguate multiple imports.

property evaluated_name
    Returns the string name this *ImportAlias* represents.

property evaluated_alias
    Returns the string name for any alias that this *ImportAlias* has. If there is no asname attribute, this
    returns None.

class libcst.**NameItem**
    A single identifier name inside a *Global* or *Nonlocal* statement.

    This exists because a list of names in a global or nonlocal statement need to be separated by a comma,
    which ends up owned by the *NameItem* node.

name : expression.Name
    Identifier name.

comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
    This is forbidden for the last *NameItem* in a *Global*/*Nonlocal*, but all other tems inside a global
    or nonlocal statement must contain a comma to separate them.

class libcst.**Parameters**
    A function or lambda parameter list.

params : typing.Sequence[expression.Param]
    Positional parameters, with or without defaults. Positional parameters with defaults must all be after those
    without defaults.

star_arg : typing.Union[expression.Param, expression.ParamStar, MaybeSentinel] = Maybe

kwonly_params : typing.Sequence[expression.Param]
    Keyword-only params that may or may not have defaults.

star_kwarg : typing.Optional[expression.Param]
    Optional parameter that captures unspecified kwargs.

posonly_params : typing.Sequence[expression.Param]
    Positional-only parameters, with or without defaults. Positional-only parameters with defaults must all be
    after those without defaults.

posonly_ind : typing.Union[expression.ParamSlash, MaybeSentinel] = MaybeSentinel.DEFAU
    Optional sentinel that dictates parameters preceeding are positional-only args.

class libcst.**Param**
    A positional or keyword argument in a *Parameters* list. May contain an *Annotation* and, in some cases,
    a default.

name : expression.Name
    The parameter name itself.

annotation : typing.Optional[expression.Annotation]
    Any optional *Annotation*. These annotations are usually used as type hints.

equal : typing.Union[op.AssignEqual, MaybeSentinel] = MaybeSentinel.DEFAULT
The equal sign used to denote assignment if there is a default.

default : typing.Optional[expression.BaseExpression]
Any optional default value, used when the argument is not supplied.

comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
A trailing comma. If one is not provided, *MaybeSentinel* will be replaced with a comma only if a comma is required.

star : typing.Union[str, MaybeSentinel] = MaybeSentinel.DEFAULT
Zero, one, or two asterisks appearing before name for *Param*'s star_arg and star_kwarg.

whitespace_after_star : whitespace.BaseParenthesizableWhitespace
The whitespace before name. It will appear after star when a star exists.

whitespace_after_param : whitespace.BaseParenthesizableWhitespace
The whitespace after this entire node.

class libcst.**ParamSlash**
A sentinel indicator on a *Parameters* list to denote that the previous params are positional-only args.

This syntax is described in PEP 570.

comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT

class libcst.**ParamStar**
A sentinel indicator on a *Parameters* list to denote that the subsequent params are keyword-only args.

This syntax is described in PEP 3102.

comma : op.Comma

class libcst.**WithItem**
A single context manager in a *With* block, with an optional variable name.

item : expression.BaseExpression
Expression that evaluates to a context manager.

asname : typing.Optional[statement.AsName]
Variable to assign the context manager to, if it is needed in the *With* body.

comma : typing.Union[op.Comma, MaybeSentinel] = MaybeSentinel.DEFAULT
This is forbidden for the last *WithItem* in a *With*, but all other items inside a with block must contain a comma to separate them.

10.4.4 Statement Blocks

Nodes that represent some group of statements.

class libcst.**BaseSuite**
A dummy base-class for both *SimpleStatementSuite* and *IndentedBlock*. This exists to simplify type definitions and isinstance checks.

A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines.

—https://docs.python.org/3/reference/compound_stmts.html

---

class libcst.**SimpleStatementLine**
> A simple statement that's part of an IndentedBlock or Module. A simple statement is a series of small statements joined together by semicolons.
>
> This isn't differentiated from a *SimpleStatementSuite* in the grammar, but because a *SimpleStatementLine* can own additional whitespace that a *SimpleStatementSuite* doesn't have, we're differentiating it in the CST.
>
> **body : typing.Sequence[statement.BaseSmallStatement]**
> > Sequence of small statements. All but the last statement are required to have a semicolon.
>
> **leading_lines : typing.Sequence[whitespace.EmptyLine]**
> > Sequence of empty lines appearing before this simple statement line.
>
> **trailing_whitespace : whitespace.TrailingWhitespace**
> > Any optional trailing comment and the final NEWLINE at the end of the line.

class libcst.**SimpleStatementSuite**
> A simple statement that's used as a suite. A simple statement is a series of small statements joined together by semicolons. A suite is the thing that follows the colon in a compound statement.
>
> ```
> if test:<leading_whitespace><body><trailing_whitespace>
> ```
>
> This isn't differentiated from a *SimpleStatementLine* in the grammar, but because the two classes need to track different whitespace, we're differentiating it in the CST.
>
> **body : typing.Sequence[statement.BaseSmallStatement]**
> > Sequence of small statements. All but the last statement are required to have a semicolon.
>
> **leading_whitespace : whitespace.SimpleWhitespace**
> > The whitespace between the colon in the parent statement and the body.
>
> **trailing_whitespace : whitespace.TrailingWhitespace**
> > Any optional trailing comment and the final NEWLINE at the end of the line.

class libcst.**IndentedBlock**
> Represents a block of statements beginning with an INDENT token and ending in a DEDENT token. Used as the body of compound statements, such as an if statement's body.
>
> A common alternative to an *IndentedBlock* is a *SimpleStatementSuite*, which can also be used as a *BaseSuite*, meaning that it can be used as the body of many compound statements.
>
> An *IndentedBlock* always occurs after a colon in a *BaseCompoundStatement*, so it owns the trailing whitespace for the compound statement's clause.
>
> ```
> if test: # IndentedBlock's header
>     body
> ```
>
> **body : typing.Sequence[statement.BaseStatement]**
> > Sequence of statements belonging to this indented block.
>
> **header : whitespace.TrailingWhitespace**
> > Any optional trailing comment and the final NEWLINE at the end of the line.
>
> **indent : typing.Optional[str]**
> > A string represents a specific indentation. A None value uses the modules's default indentation. This is included because indentation is allowed to be inconsistent across a file, just not ambiguously.
>
> **footer : typing.Sequence[whitespace.EmptyLine]**
> > Any trailing comments or lines after the dedent that are owned by this indented block. Statements own preceeding and same-line trailing comments, but not trailing lines, so it falls on *IndentedBlock* to own it. In the case that a statement follows an *IndentedBlock*, that statement will own the comments and

lines that are at the same indent as the statement, and this *IndentedBlock* will own the comments and lines that are indented further.

10.5 Operators

Nodes that are used to signify an operation to be performed on a variable or value.

10.5.1 Unary Operators

Nodes that are used with *UnaryOperation* to perform some unary operation.

class libcst.**BitInvert**

class libcst.**Minus**

class libcst.**Not**

class libcst.**Plus**
> A unary operator that can be used in a *UnaryOperation* expression.
>
> > **whitespace_after : whitespace.BaseParenthesizableWhitespace**
> > > Any space that appears directly after this operator.

In addition, *BaseUnaryOp* is defined purely for typing and isinstance checks.

class libcst.**BaseUnaryOp**

10.5.2 Boolean Operators

Nodes that are used with *BooleanOperation* to perform some boolean operation.

class libcst.**And**

class libcst.**Or**
> A boolean operator that can be used in a *BooleanOperation* expression.
>
> > **whitespace_before : whitespace.BaseParenthesizableWhitespace**
> > > Any space that appears directly before this operator.
>
> > **whitespace_after : whitespace.BaseParenthesizableWhitespace**
> > > Any space that appears directly after this operator.

In addition, *BaseBooleanOp* is defined purely for typing and isinstance checks.

class libcst.**BaseBooleanOp**

10.5.3 Binary Operators

Nodes that are used with *BinaryOperation* to perform some binary operation.

class libcst.**Add**

class libcst.**BitAnd**

class libcst.**BitOr**

class libcst.**BitXor**

class libcst.**Divide**

---

class libcst.**FloorDivide**

class libcst.**LeftShift**

class libcst.**MatrixMultiply**

class libcst.**Modulo**

class libcst.**Multiply**

class libcst.**Power**

class libcst.**RightShift**

class libcst.**Subtract**
    A binary operator that can be used in a *BinaryOperation* expression.

    whitespace_before : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly before this operator.

    whitespace_after : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly after this operator.

In addition, *BaseBinaryOp* is defined purely for typing and isinstance checks.

class libcst.**BaseBinaryOp**

10.5.4 Comparison Operators

Nodes that are used with *Comparison* to perform some comparison operation.

class libcst.**Equal**

class libcst.**GreaterThan**

class libcst.**GreaterThanEqual**

class libcst.**In**

class libcst.**Is**

class libcst.**LessThan**

class libcst.**LessThanEqual**
    A comparision operator that can be used in a *Comparison* expression.

    whitespace_before : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly before this operator.

    whitespace_after : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly after this operator.

class libcst.**NotEqual**
    A comparison operator that can be used in a *Comparison* expression.

This node defines a static value for convenience, but in reality due to PEP 401 it can be one of two values, both of which should be a *NotEqual Comparison* operator.

    value : str = "!="
        The actual text value of this operator. Can be either != or <>.

    whitespace_before : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly before this operator.

    whitespace_after : whitespace.BaseParenthesizableWhitespace
        Any space that appears directly after this operator.

class libcst.**IsNot**

class libcst.**NotIn**

A comparision operator that can be used in a [`Comparison`](#) expression.

This operator spans two tokens that must be separated by at least one space, so there is a third whitespace attribute to represent this.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this operator.

whitespace_between : whitespace.BaseParenthesizableWhitespace
Any space that appears between the `not` and `in` tokens.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this operator.

In addition, [`BaseCompOp`](#) is defined purely for typing and isinstance checks.

class libcst.**BaseCompOp**

10.5.5 Augmented Assignment Operators

Nodes that are used with [`AugAssign`](#) to perform some agumented assignment.

class libcst.**AddAssign**

class libcst.**BitAndAssign**

class libcst.**BitOrAssign**

class libcst.**BitXorAssign**

class libcst.**DivideAssign**

class libcst.**FloorDivideAssign**

class libcst.**LeftShiftAssign**

class libcst.**MatrixMultiplyAssign**

class libcst.**ModuloAssign**

class libcst.**MultiplyAssign**

class libcst.**PowerAssign**

class libcst.**RightShiftAssign**

class libcst.**SubtractAssign**
An augmented assignment operator that can be used in a [`AugAssign`](#) statement.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this operator.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this operator.

In addition, [`BaseAugOp`](#) is defined purely for typing and isinstance checks.

class libcst.**BaseAugOp**

10.6 Miscellaneous

Miscelaneous nodes that are purely syntactic trivia. While python requires these nodes in order to parse a module, statement or expression they do not carry any meaning on their own.

class libcst.**AssignEqual**

Used by *AnnAssign* to denote a single equal character when doing an assignment on top of a type annotation. Also used by *Param* and *Arg* to denote assignment of a default value, and by *FormattedStringExpression* to denote usage of self-documenting expressions.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this equal sign.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this equal sign.

class libcst.**Colon**

Used by *Slice* as a separator between subsequent expressions, and in *Lambda* to separate arguments and body.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this colon.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this colon.

class libcst.**Comma**

Syntactic trivia used as a separator between subsequent items in various parts of the grammar.

Some use-cases are:

- *Import* or *ImportFrom*.
- *FunctionDef* arguments.
- *Tuple*/*List*/*Set*/*Dict* elements.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this comma.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this comma.

class libcst.**Dot**

Used by *Attribute* as a separator between subsequent *Name* nodes.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this dot.

whitespace_after : whitespace.BaseParenthesizableWhitespace
Any space that appears directly after this dot.

class libcst.**ImportStar**

Used by *ImportFrom* to denote a star import instead of a list of importable objects.

class libcst.**Semicolon**

Used by any small statement (any subclass of *BaseSmallStatement* such as *Pass*) as a separator between subsequent nodes contained within a *SimpleStatementLine* or *SimpleStatementSuite*.

whitespace_before : whitespace.BaseParenthesizableWhitespace
Any space that appears directly before this semicolon.

whitespace_after : whitespace.BaseParenthesizableWhitespace
    Any space that appears directly after this semicolon.

10.7 Whitespace

Nodes that encapsulate pure whitespace.

class libcst.**Comment**
    A comment including the leading pound (#) character.

    The leading pound character is included in the 'value' property (instead of being stripped) to help re-enforce the idea that whitespace immediately after the pound character may be significant. E.g:

```
# comment with whitespace at the start (usually preferred)
#comment without whitespace at the start (usually not desirable)
```

    Usually wrapped in a *TrailingWhitespace* or *EmptyLine* node.

    value : str
        The comment itself. Valid values start with the pound (#) character followed by zero or more non-newline characters. Comments cannot include newlines.

class libcst.**EmptyLine**
    Represents a line with only whitespace/comments. Usually statements will own any *EmptyLine* nodes above themselves, and a *Module* will own the document's header/footer *EmptyLine* nodes.

    indent : bool = True
        An empty line doesn't have to correspond to the current indentation level. For example, this happens when all trailing whitespace is stripped and there is an empty line between two statements.

    whitespace : whitespace.SimpleWhitespace
        Extra whitespace after the indent, but before the comment.

    comment : typing.Optional[whitespace.Comment]
        An optional comment appearing after the indent and extra whitespace.

    newline : whitespace.Newline
        The newline character that terminates this empty line.

class libcst.**Newline**
    Represents the newline that ends an *EmptyLine* or a statement (as part of *TrailingWhitespace*).

    Other newlines may occur in the document after continuation characters (the backslash, \\), but those newlines are treated as part of the *SimpleWhitespace*.

    Optionally, a value can be specified in order to overwrite the module's default newline. In general, this should be left as the default, which is None. This is allowed because python modules are permitted to mix multiple unambiguous newline markers.

    value : typing.Optional[str]
        A value of None indicates that the module's default newline sequence should be used. A value of \\n or \\r\\n indicates that the exact value specified will be used for this newline.

class libcst.**ParenthesizedWhitespace**
    This is the kind of whitespace you might see inside a parenthesized expression or statement between two tokens when there is a newline without a line continuation (\\) character.

    https://docs.python.org/3/reference/lexical_analysis.html#implicit-line-joining

A parenthesized whitespace cannot be empty since it requires at least one *TrailingWhitespace*. If you have whitespace that does not contain comments or newlines, use *SimpleWhitespace* instead.

first_line : whitespace.TrailingWhitespace
    The whitespace that comes after the previous node, up to and including the end-of-line comment and newline.

empty_lines : typing.Sequence[whitespace.EmptyLine]
    Any lines after the first that contain only indentation and/or comments.

indent : bool
    Whether or not the final simple whitespace is indented regularly.

last_line : whitespace.SimpleWhitespace
    Extra whitespace after the indent, but before the next node.

property empty
    Indicates that this node is empty (zero whitespace characters). For *ParenthesizedWhitespace* this will always be False.

class libcst.**SimpleWhitespace**
    This is the kind of whitespace you might see inside the body of a statement or expression between two tokens. This is the most common type of whitespace.

    A simple whitespace cannot contain a newline character unless it is directly preceeded by a line continuation character (\). It can contain zero or more spaces or tabs. If you need a newline character without a line continuation character, use *ParenthesizedWhitespace* instead.

    Simple whitespace is often non-semantic (optional), but in cases where whitespace solves a grammar ambiguity between tokens (e.g. if test, versus iftest), it has some semantic value.

    An example *SimpleWhitespace* containing a space, a line continuation, a newline and another space is as follows:

```
SimpleWhitespace(r" \\n ")
```

value : str
    Actual string value of the simple whitespace. A legal value contains only space, \f and \t characters, and optionally a continuation (\) followed by a newline (\n or \r\n).

property empty
    Indicates that this node is empty (zero whitespace characters).

class libcst.**TrailingWhitespace**
    The whitespace at the end of a line after a statement. If a line contains only whitespace, *EmptyLine* should be used instead.

whitespace : whitespace.SimpleWhitespace
    Any simple whitespace before any comment or newline.

comment : typing.Optional[whitespace.Comment]
    An optional comment appearing after any simple whitespace.

newline : whitespace.Newline
    The newline character that terminates this trailing whitespace.

class libcst.**BaseParenthesizableWhitespace**
    This is the kind of whitespace you might see inside the body of a statement or expression between two tokens. This is the most common type of whitespace.

The list of allowed characters in a whitespace depends on whether it is found inside a parenthesized expression or not. This class allows nodes which can be found inside or outside a `()`, `[]` or `{}` section to accept either whitespace form.

https://docs.python.org/3/reference/lexical_analysis.html#implicit-line-joining

Parenthesizable whitespace may contain a backslash character (`\`), when used as a line-continuation character. While the continuation character isn't technically "whitespace", it serves the same purpose.

Parenthesizable whitespace is often non-semantic (optional), but in cases where whitespace solves a grammar ambiguity between tokens (e.g. `if test`, versus `iftest`), it has some semantic value.

abstract property empty
> Indicates that this node is empty (zero whitespace characters).

10.8 Maybe Sentinel

class libcst.**MaybeSentinel**
> A *MaybeSentinel* value is used as the default value for some attributes to denote that when generating code (when *Module.code* is evaluated) we should optionally include this element in order to generate valid code.
>
> *MaybeSentinel* is only used for "syntactic trivia" that most users shouldn't care much about anyways, like commas, semicolons, and whitespace.
>
> For example, a function call's *Arg.comma* value defaults to *MaybeSentinel.DEFAULT*. A comma is required after every argument, except for the last one. If a comma is required and *Arg.comma* is a *MaybeSentinel*, one is inserted.
>
> This makes manual node construction easier, but it also means that we safely add arguments to a preexisting function call without manually fixing the commas:
>
> ```
> >>> import libcst as cst
> >>> fn_call = cst.parse_expression("fn(1, 2)")
> >>> new_fn_call = fn_call.with_changes(
> ...     args=[*fn_call.args, cst.Arg(cst.Integer("3"))]
> ... )
> >>> dummy_module = cst.parse_module("")  # we need to use Module.code_for_node
> >>> dummy_module.code_for_node(fn_call)
> 'fn(1, 2)'
> >>> dummy_module.code_for_node(new_fn_call)
> 'fn(1, 2, 3)'
> ```
>
> Notice that a comma was automatically inserted after the second argument. Since the original second argument had no comma, it was initialized to *MaybeSentinel.DEFAULT*. During the code generation of the second argument, a comma was inserted to ensure that the resulting code is valid.
>
> > **Warning:** While this sentinel is used in place of nodes, it is not a *CSTNode*, and will not be visited by a *CSTVisitor*.
>
> Some other libraries, like RedBaron, take other approaches to this problem. RedBaron's tree is mutable (LibCST's tree is immutable), and so they're able to solve this problem with "proxy lists". Both approaches come with different sets of tradeoffs.
>
> **DEFAULT = 1**

VISITORS

class libcst.**CSTVisitor**

    The low-level base visitor class for traversing a CST. This should be used in conjunction with the *visit()* method on a *CSTNode* to visit each element in a tree starting with that node. Unlike *CSTTransformer*, instances of this class cannot modify the tree.

    When visiting nodes using a *CSTVisitor*, the return value of *visit()* will equal the passed in tree.

    on_visit(*node: CSTNode*) → bool

        Called every time a node is visited, before we've visited its children.

        Returns True if children should be visited, and returns False otherwise.

    on_leave(*original_node: CSTNode*) → None

        Called every time we leave a node, after we've visited its children. If the *on_visit()* function for this node returns False, this function will still be called on that node.

    on_visit_attribute(*node: CSTNode*, *attribute: str*) → None

        Called before a node's child attribute is visited and after we have called *on_visit()* on the node. A node's child attributes are visited in the order that they appear in source that this node originates from.

    on_leave_attribute(*original_node: CSTNode*, *attribute: str*) → None

        Called after a node's child attribute is visited and before we have called *on_leave()* on the node.

class libcst.**CSTTransformer**

    The low-level base visitor class for traversing a CST and creating an updated copy of the original CST. This should be used in conjunction with the *visit()* method on a *CSTNode* to visit each element in a tree starting with that node, and possibly returning a new node in its place.

    When visiting nodes using a *CSTTransformer*, the return value of *visit()* will be a new tree with any changes made in *on_leave()* calls reflected in its children.

    on_visit(*node: CSTNode*) → bool

        Called every time a node is visited, before we've visited its children.

        Returns True if children should be visited, and returns False otherwise.

    on_leave()

        Called every time we leave a node, after we've visited its children. If the *on_visit()* function for this node returns False, this function will still be called on that node.

        original_node is guaranteed to be the same node as is passed to *on_visit()*, so it is safe to do state-based checks using the is operator. Modifications should always be performed on the updated_node so as to not overwrite changes made by child visits.

        Returning *RemovalSentinel.REMOVE* indicates that the node should be removed from its parent. This is not always possible, and may raise an exception if this node is required. As a convenience, you can use *RemoveFromParent()* as an alias to *RemovalSentinel.REMOVE*.

on_visit_attribute(*node: CSTNode*, *attribute: str*) → None
    Called before a node's child attribute is visited and after we have called *on_visit()* on the node. A node's child attributes are visited in the order that they appear in source that this node originates from.

on_leave_attribute(*original_node: CSTNode*, *attribute: str*) → None
    Called after a node's child attribute is visited and before we have called *on_leave()* on the node.

    Unlike *on_leave()*, this function does not allow modifications to the tree and is provided solely for state management.

libcst.**RemoveFromParent**() → libcst._removal_sentinel.RemovalSentinel
    A convenience method for requesting that this node be removed by its parent. Use this in place of returning *RemovalSentinel* directly. For example, to remove all arguments unconditionally:

```python
def leave_Arg(
    self, original_node: cst.Arg, updated_node: cst.Arg
) -> Union[cst.Arg, cst.RemovalSentinel]:
    return RemoveFromParent()
```

class libcst.**RemovalSentinel**
    A *RemovalSentinel.REMOVE* value should be returned by a *CSTTransformer.on_leave()* method when we want to remove that child from its parent. As a convenience, this can be constructed by calling *libcst.RemoveFromParent()*.

    The parent node should make a best-effort to remove the child, but may raise an exception when removing the child doesn't make sense, or could change the semantics in an unexpected way. For example, a function definition with no name doesn't make sense, but removing one of the arguments is valid.

    In we can't automatically remove the child, the developer should instead remove the child by constructing a new parent in the parent's *on_leave()* call.

    We use this instead of None to force developers to be explicit about deletions. Because None is the default return value for a function with no return statement, it would be too easy to accidentally delete nodes from the tree by forgetting to return a value.

    REMOVE = 1

class libcst.**FlattenSentinel**
    A *FlattenSentinel* may be returned by a *CSTTransformer.on_leave()* method when one wants to replace a node with multiple nodes. The replaced node must be contained in a *Sequence* attribute such as *body*. This is generally the case for BaseStatement and *BaseSmallStatement*. For example to insert a print before every return:

```python
def leave_Return(
    self, original_node: cst.Return, updated_node: cst.Return
) -> Union[cst.Return, cst.RemovalSentinel, cst.FlattenSentinel[cst.
↪BaseSmallStatement]]:
    log_stmt = cst.Expr(cst.parse_expression("print('returning')"))
    return cst.FlattenSentinel([log_stmt, updated_node])
```

    Returning an empty *FlattenSentinel* is equivalent to returning cst.RemovalSentinel.REMOVE and is subject to its requirements.

11.1 Visit and Leave Helper Functions

While it is possible to subclass from *CSTVisitor* or *CSTTransformer* and override the on_visit/on_leave/on_visit_attribute/on_leave_attribute functions directly, it is not recommended. The default implementation for both visitors will look up any

visit_<Type[CSTNode]>, leave_<Type[CSTNode]>, visit_<Type[CSTNode]>_<attribute> and leave_<Type[CSTNode]>_<attribute> method on the visitor subclass and call them directly. If such a function exists for the node in question, the visitor base class will call the relevant function, respecting the above outlined semantics. If the function does not exist, the visitor base class will assume that you do not care about that node and visit its children for you without requiring a default implementation.

Much like on_visit, visit_<Type[CSTNode]> return a boolean specifying whether or not LibCST should visit a node's children. As a convenience, you can return None instead of a boolean value from your visit_<Type[CSTNode]> functions. Returning a None value is treated as a request for default behavior, which causes the visitor to traverse children. It is equivalent to returning True, but requires no explicit return.

For example, the below visitor will visit every function definition, traversing to its children only if the function name doesn't include the word "foo". Notice that we don't need to provide our own on_visit or on_leave function, nor do we need to provide visit and leave functions for the rest of the nodes which we do not care about. This will have the effect of visiting all strings not inside of functions that have "foo" in the name. Note that we take advantage of default behavior when we decline to return a value in visit_SimpleString.

```python
class FooingAround(libcst.CSTVisitor):
    def visit_FunctionDef(self, node: libcst.FunctionDef) -> bool:
        return "foo" not in node.name.value

    def visit_SimpleString(self, node: libcst.SimpleString) -> None:
        print(node.value)
```

An example Python REPL using the above visitor is as follows:

```python
>>> import libcst
>>> demo = libcst.parse_module("'abc'\n'123'\ndef foo():\n    'not printed'")
>>> _ = demo.visit(FooingAround())
'abc'
'123'
```

11.2 Traversal Order

Traversal of any parsed tree directly matches the order that tokens appear in the source which was parsed. LibCST will first call on_visit for the node. Then, for each of the node's child attributes, LibCST will call on_visit_attribute for the node's attribute, followed by running the same visit algorithm on each child node in the node's attribute. Then, on_leave_attribute is called. After each attribute has been fully traversed, LibCST will call on_leave for the node. Note that LibCST will only call on_visit_attribute and on_leave_attribute for attributes in which there might be a LibCST node as a child. It will not call attribute visitors for attributes which are built-in python types.

For example, take the following simple tree generated by calling parse_expression("1+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=[],
)
```

Assuming you have a visitor that overrides every convenience helper method available, methods will be called in this order:

```
visit_BinaryOperation
visit_BinaryOperation_lpar
leave_BinaryOperation_lpar
visit_BinaryOperation_left
visit_Integer
visit_Integer_lpar
leave_Integer_lpar
visit_Integer_rpar
leave_Integer_rpar
leave_Integer
leave_BinaryOperation_left
visit_BinaryOperation_operator
visit_Add
visit_Add_whitespace_before
visit_SimpleWhitespace
leave_SimpleWhitespace
leave_Add_whitespace_before
visit_Add_whitespace_after
visit_SimpleWhitespace
leave_SimpleWhitespace
leave_Add_whitespace_after
leave_Add
leave_BinaryOperation_operator
visit_BinaryOperation_right
visit_Integer
visit_Integer_lpar
leave_Integer_lpar
visit_Integer_rpar
leave_Integer_rpar
leave_Integer
leave_BinaryOperation_right
visit_BinaryOperation_rpar
leave_BinaryOperation_rpar
leave_BinaryOperation
```

11.3 Batched Visitors

A batchable visitor class is provided to facilitate performing operations that can be performed in parallel in a single traversal over a CST. An example of this is *metadata computation*.

class libcst.**BatchableCSTVisitor**

The low-level base visitor class for traversing a CST as part of a batched set of traversals. This should be used in conjunction with the `visit_batched()` function or the `visit_batched()` method from `MetadataWrapper` to visit a tree. Instances of this class cannot modify the tree.

> **get_visitors**()
>
> Returns a mapping of all the `visit_<Type[CSTNode]>`, `visit_<Type[CSTNode]>_<attribute>`, `leave_<Type[CSTNode]>` and *leave_<Type[CSTNode]>_<attribute>*' methods defined by this visitor, excluding all empty stubs.

libcst.**visit_batched**()

Do a batched traversal over `node` with all `visitors`.

`before_visit` and `after_leave` are provided as optional hooks to execute before the `visit_<Type[CSTNode]>` and after the `leave_<Type[CSTNode]>` methods from each visitor in `visitor` are executed by the batched visitor.

This function does not handle metadata dependency resolution for `visitors`. See `visit_batched()` from `MetadataWrapper` for batched traversal with metadata dependency resolution.

METADATA

12.1 Metadata APIs

LibCST ships with a metadata interface that defines a standardized way to associate nodes in a CST with arbitrary metadata while maintaining the immutability of the tree. The metadata interface is designed to be declarative and type safe. Here's a quick example of using the metadata interface to get line and column numbers of nodes through the *PositionProvider*:

```python
class NamePrinter(cst.CSTVisitor):
    METADATA_DEPENDENCIES = (cst.PositionProvider,)

    def visit_Name(self, node: cst.Name) -> None:
        pos = self.get_metadata(cst.PositionProvider, node).start
        print(f"{node.value} found at line {pos.line}, column {pos.column}")

wrapper = cst.metadata.MetadataWrapper(cst.parse_module("x = 1"))
result = wrapper.visit(NamePrinter())  # should print "x found at line 1, column 0"
```

More examples of using the metadata interface can be found on the *Metadata Tutorial*.

12.1.1 Accessing Metadata

To work with metadata you need to wrap a module with a *MetadataWrapper*. The wrapper provides a *resolve()* function and a *resolve_many()* function to generate metadata.

class libcst.metadata.**MetadataWrapper**

A wrapper around a *Module* that stores associated metadata for that module.

When a *MetadataWrapper* is constructed over a module, the wrapper will store a deep copy of the original module. This means MetadataWrapper(module).module == module is False.

This copying operation ensures that a node will never appear twice (by identity) in the same tree. This allows us to uniquely look up metadata for a node based on a node's identity.

__init__()

Parameters

- **module** – The module to wrap. This is deeply copied by default.

- **unsafe_skip_copy** – When true, this skips the deep cloning of the module. This can provide a small performance benefit, but you should only use this if you know that there are no duplicate nodes in your tree (e.g. this module came from the parser).

- **cache** – Pass the needed cache to wrapper to be used when resolving metadata.

property module
 The module that's wrapped by this MetadataWrapper. By default, this is a deep copy of the passed in
 module.

```
mw = ModuleWrapper(module)
# Because `mw.module is not module`, you probably want to do visit and do
# your analysis on `mw.module`, not `module`.
mw.module.visit(DoSomeAnalysisVisitor)
```

resolve()
 Returns a copy of the metadata mapping computed by `provider`.

resolve_many()
 Returns a copy of the map of metadata mapping computed by each provider in `providers`.

 The returned map does not contain any metadata from undeclared metadata dependencies that
 `providers` has.

visit(*visitor: CSTVisitorT*) → Module
 Convenience method to resolve metadata before performing a traversal over `self.module` with
 visitor. See *visit()*.

visit_batched()
 Convenience method to resolve metadata before performing a traversal over `self.module` with
 visitors. See *visit_batched()*.

If you're working with visitors, which extend *MetadataDependent*, metadata dependencies will be automatically
computed when visited by a *MetadataWrapper* and are accessible through *get_metadata()*

class `libcst.`**MetadataDependent**
 The low-level base class for all classes that declare required metadata dependencies. *CSTVisitor* and
 CSTTransformer extend this class.

METADATA_DEPENDENCIES = ()
 The set of metadata dependencies declared by this class.

metadata
 A cached copy of metadata computed by *resolve()*. Prefer using *get_metadata()* over accessing
 this attribute directly.

classmethod get_inherited_dependencies()
 Returns all metadata dependencies declared by classes in the MRO of `cls` that subclass this class.

 Recursively searches the MRO of the subclass for metadata dependencies.

resolve()
 Context manager that resolves all metadata dependencies declared by `self` (using
 get_inherited_dependencies()) on `wrapper` and caches it on `self` for use with
 get_metadata().

 Upon exiting this context manager, the metadata cache on `self` is cleared.

get_metadata()
 Returns the metadata provided by the `key` if it is accessible from this visitor. Metadata is accessible in a
 subclass of this class if `key` is declared as a dependency by any class in the MRO of this class.

12.1.2 Providing Metadata

Metadata is generated through provider classes that can be be passed to *MetadataWrapper.resolve()* or de-
clared as a dependency of a `MetadataDependent`. These providers are then resolved automatically using methods

provided by *MetadataWrapper*.

In most cases, you should extend *BatchableMetadataProvider* when writing a provider, unless you have a particular reason to not to use a batchable visitor. Only extend from BaseMetadataProvider if your provider does not use the visitor pattern for computing metadata for a tree.

class libcst.**BaseMetadataProvider**
> The low-level base class for all metadata providers. This class should be extended for metadata providers that are not visitor-based.
>
> This class is generic. A subclass of BaseMetadataProvider[T] will provider metadata of type T.
>
> > **gen_cache**
> > > Implement gen_cache to indicate the matadata provider depends on cache from external system. This function will be called by *FullRepoManager* to compute required cache object per file path.
> >
> > **set_metadata**(*node: CSTNode*, *value: _ProvidedMetadataT*) → None
> > > Record a metadata value value for node.
> >
> > **get_metadata**()
> > > The same method as *get_metadata()* except metadata is accessed from self._computed in addition to self.metadata. See *get_metadata()*.

class libcst.metadata.**BatchableMetadataProvider**
> The low-level base class for all batchable visitor-based metadata providers. Batchable providers should be preferred when possible as they are more efficient to run compared to non-batchable visitor-based providers. Inherits from *BatchableCSTVisitor*.
>
> This class is generic. A subclass of BatchableMetadataProvider[T] will provider metadata of type T.

class libcst.metadata.**VisitorMetadataProvider**
> The low-level base class for all non-batchable visitor-based metadata providers. Inherits from *CSTVisitor*.
>
> This class is generic. A subclass of VisitorMetadataProvider[T] will provider metadata of type T.

12.2 Metadata Providers

PositionProvider, *ByteSpanPositionProvider*, *WhitespaceInclusivePositionProvider*, *ExpressionContextProvider*, *ScopeProvider*, *QualifiedNameProvider*, *ParentNodeProvider*, and *TypeInferenceProvider* are currently provided. Each metadata provider may has its own custom data structure.

12.2.1 Position Metadata

There are two types of position metadata available. They both track the same position concept, but differ in terms of representation. One represents position with line and column numbers, while the other outputs byte offset and length pairs.

Line and column numbers are available through the metadata interface by declaring one of *PositionProvider* or *WhitespaceInclusivePositionProvider*. For most cases, *PositionProvider* is what you probably want.

Node positions are is represented with *CodeRange* objects. See *the above example*.

class libcst.metadata.**PositionProvider**
> Generates line and column metadata.
>
> These positions are defined by the start and ending bounds of a node ignoring most instances of leading and trailing whitespace when it is not syntactically significant.

The positions provided by this provider should eventually match the positions used by Pyre for equivalent nodes.

class libcst.metadata.**WhitespaceInclusivePositionProvider**
Generates line and column metadata.

The start and ending bounds of the positions produced by this provider include all whitespace owned by the node.

class libcst.metadata.**CodeRange**

start : libcst._position.CodePosition
Starting position of a node (inclusive).

end : libcst._position.CodePosition
Ending position of a node (exclusive).

class libcst.metadata.**CodePosition**

line : int
Line numbers are 1-indexed.

column : int
Column numbers are 0-indexed.

Byte offset and length pairs can be accessed using *ByteSpanPositionProvider*. This provider represents positions using *CodeSpan*, which will contain the byte offsets of a *CSTNode* from the start of the file, and its length (also in bytes).

class libcst.metadata.**ByteSpanPositionProvider**
Generates offset and length metadata for nodes' positions.

For each CSTNode this provider generates a *CodeSpan* that contains the byte-offset of the node from the start of the file, and its length (also in bytes). The whitespace owned by the node is not included in this length.

Note: offset and length measure bytes, not characters (which is significant for example in the case of Unicode characters encoded in more than one byte)

class libcst.metadata.**CodeSpan**
Represents the position of a piece of code by its starting position and length.

Note: This class does not specify the unit of distance - it can be bytes, Unicode characters, or something else entirely.

start
Offset of the code from the beginning of the file. Can be 0.

length
Length of the span

12.2.2 Expression Context Metadata

class libcst.metadata.**ExpressionContextProvider**
Provides *ExpressionContext* metadata (mimics the expr_context in ast) for the following node types: *Attribute*, *Subscript*, *StarredElement*, *List*, *Tuple* and *Name*. Not that a *Name* may not always has context because of the differences between ast and LibCST. E.g. *attr* is a *Name* in LibCST but a str in ast. To honor ast implementation, we don't assignment context to *attr*.

Three context types *ExpressionContext.STORE*, *ExpressionContext.LOAD* and *ExpressionContext.DEL* are provided.

class libcst.metadata.**ExpressionContext**

Used in *ExpressionContextProvider* to represent context of a variable reference.

LOAD = 1

Load the value of a variable reference.

```
>>> libcst.MetadataWrapper(libcst.parse_module("a")).resolve(libcst.
↪ExpressionContextProvider)
mappingproxy({Name(
            value='a',
            lpar=[],
            rpar=[],
        ): <ExpressionContext.LOAD: 1>})
```

STORE = 2

Store a value to a variable reference by *Assign* (=), *AugAssign* (e.g. +=, -=, etc), or *AnnAssign*.

```
>>> libcst.MetadataWrapper(libcst.parse_module("a = b")).resolve(libcst.
↪ExpressionContextProvider)
mappingproxy({Name(
            value='a',
            lpar=[],
            rpar=[],
        ): <ExpressionContext.STORE: 2>, Name(
            value='b',
            lpar=[],
            rpar=[],
        ): <ExpressionContext.LOAD: 1>})
```

DEL = 3

Delete value of a variable reference by del.

```
>>> libcst.MetadataWrapper(libcst.parse_module("del a")).resolve(libcst.
↪ExpressionContextProvider)
mappingproxy({Name(
            value='a',
            lpar=[],
            rpar=[],
        ): < ExpressionContext.DEL: 3 >})
```

12.2.3 Scope Metadata

Scopes contain and separate variables from each other. Scopes enforce that a local variable name bound inside of a function is not available outside of that function.

While many programming languages are "block-scoped", Python is function-scoped. New scopes are created for classes, functions, and comprehensions. Other block constructs like conditional statements, loops, and try…except don't create their own scope.

There are five different type of scope in Python: *BuiltinScope*, *GlobalScope*, *ClassScope*, *FunctionScope*, and *ComprehensionScope*.
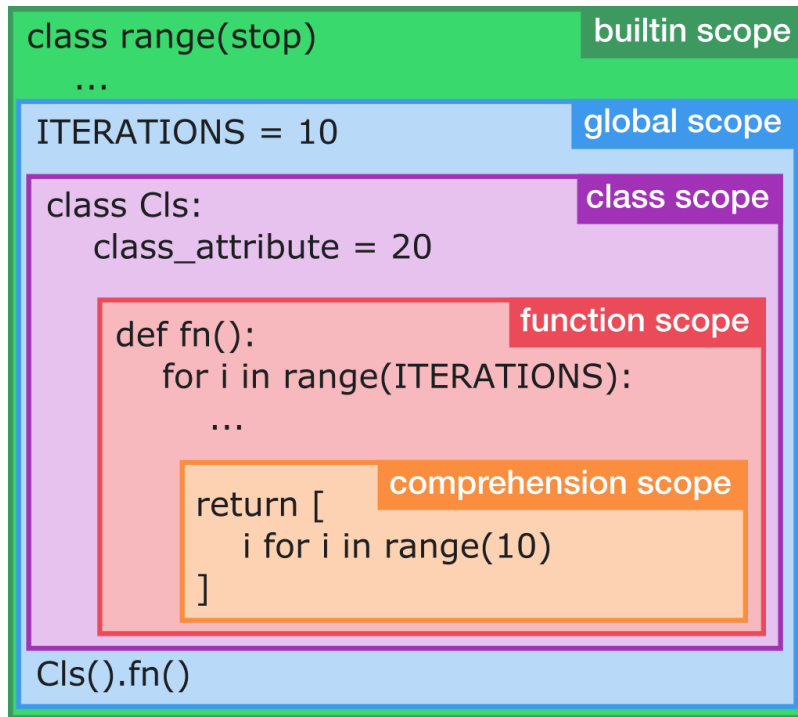
LibCST allows you to inspect these scopes to see what local variables are assigned or accessed within.

Note: Import statements bring new symbols into scope that are declared in other files. As such, they are represented by *Assignment* for scope analysis purposes. Dotted imports (e.g. `import a.b.c`) generate multiple *Assignment* objects — one for each module. When analyzing references, only the most specific access is recorded.

For example, the above `import a.b.c` statement generates three *Assignment* objects: one for `a`, one for `a.b`, and one for `a.b.c`. A reference for `a.b.c` records an access only for the last assignment, while a reference for `a.d` only records an access for the *Assignment* representing `a`.

---

class `libcst.metadata.`**ScopeProvider**
> *ScopeProvider* traverses the entire module and creates the scope inheritance structure. It provides the scope
> of name assignment and accesses. It is useful for more advanced static analysis. E.g. given a *FunctionDef*
> node, we can check the type of its Scope to figure out whether it is a class method (*ClassScope*) or a regular
> function (*GlobalScope*).
>
> Scope metadata is available for most node types other than formatting information nodes (whitespace, parenthe-
> ses, etc.).

class `libcst.metadata.`**BaseAssignment**
> Abstract base class of *Assignment* and BuitinAssignment.
>
> **name**
>> The name of assignment.
>
> **scope**
>> The scope associates to assignment.
>
> **property references**
>> Return all accesses of the assignment.

class `libcst.metadata.`**Access**
> An Access records an access of an assignment.

Note: This scope analysis only analyzes access via a *Name* or a *Name* node embedded in other node like *Call* or *Attribute*. It doesn't support type annotation using *SimpleString* literal for forward references. E.g. in this example, the `"Tree"` isn't parsed as an access:

```python
class Tree:
    def __new__(cls) -> "Tree":
        ...
```

node : typing.Union[libcst._nodes.expression.Name, libcst._nodes.expression.Attribute,
The node of the access. A name is an access when the expression context is *ExpressionContext.LOAD*. This is usually the name node representing the access, except for: 1) dotted imports, when it might be the attribute that represents the most specific part of the imported symbol; and 2) string annotations, when it is the entire string literal

scope : 'Scope'
The scope of the access. Note that a access could be in a child scope of its assignment.

is_annotation : bool

is_type_hint : bool

property referents
Return all assignments of the access.

record_assignment(*assignment: libcst.metadata.scope_provider.BaseAssignment*) → None

record_assignments(*name: str*) → None

class libcst.metadata.**Assignment**
An assignment records the name, CSTNode and its accesses.

node
The node of assignment, it could be a *Import*, *ImportFrom*, *Name*, *FunctionDef*, or *ClassDef*.

get_qualified_names_for()

class libcst.metadata.**BuiltinAssignment**
A BuiltinAssignment represents an value provide by Python as a builtin, including functions, constants, and types.

get_qualified_names_for()

class libcst.metadata.**Scope**
Base class of all scope classes. Scope object stores assignments from imports, variable assignments, function definition or class definition. A scope has a parent scope which represents the inheritance relationship. That means an assignment in parent scope is viewable to the child scope and the child scope may overwrites the assignment by using the same name.

Use `name in scope` to check whether a name is viewable in the scope. Use `scope[name]` to retrieve all viewable assignments in the scope.

Note: This scope analysis module only analyzes local variable names and it doesn't handle attribute names; for example, given `a.b.c = 1`, local variable name `a` is recorded as an assignment instead of `c` or `a.b.c`. To analyze the assignment/access of arbitrary object attributes, we leave the job to type inference metadata provider coming in the future.

parent
Parent scope. Note the parent scope of a GlobalScope is itself.

globals
> Refers to the GlobalScope.

abstract __contains__(*name: str*) → bool
> Check if the name str exist in current scope by `name in scope`.

abstract __getitem__()
> Get assignments given a name str by `scope[name]`.

> **Note:** *Why does it return a list of assignments given a name instead of just one assignment?*
>
> Many programming languages differentiate variable declaration and assignment. Further, those programming languages often disallow duplicate declarations within the same scope, and will often hoist the declaration (without its assignment) to the top of the scope. These design decisions make static analysis much easier, because it's possible to match a name against its single declaration for a given scope.
>
> As an example, the following code would be valid in JavaScript:
>
> ```
> function fn() {
>   console.log(value);  // value is defined here, because the declaration is
> →hoisted, but is currently 'undefined'.
>   var value = 5;  // A function-scoped declaration.
> }
> fn();  // prints 'undefined'.
> ```
>
> In contrast, Python's declaration and assignment are identical and are not hoisted:
>
> ```python
> if conditional_value:
>     value = 5
> elif other_conditional_value:
>     value = 10
> print(value)  # possibly valid, depending on conditional execution
> ```
>
> This code may throw a `NameError` if both conditional values are falsy. It also means that depending on the codepath taken, the original declaration could come from either `value = ...` assignment node. As a result, instead of returning a single declaration, we're forced to return a collection of all of the assignments we think could have defined a given name by the time a piece of code is executed. For the above example, value would resolve to a set of both assignments.

---

get_qualified_names_for()
> Get all *QualifiedName* in current scope given a *CSTNode*. The source of a qualified name can be either *QualifiedNameSource.IMPORT*, *QualifiedNameSource.BUILTIN* or *QualifiedNameSource.LOCAL*. Given the following example, c has qualified name `a.b.c` with source `IMPORT`, f has qualified name `Cls.f` with source `LOCAL`, a has qualified name `Cls.f.<locals>.a`, i has qualified name `Cls.f.<locals>.<comprehension>.i`, and the builtin `int` has qualified name `builtins.int` with source `BUILTIN`:
>
> ```python
> from a.b import c
> class Cls:
>     def f(self) -> "c":
>         c()
>         a = int("1")
>         [i for i in c()]
> ```
>
> We extends PEP-3155 (defines `__qualname__` for class and function only; function namespace is followed by a `<locals>`) to provide qualified name for all *CSTNode* recorded by *Assignment* and

---

> *Access*. The namespace of a comprehension (*ListComp*, *SetComp*, *DictComp*) is represented with
> `<comprehension>`.
>
> An imported name may be used for type annotation with *SimpleString* and currently resolving the
> qualified given *SimpleString* is not supported considering it could be a complex type annotation in
> the string which is hard to resolve, e.g. `List[Union[int, str]]`.

> **property assignments**
>> Return an *Assignments* contains all assignmens in current scope.

> **property accesses**
>> Return an *Accesses* contains all accesses in current scope.

class `libcst.metadata.`**BuiltinScope**
> A BuiltinScope represents python builtin declarations. See https://docs.python.org/3/library/builtins.html

class `libcst.metadata.`**GlobalScope**
> A GlobalScope is the scope of module. All module level assignments are recorded in GlobalScope.

class `libcst.metadata.`**FunctionScope**
> When a function is defined, it creates a FunctionScope.

class `libcst.metadata.`**ClassScope**
> When a class is defined, it creates a ClassScope.

class `libcst.metadata.`**ComprehensionScope**
> Comprehensions and generator expressions create their own scope. For example, in
>
>> [i for i in range(10)]
>
> The variable `i` is only viewable within the ComprehensionScope.

class `libcst.metadata.`**Assignments**
> A container to provide all assignments in a scope.

> **__iter__**()
>> Iterate through all assignments by `for i in scope.assignments`.

> **__getitem__**()
>> Get assignments given a name str or *CSTNode* by `scope.assignments[node]`

> **__contains__**(*node: Union[str, libcst._nodes.base.CSTNode]*) → bool
>> Check if a name str or *CSTNode* has any assignment by `node in scope.assignments`

class `libcst.metadata.`**Accesses**
> A container to provide all accesses in a scope.

> **__iter__**()
>> Iterate through all accesses by `for i in scope.accesses`.

> **__getitem__**()
>> Get accesses given a name str or *CSTNode* by `scope.accesses[node]`

> **__contains__**(*node: Union[str, libcst._nodes.base.CSTNode]*) → bool
>> Check if a name str or *CSTNode* has any access by `node in scope.accesses`

12.2.4 Qualified Name Metadata

Qualified name provides an unambiguous name to locate the definition of variable and it's introduced for class and
function in PEP-3155. QualifiedNameProvider provides possible *QualifiedName* given a *CSTNode*.

We don't call it fully qualified name because the name refers to the current module which doesn't consider the hierar-
chy of code repository.

For fully qualified names, there's *FullyQualifiedNameProvider* which is similar to the above but takes the
current module's location (relative to some python root folder, usually the repository's root) into account.

class libcst.metadata.**QualifiedNameSource**
> An enumeration.

> **IMPORT = 1**

> **BUILTIN = 2**

> **LOCAL = 3**

class libcst.metadata.**QualifiedName**

> **name : str**
> > Qualified name, e.g. `a.b.c` or `fn.<locals>.var`.

> **source : scope_provider.QualifiedNameSource**
> > Source of the name, either *QualifiedNameSource.IMPORT*, *QualifiedNameSource.*
> > *BUILTIN* or *QualifiedNameSource.LOCAL*.

class libcst.metadata.**QualifiedNameProvider**
> Compute possible qualified names of a variable CSTNode (extends PEP-3155). It uses the
> *get_qualified_names_for()* underlying to get qualified names. Multiple qualified names may be re-
> turned, such as when we have conditional imports or an import shadows another. E.g., the provider finds `a.b`,
> `d.e` and `f.g` as possible qualified names of `c`:

```
>>> wrapper = MetadataWrapper(
>>>     cst.parse_module(dedent(
>>>     '''
>>>         if something:
>>>             from a import b as c
>>>         elif otherthing:
>>>             from d import e as c
>>>         else:
>>>             from f import g as c
>>>         c()
>>>     '''
>>>     ))
>>> )
>>> call = wrapper.module.body[1].body[0].value
>>> wrapper.resolve(QualifiedNameProvider)[call],
{
    QualifiedName(name="a.b", source=QualifiedNameSource.IMPORT),
    QualifiedName(name="d.e", source=QualifiedNameSource.IMPORT),
    QualifiedName(name="f.g", source=QualifiedNameSource.IMPORT),
}
```

> For qualified name of a variable in a function or a comprehension, please refer
> *get_qualified_names_for()* for more detail.

> **static has_name**(*visitor:* *libcst._metadata_dependent.MetadataDependent,*
> > *node:* *libcst._nodes.base.CSTNode,* *name:* *Union[str,*
> > *libcst.metadata.scope_provider.QualifiedName]*) → bool
> > Check if any of qualified name has the str name or *QualifiedName* name.

class libcst.metadata.**FullyQualifiedNameProvider**
> Provide fully qualified names for CST nodes. Like *QualifiedNameProvider*, but the provided
> *QualifiedName* instances have absolute identifier names instead of local to the current module.

This provider is initialized with the current module's fully qualified name, and can be used with *FullRepoManager*. The module's fully qualified name itself is stored as a metadata of the *Module* node. Compared to *QualifiedNameProvider*, it also resolves relative imports.

Example usage:

```
>>> mgr = FullRepoManager(".", {"dir/a.py"}, {FullyQualifiedNameProvider})
>>> wrapper = mgr.get_metadata_wrapper_for_path("dir/a.py")
>>> fqnames = wrapper.resolve(FullyQualifiedNameProvider)
>>> {type(k): v for (k, v) in fqnames.items()}
{<class 'libcst._nodes.module.Module'>: {QualifiedName(name='dir.a', source=
→<QualifiedNameSource.LOCAL: 3>)}}
```

12.2.5 Parent Node Metadata

A *CSTNode* only has attributes link to its child nodes and thus only top-down tree traversal is doable. Sometimes user may want to access the parent *CSTNode* for more information or traverse in bottom-up manner. We provide *ParentNodeProvider* for those use cases.

class libcst.metadata.**ParentNodeProvider**

12.2.6 Type Inference Metadata

Type inference is to automatically infer data types of expression for deeper understanding source code. In Python, type checkers like Mypy or Pyre analyze type annotations and infer types for expressions. *TypeInferenceProvider* is provided by Pyre Query API which requires setup watchman for incremental typechecking. FullRepoManger is built for manage the inter process communication to Pyre.

class libcst.metadata.**TypeInferenceProvider**

Access inferred type annotation through Pyre Query API. It requires setup watchman and start pyre server by running `pyre` command. The inferred type is a string of type annotation. E.g. `typing.List[libcst._nodes.expression.Name]` is the inferred type of name n in expression `n = [cst.Name("")]`. All name references use the fully qualified name regardless how the names are imported. (e.g. `import libcst; libcst.Name` and `import libcst as cst; cst.Name` refer to the same name.) Pyre infers the type of *Name*, *Attribute* and *Call* nodes. The inter process communication to Pyre server is managed by *FullRepoManager*.

class libcst.metadata.**FullRepoManager**

__init__()

Given project root directory with pyre and watchman setup, *FullRepoManager* handles the inter process communication to read the required full repository cache data for metadata provider like *TypeInferenceProvider*.

Parameters

- **paths** – a collection of paths to access full repository data.

- **providers** – a collection of metadata provider classes require accessing full repository data, currently supports

TypeInferenceProvider and *FullyQualifiedNameProvider*. :param timeout: number of seconds. Raises TimeoutExpired

when timeout.

---

property cache
    The full repository cache data for all metadata providers passed in the `providers` parameter when constructing *FullRepoManager*. Each provider is mapped to a mapping of path to cache.

resolve_cache() → None
    Resolve cache for all providers that require it. Normally this is called by *get_cache_for_path()* so you do not need to call it manually. However, if you intend to do a single cache resolution pass before forking, it is a good idea to call this explicitly to control when cache resolution happens.

get_cache_for_path()
    Retrieve cache for a source file. The file needs to appear in the `paths` parameter when constructing *FullRepoManager*.

```
manager = FullRepoManager(".", {"a.py", "b.py"}, {TypeInferenceProvider})
MetadataWrapper(module, cache=manager.get_cache_for_path("a.py"))
```

get_metadata_wrapper_for_path(*path: str*) → libcst.metadata.wrapper.MetadataWrapper
    Create a *MetadataWrapper* given a source file path. The path needs to be a path relative to project root directory. The source code is read and parsed as *Module* for *MetadataWrapper*.

```
manager = FullRepoManager(".", {"a.py", "b.py"}, {TypeInferenceProvider})
wrapper = manager.get_metadata_wrapper_for_path("a.py")
```

MATCHERS

Matchers are provided as a way of asking whether a particular LibCST node and its children match a particular shape. It is possible to write a visitor that tracks attributes using `visit_<Node>` methods. It is also possible to implement manual instance checking and traversal of a node's children. However, both are cumbersome to write and hard to understand. Matchers offer a more concise way of defining what attributes on a node matter when matching against predefined patterns.

To accomplish this, a matcher has been created which corresponds to each LibCST node documented in *Nodes*. Matchers default each of their attributes to the special sentinel matcher *DoNotCare()*. When constructing a matcher, you can initialize the node with only the values of attributes that you are concerned with, leaving the rest of the attributes set to *DoNotCare()* in order to skip comparing against them.

13.1 Matcher APIs

13.1.1 Functions

Matchers can be used either by calling *matches()* or *findall()* directly, or by using various decorators to selectively control when LibCST calls visitor functions.

libcst.matchers.**matches**(*node:*   *Union[libcst._maybe_sentinel.MaybeSentinel, libcst._removal_sentinel.RemovalSentinel, libcst._nodes.base.CSTNode], matcher: libcst.matchers._matcher_base.BaseMatcherNode, *, metadata_resolver: Union[libcst._metadata_dependent.MetadataDependent, libcst.metadata.wrapper.MetadataWrapper, None] = None*) → bool

    Given an arbitrary node from a LibCST tree, and an arbitrary matcher, returns `True` if the node matches the shape defined by the matcher. Note that the node can also be a *RemovalSentinel* or a *MaybeSentinel* in order to use matches directly on transform results and node attributes. In these cases, *matches()* will always return `False`.

    The matcher can be any concrete matcher that subclasses from *BaseMatcherNode*, or a *OneOf*/*AllOf* special matcher. It cannot be a *MatchIfTrue* or a *DoesNotMatch()* matcher since these are redundant. It cannot be a *AtLeastN* or *AtMostN* matcher because these types are wildcards which can only be used inside sequences.

libcst.matchers.**findall**()

    Given an arbitrary node from a LibCST tree and an arbitrary matcher, iterates over that node and all children returning a sequence of all child nodes that match the given matcher. Note that the tree can also be a *RemovalSentinel* or a *MaybeSentinel* in order to use findall directly on transform results and node attributes. In these cases, *findall()* will always return an empty sequence. Note also that instead of a LibCST tree, you can instead pass in a *MetadataWrapper*. This mirrors the fact that you can call `visit` on a *MetadataWrapper* in order to iterate over it with a transform. If you provide a wrapper for the tree and do not set the `metadata_resolver` parameter specifically, it will automatically be set to the wrapper for you.

The matcher can be any concrete matcher that subclasses from *BaseMatcherNode*, or a *OneOf*/*AllOf* special matcher. Unlike *matches()*, it can also be a *MatchIfTrue* or *DoesNotMatch()* matcher, since we are traversing the tree looking for matches. It cannot be a *AtLeastN* or *AtMostN* matcher because these types are wildcards which can only be used inside sequences.

libcst.matchers.**extract**()
> Given an arbitrary node from a LibCST tree, and an arbitrary matcher, returns a dictionary of extracted children of the tree if the node matches the shape defined by the matcher. Note that the node can also be a *RemovalSentinel* or a *MaybeSentinel* in order to use extract directly on transform results and node attributes. In these cases, *extract()* will always return None.
>
> If the node matches the shape defined by the matcher, the return will be a dictionary whose keys are defined by the *SaveMatchedNode()* name parameter, and the values will be the node or sequence that was present at that location in the shape defined by the matcher. In the case of multiple *SaveMatchedNode()* matches with the same name, parent nodes will take prioirity over child nodes, and nodes later in sequences will take priority over nodes earlier in sequences.
>
> The matcher can be any concrete matcher that subclasses from *BaseMatcherNode*, or a *OneOf*/*AllOf* special matcher. It cannot be a *MatchIfTrue* or a *DoesNotMatch()* matcher since these are redundant. It cannot be a *AtLeastN* or *AtMostN* matcher because these types are wildcards which can only be used inside sequences.

libcst.matchers.**extractall**()
> Given an arbitrary node from a LibCST tree and an arbitrary matcher, iterates over that node and all children returning a sequence of dictionaries representing the saved and extracted children specified by *SaveMatchedNode()* for each match found in the tree. This is analogous to running a *findall()* over a tree, then running *extract()* with the same matcher over each of the returned nodes. Note that the tree can also be a *RemovalSentinel* or a *MaybeSentinel* in order to use extractall directly on transform results and node attributes. In these cases, *extractall()* will always return an empty sequence. Note also that instead of a LibCST tree, you can instead pass in a *MetadataWrapper*. This mirrors the fact that you can call visit on a *MetadataWrapper* in order to iterate over it with a transform. If you provide a wrapper for the tree and do not set the metadata_resolver parameter specifically, it will automatically be set to the wrapper for you.
>
> The matcher can be any concrete matcher that subclasses from *BaseMatcherNode*, or a *OneOf*/*AllOf* special matcher. Unlike *matches()*, it can also be a *MatchIfTrue* or *DoesNotMatch()* matcher, since we are traversing the tree looking for matches. It cannot be a *AtLeastN* or *AtMostN* matcher because these types are wildcards which can only be usedi inside sequences.

libcst.matchers.**replace**()
> Given an arbitrary node from a LibCST tree and an arbitrary matcher, iterates over that node and all children and replaces each node that matches the supplied matcher with a supplied replacement. Note that the replacement can either be a valid node type, or a callable which takes the matched node and a dictionary of any extracted child values and returns a valid node type. If you provide a valid LibCST node type, *replace()* will replace every node that matches the supplied matcher with the replacement node. If you provide a callable, *replace()* will run *extract()* over all matched nodes and call the callable with both the node that should be replaced and the dictionary returned by *extract()*. Under all circumstances a new tree is returned. *extract()* should be viewed as a short-cut to writing a transform which also returns a new tree even when no changes are applied.
>
> Note that the tree can also be a *RemovalSentinel* or a *MaybeSentinel* in order to use replace directly on transform results and node attributes. In these cases, *replace()* will return the same *RemovalSentinel* or *MaybeSentinel*. Note also that instead of a LibCST tree, you can instead pass in a *MetadataWrapper*. This mirrors the fact that you can call visit on a *MetadataWrapper* in order to iterate over it with a transform. If you provide a wrapper for the tree and do not set the metadata_resolver parameter specifically, it will automatically be set to the wrapper for you.
>
> The matcher can be any concrete matcher that subclasses from *BaseMatcherNode*, or a *OneOf*/*AllOf* special matcher. Unlike *matches()*, it can also be a *MatchIfTrue* or *DoesNotMatch()* matcher, since

we are traversing the tree looking for matches. It cannot be a *AtLeastN* or *AtMostN* matcher because these types are wildcards which can only be usedi inside sequences.

13.1.2 Decorators

The following decorators can be placed onto a method in a visitor or transformer in order to convert it into a visitor which is called when the provided matcher is true.

libcst.matchers.**visit**(*matcher:*   *libcst.matchers._matcher_base.BaseMatcherNode*)   →   Callable[[_CSTVisitFuncT], _CSTVisitFuncT]
    A decorator that allows a method inside a *MatcherDecoratableTransformer* or a *MatcherDecoratableVisitor* visitor to be called when visiting a node that matches the provided matcher. Note that you can use this in combination with *call_if_inside()* and *call_if_not_inside()* decorators. Unlike explicit visit_<Node> and leave_<Node> methods, functions decorated with this decorator cannot stop child traversal by returning False. Decorated visit functions should always have a return annotation of None.

    There is no restriction on the number of visit decorators allowed on a method. There is also no restriction on the number of methods that may be decorated with the same matcher. When multiple visit decorators are found on the same method, they act as a simple or, and the method will be called when any one of the contained matches is True.

libcst.matchers.**leave**(*matcher:*   *libcst.matchers._matcher_base.BaseMatcherNode*)   →   Callable[[_CSTVisitFuncT], _CSTVisitFuncT]
    A decorator that allows a method inside a *MatcherDecoratableTransformer* or a *MatcherDecoratableVisitor* visitor to be called when leaving a node that matches the provided matcher. Note that you can use this in combination with *call_if_inside()* and *call_if_not_inside()* decorators.

    There is no restriction on the number of leave decorators allowed on a method. There is also no restriction on the number of methods that may be decorated with the same matcher. When multiple leave decorators are found on the same method, they act as a simple or, and the method will be called when any one of the contained matches is True.

The following decorators can be placed onto any existing visit_<Node> or leave_<Node> visitor, as well as any visitor created using either *visit()* or *leave()*. They control whether the visitor itself gets called or skipped by LibCST when traversing a tree. Note that when a visitor function is skipped, its children will still be visited based on the rules set forth in *Visitors*. Namely, if you have a separate visit_<Node> visitor that returns False for a particular node, we will not traverse to its children.

libcst.matchers.**call_if_inside**(*matcher: libcst.matchers._matcher_base.BaseMatcherNode*) →   Callable[[_CSTVisitFuncT], _CSTVisitFuncT]
    A decorator for visit and leave methods inside a *MatcherDecoratableTransformer* or a *MatcherDecoratableVisitor*. A method that is decorated with this decorator will only be called if it or one of its parents matches the supplied matcher. Use this to selectively gate visit and leave methods to be called only when inside of another relevant node. Note that this works for both node and attribute methods, so you can decorate a visit_<Node> or a visit_<Node>_<Attr> method.

libcst.matchers.**call_if_not_inside**(*matcher: libcst.matchers._matcher_base.BaseMatcherNode*) → Callable[[_CSTVisitFuncT], _CSTVisitFuncT]
    A decorator for visit and leave methods inside a *MatcherDecoratableTransformer* or a *MatcherDecoratableVisitor*. A method that is decorated with this decorator will only be called if it or one of its parents does not match the supplied matcher. Use this to selectively gate visit and leave methods to be called only when outside of another relevant node. Note that this works for both node and attribute methods, so you can decorate a visit_<Node> or a visit_<Node>_<Attr> method.

When using matcher decorators, your visitors must subclass from *MatcherDecoratableVisitor* instead of *libcst.CSTVisitor*, and from *MatcherDecoratableTransformer* instead of *libcst.*

CSTTransformer. This is so that visitors and transformers not making use of matcher decorators do not pay the extra cost of their implementation. Note that if you do not subclass from *MatcherDecoratableVisitor* or *MatcherDecoratableTransformer*, you can still use the *matches()* function.

Both of these classes are strict subclasses of their corresponding LibCST base class, so they can be used anywhere that expects a LibCST base class. See *Visitors* for more information.

class libcst.matchers.**MatcherDecoratableVisitor**
> This class provides all of the features of a *libcst.CSTVisitor*, and additionally supports various decorators to control when methods get called when traversing a tree. Use this instead of a *libcst.CSTVisitor* if you wish to do more powerful decorator-based visiting.
>
> **on_visit**(*node: libcst._nodes.base.CSTNode*) → bool
> > Called every time a node is visited, before we've visited its children.
> >
> > Returns `True` if children should be visited, and returns `False` otherwise.
>
> **on_leave**(*original_node: libcst._nodes.base.CSTNode*) → None
> > Called every time we leave a node, after we've visited its children. If the *on_visit()* function for this node returns `False`, this function will still be called on that node.
>
> **on_visit_attribute**(*node: libcst._nodes.base.CSTNode, attribute: str*) → None
> > Called before a node's child attribute is visited and after we have called *on_visit()* on the node. A node's child attributes are visited in the order that they appear in source that this node originates from.
>
> **on_leave_attribute**(*original_node: libcst._nodes.base.CSTNode, attribute: str*) → None
> > Called after a node's child attribute is visited and before we have called *on_leave()* on the node.
>
> **matches**(*node: Union[libcst._maybe_sentinel.MaybeSentinel, libcst._removal_sentinel.RemovalSentinel, libcst._nodes.base.CSTNode], matcher: libcst.matchers._matcher_base.BaseMatcherNode*)
> > → bool
> > A convenience method to call *matches()* without requiring an explicit parameter for metadata. Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please see documentation for *matches()* as it is identical to this function.
>
> **findall**()
> > A convenience method to call *findall()* without requiring an explicit parameter for metadata. Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please see documentation for *findall()* as it is identical to this function.
>
> **extract**()
> > A convenience method to call *extract()* without requiring an explicit parameter for metadata. Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please see documentation for *extract()* as it is identical to this function.
>
> **extractall**()
> > A convenience method to call *extractall()* without requiring an explicit parameter for metadata. Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please see documentation for *extractall()* as it is identical to this function.
>
> **replace**()
> > A convenience method to call *replace()* without requiring an explicit parameter for metadata. Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please see documentation for *replace()* as it is identical to this function.

class libcst.matchers.**MatcherDecoratableTransformer**
> This class provides all of the features of a *libcst.CSTTransformer*, and additionally supports various decorators to control when methods get called when traversing a tree. Use this instead of a *libcst.CSTTransformer* if you wish to do more powerful decorator-based visiting.

on_visit(*node: libcst._nodes.base.CSTNode*) → bool
    Called every time a node is visited, before we've visited its children.

    Returns `True` if children should be visited, and returns `False` otherwise.

on_leave(*original_node: CSTNodeT*,   *updated_node: CSTNodeT*)   →   Union[CSTNodeT,
        libcst._removal_sentinel.RemovalSentinel]
    Called every time we leave a node, after we've visited its children. If the *on_visit()* function for this
    node returns `False`, this function will still be called on that node.

    `original_node` is guaranteed to be the same node as is passed to *on_visit()*, so it is safe
    to do state-based checks using the `is` operator. Modifications should always be performed on the
    `updated_node` so as to not overwrite changes made by child visits.

    Returning `RemovalSentinel.REMOVE` indicates that the node should be removed from its parent.
    This is not always possible, and may raise an exception if this node is required. As a convenience, you can
    use `RemoveFromParent()` as an alias to `RemovalSentinel.REMOVE`.

on_visit_attribute(*node: libcst._nodes.base.CSTNode*, *attribute: str*) → None
    Called before a node's child attribute is visited and after we have called *on_visit()* on the node. A
    node's child attributes are visited in the order that they appear in source that this node originates from.

on_leave_attribute(*original_node: libcst._nodes.base.CSTNode*, *attribute: str*) → None
    Called after a node's child attribute is visited and before we have called *on_leave()* on the node.

    Unlike *on_leave()*, this function does not allow modifications to the tree and is provided solely for
    state management.

matches(*node: Union[libcst._maybe_sentinel.MaybeSentinel, libcst._removal_sentinel.RemovalSentinel,*
        libcst._nodes.base.CSTNode], matcher: libcst.matchers._matcher_base.BaseMatcherNode)
        → bool
    A convenience method to call *matches()* without requiring an explicit parameter for metadata. Since
    our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please
    see documentation for *matches()* as it is identical to this function.

findall()
    A convenience method to call *findall()* without requiring an explicit parameter for metadata. Since
    our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please
    see documentation for *findall()* as it is identical to this function.

extract()
    A convenience method to call *extract()* without requiring an explicit parameter for metadata. Since
    our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please
    see documentation for *extract()* as it is identical to this function.

extractall()
    A convenience method to call *extractall()* without requiring an explicit parameter for metadata.
    Since our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver.
    Please see documentation for *extractall()* as it is identical to this function.

replace()
    A convenience method to call *replace()* without requiring an explicit parameter for metadata. Since
    our instance is an instance of *libcst.MetadataDependent*, we work as a metadata resolver. Please
    see documentation for *replace()* as it is identical to this function.

13.1.3 Traversal Order

Visit and leave functions created using *visit()* or *leave()* follow the traversal order rules laid out in LibCST's
visitor *Traversal Order* with one additional rule. Any visit function created using the *visit()* decorator will be
called **before** a `visit_<Node>` function if it is defined for your visitor. The order in which various visit functions

which are created with *visit()* are called is indeterminate, but all such functions will be called before calling the visit_<Node> method. Similarly, any leave function created using the *leave()* decorator will be called **after** a leave_<Node> function if it is defined for your visitor. The order in which various leave functions which are created with *leave()* are called is indeterminate, but all such functions will be called after calling the visit_<Node> function if it is defined for your visitor.

This has a few implications. The first is that if you return False from a visit_<Node> method, we are guaranteed to call your decorated visit functions as well. Second, when modifying a node in both leave_<Node> and a visitor created with *leave()*, the original_node will be unchanged for both and the updated_node available to the decorated leave method will be the node that is returned by the leave_<Node> method. Chaining modifications across multiple leave functions is supported, but must be done with care.

13.2 Matcher Types

13.2.1 Concrete Matchers

For each node found in *Nodes*, a corresponding concrete matcher has been generated. Each matcher has attributes identical to its LibCST node counterpart. For example, *libcst.Expr* includes the value and semicolon attributes, and therefore libcst.matchers.Expr similarly includes the same attributes. Just as *libcst.Expr*'s value is typed as taking a *libcst.BaseExpression*, libcst.matchers.Expr's value is typed as taking a libcst.matchers.BaseExpression. For every node that exists in LibCST, both concrete and abstract, a corresponding matcher has been defined.

There are a few special cases to the rules laid out above. For starters, matchers don't support evaluating *MaybeSentinel*. There is no way to specify that you wish to match against a *MaybeSentinel* except with the *DoNotCare()* matcher. This tends not to be an issue in practice because *MaybeSentinel* is only found on syntax nodes.

While there are base classes such as libcst.matchers.BaseExpression, you cannot match directly on them. They are provided for typing purposes only in order to exactly match the types on LibCST node attributes. If you need to match on all concrete subclasses of a base class, we recommend using the special matcher *OneOf*.

class libcst.matchers.**BaseMatcherNode**

Base class that all concrete matchers subclass from. *OneOf* and *AllOf* also subclass from this in order to allow them to be used in any place that a concrete matcher is allowed. This means that, for example, you can call *matches()* with a concrete matcher, or a *OneOf* with several concrete matchers as options.

13.2.2 Special Matchers

Special matchers are matchers that don't have a corresponding LibCST node. Concrete matchers only match against their corresponding LibCST node, limiting their use under certain circumstances. Special matchers fill in the gap by allowing higher-level logic constructs such as inversion. You can use any special matcher in place of a concrete matcher when specifying matcher attributes. Additionally, you can also use the *AllOf* and *OneOf* special matchers in place of a concrete matcher when calling *matches()* or using decorators.

class libcst.matchers.**OneOf**

Matcher that matches any one of its options. Useful when you want to match against one of several options for a single node. You can also construct a *OneOf* matcher by using Python's bitwise or operator with concrete matcher classes.

For example, you could match against True/False like:

```
m.OneOf(m.Name("True"), m.Name("False"))
```

Or you could use the shorthand, like:

```
m.Name("True") | m.Name("False")
```

property options
> The normalized list of options that we can choose from to satisfy a *OneOf* matcher. If any of these matchers are true, the *OneOf* matcher will also be considered a match.

class libcst.matchers.**AllOf**
> Matcher that matches all of its options. Useful when you want to match against a concrete matcher and a *MatchIfTrue* at the same time. Also useful when you want to match against a concrete matcher and a *DoesNotMatch()* at the same time. You can also construct a *AllOf* matcher by using Python's bitwise and operator with concrete matcher classes.
>
> For example, you could match against True in a roundabout way like:

```
m.AllOf(m.Name(), m.Name("True"))
```

> Or you could use the shorthand, like:

```
m.Name() & m.Name("True")
```

> Similar to *OneOf*, this can be used in place of any concrete matcher.
>
> Real-world cases where *AllOf* is useful are hard to come by but they are still provided for the limited edge cases in which they make sense. In the example above, we are redundantly matching against any LibCST *Name* node as well as LibCST *Name* nodes that have the value of True. We could drop the first option entirely and get the same result. Often, if you are using a *AllOf*, you can refactor your code to be simpler.
>
> For example, the following matches any function call to foo, and any function call which takes zero arguments:

```
m.AllOf(m.Call(func=m.Name("foo")), m.Call(args=()))
```

> This could be refactored into the following equivalent concrete matcher:

```
m.Call(func=m.Name("foo"), args=())
```

property options
> The normalized list of options that we can choose from to satisfy a *AllOf* matcher. If all of these matchers are true, the *AllOf* matcher will also be considered a match.

class libcst.matchers.**TypeOf**
> Matcher that matches any one of the given types. Useful when you want to work with trees where a common property might belong to more than a single type.
>
> For example, if you want either a binary operation or a boolean operation where the left side has a name foo:

```
m.TypeOf(m.BinaryOperation, m.BooleanOperation)(left = m.Name("foo"))
```

> Or you could use the shorthand, like:

```
(m.BinaryOperation | m.BooleanOperation)(left = m.Name("foo"))
```

> Also *TypeOf* matchers can be used with initalizing in the default state of other node matchers (without passing any extra patterns):

```
m.Name | m.SimpleString
```

> The will be equal to:

```
m.OneOf(m.Name(), m.SimpleString())
```

property initalized

property options

libcst.matchers.**DoesNotMatch**(*obj: _OtherNodeT*) → _OtherNodeT

Matcher helper that inverts the match result of its child. You can also invert a matcher by using Python's bitwise invert operator on concrete matchers or any special matcher.

For example, the following matches against any identifier that isn't `True`/`False`:

```
m.DoesNotMatch(m.OneOf(m.Name("True"), m.Name("False")))
```

Or you could use the shorthand, like:

```
~(m.Name("True") | m.Name("False"))
```

This can be used in place of any concrete matcher as long as it is not the root matcher. Calling *matches()* directly on a *DoesNotMatch()* is redundant since you can invert the return of *matches()* using a bitwise not.

class libcst.matchers.**MatchIfTrue**

Matcher that matches if its child callable returns `True`. The child callable should take one argument which is the attribute on the LibCST node we are trying to match against. This is useful if you want to do complex logic to determine if an attribute should match or not. One example of this is the *MatchRegex()* matcher build on top of *MatchIfTrue* which takes a regular expression and matches any string attribute where a regex match is found.

For example, to match on any identifier spelled with the letter `e`:

```
m.Name(value=m.MatchIfTrue(lambda value: "e" in value))
```

This can be used in place of any concrete matcher as long as it is not the root matcher. Calling *matches()* directly on a *MatchIfTrue* is redundant since you can just call the child callable directly with the node you are passing to *matches()*.

property func

The function that we will call with a LibCST node in order to determine if we match. If the function returns `True` then we consider ourselves to be a match.

libcst.matchers.**MatchRegex**()

Used as a convenience wrapper to *MatchIfTrue* which allows for matching a string attribute against a regex. `regex` can be any regular expression string or a compiled `Pattern`. This uses Python's re module under the hood and is compatible with syntax documented on docs.python.org.

For example, to match against any identifier that is at least one character long and only contains alphabetical characters:

```
m.Name(value=m.MatchRegex(r'[A-Za-z]+'))
```

This can be used in place of any string literal when constructing a concrete matcher.

class libcst.matchers.**MatchMetadata**

Matcher that looks up the metadata on the current node using the provided metadata provider and compares the value on the node against the value provided to *MatchMetadata*. If the metadata value does not exist for a particular node, *MatchMetadata* will always be considered not a match.

For example, to match against any function call which has one parameter which is used in a load expression context:

```
m.Call(
    args=[
        m.Arg(
            m.MatchMetadata(
                meta.ExpressionContextProvider,
                meta.ExpressionContext.LOAD,
            )
        )
    ]
)
```

To match against any *Name* node for the identifier foo which is the target of an assignment:

```
m.Name(
    value="foo",
    metadata=m.MatchMetadata(
        meta.ExpressionContextProvider,
        meta.ExpressionContext.STORE,
    )
)
```

This can be used in place of any concrete matcher as long as it is not the root matcher. Calling *matches()* directly on a *MatchMetadata* is redundant since you can just check the metadata on the root node that you are passing to *matches()*.

property key
> The metadata provider that we will use to fetch values when identifying whether a node matches this matcher. We compare the value returned from the metadata provider to the value provided in value when determining a match.

property value
> The value that we will compare against the return from the metadata provider for each node when determining a match.

class libcst.matchers.**MatchMetadataIfTrue**
> Matcher that looks up the metadata on the current node using the provided metadata provider and passes it to a callable which can inspect the metadata further, returning True if the matcher should be considered a match. If the metadata value does not exist for a particular node, *MatchMetadataIfTrue* will always be considered not a match.

> For example, to match against any arg whose qualified name might be typing.Dict:

```
m.Call(
    args=[
        m.Arg(
            m.MatchMetadataIfTrue(
                meta.QualifiedNameProvider,
                lambda qualnames: any(n.name == "typing.Dict" for n in qualnames)
            )
        )
    ]
)
```

> To match against any *Name* node for the identifier foo as long as that identifier is found at the beginning of an unindented line:

```
m.Name(
    value="foo",
```

```
    metadata=m.MatchMetadataIfTrue(
        meta.PositionProvider,
        lambda position: position.start.column == 0,
    )
)
```

This can be used in place of any concrete matcher as long as it is not the root matcher. Calling *matches()* directly on a *MatchMetadataIfTrue* is redundant since you can just check the metadata on the root node that you are passing to *matches()*.

property key

> The metadata provider that we will use to fetch values when identifying whether a node matches this matcher. We pass the value returned from the metadata provider to the callable given to us in func.

property func

> The function that we will call with a value retrieved from the metadata provider provided in key. If the function returns True then we consider ourselves to be a match.

libcst.matchers.**SaveMatchedNode**(*matcher: _OtherNodeT*, *name: str*) → _OtherNodeT

> Matcher helper that captures the matched node that matched against a matcher class, making it available in the dictionary returned by *extract()* or *extractall()*.

> For example, the following will match against any binary operation whose left and right operands are not integers, saving those expressions for later inspection. If used inside *extract()* or *extractall()*, the resulting dictionary will contain the keys left_operand and right_operand:

```
m.BinaryOperation(
    left=m.SaveMatchedNode(
        m.DoesNotMatch(m.Integer()),
        "left_operand",
    ),
    right=m.SaveMatchedNode(
        m.DoesNotMatch(m.Integer()),
        "right_operand",
    ),
)
```

> This can be used in place of any concrete matcher as long as it is not the root matcher. Calling *extract()* directly on a *SaveMatchedNode()* is redundant since you already have the reference to the node itself.

libcst.matchers.**DoNotCare**() → libcst.matchers._matcher_base.DoNotCareSentinel

> Used when you want to match exactly one node, but you do not care what node it is. Useful inside sequences such as a libcst.matchers.Call's args attribte. You do not need to use this for concrete matcher attributes since *DoNotCare()* is already the default.

> For example, the following matcher would match against any function calls with three arguments, regardless of the arguments themselves and regardless of the function name that we were calling:

```
m.Call(args=[m.DoNotCare(), m.DoNotCare(), m.DoNotCare()])
```

13.2.3 Sequence Wildcard Matchers

Sequence wildcard matchers are matchers that only get used when constructing a sequence to match against. Not all LibCST nodes have attributes which are sequences, but for those that do, sequence wildcard matchers offer a great degree of flexibility. Unlike all other matcher types, these allow you to match against more than one LibCST node, much like wildcards in regular expressions do.

LibCST does not implicitly match on partial sequences for you. So, when matching against a sequence you will need to provide a complete pattern. This often means using helpers such as *ZeroOrMore()* as the first and last element of your sequence. Think of it as the difference between Python's re.match and re.fullmatch functions. LibCST matchers behave like the latter so that it is possible to specify sequences which must start with, end with or be exactly equal to some pattern.

class libcst.matchers.**AtLeastN**

Matcher that matches n or more LibCST nodes in a row in a sequence. *AtLeastN* defaults to matching against the *DoNotCare()* matcher, so if you do not specify a matcher as a child, *AtLeastN* will match only by count. If you do specify a matcher as a child, *AtLeastN* will instead make sure that each LibCST node matches the matcher supplied.

For example, this will match all function calls with at least 3 arguments:

```
m.Call(args=[m.AtLeastN(n=3)])
```

This will match all function calls with 3 or more integer arguments:

```
m.Call(args=[m.AtLeastN(n=3, matcher=m.Arg(m.Integer()))])
```

You can combine sequence matchers with concrete matchers and special matchers and it will behave as you expect. For example, this will match all function calls that have 2 or more integer arguments in a row, followed by any arbitrary argument:

```
m.Call(args=[m.AtLeastN(n=2, matcher=m.Arg(m.Integer())), m.DoNotCare()])
```

And finally, this will match all function calls that have at least 5 arguments, the final one being an integer:

```
m.Call(args=[m.AtLeastN(n=4), m.Arg(m.Integer())])
```

> **property n**
> The number of nodes in a row that must match *AtLeastN.matcher* for this matcher to be considered a match. If there are less than n matches, this matcher will not be considered a match. If there are equal to or more than n matches, this matcher will be considered a match.

> **property matcher**
> The matcher which each node in a sequence needs to match.

libcst.matchers.**ZeroOrMore**()

Used as a convenience wrapper to *AtLeastN* when n is equal to 0. Use this when you want to match against any number of nodes in a sequence.

For example, this will match any function call with zero or more arguments, as long as all of the arguments are integers:

```
m.Call(args=[m.ZeroOrMore(m.Arg(m.Integer()))])
```

This will match any function call where the first argument is an integer and it doesn't matter what the rest of the arguments are:

```
m.Call(args=[m.Arg(m.Integer()), m.ZeroOrMore()])
```

You will often want to use *ZeroOrMore* on both sides of a concrete matcher in order to match against sequences that contain a particular node in an arbitrary location. For example, the following will match any function call that takes in at least one string argument anywhere:

```
m.Call(args=[m.ZeroOrMore(), m.Arg(m.SimpleString()), m.ZeroOrMore()])
```

class libcst.matchers.**AtMostN**

Matcher that matches n or fewer LibCST nodes in a row in a sequence. *AtMostN* defaults to matching against the *DoNotCare()* matcher, so if you do not specify a matcher as a child, *AtMostN* will match only by count. If you do specify a matcher as a child, *AtMostN* will instead make sure that each LibCST node matches the matcher supplied.

For example, this will match all function calls with 3 or fewer arguments:

```
m.Call(args=[m.AtMostN(n=3)])
```

This will match all function calls with 0, 1 or 2 string arguments:

```
m.Call(args=[m.AtMostN(n=2, matcher=m.Arg(m.SimpleString()))])
```

You can combine sequence matchers with concrete matchers and special matchers and it will behave as you expect. For example, this will match all function calls that have 0, 1 or 2 string arguments in a row, followed by an arbitrary argument:

```
m.Call(args=[m.AtMostN(n=2, matcher=m.Arg(m.SimpleString())), m.DoNotCare()])
```

And finally, this will match all function calls that have at least 2 arguments, the final one being a string:

```
m.Call(args=[m.AtMostN(n=2), m.Arg(m.SimpleString())])
```

> **property n**
> The number of nodes in a row that must match *AtLeastN.matcher* for this matcher to be considered a match. If there are less than or equal to n matches, then this matcher will be considered a match. Any more than n matches in a row and this matcher will stop matching and be considered not a match.

> **property matcher**
> The matcher which each node in a sequence needs to match.

libcst.matchers.**ZeroOrOne**()

Used as a convenience wrapper to *AtMostN* when n is equal to 1. This is effectively a maybe clause.

For example, this will match any function call with zero or one integer argument:

```
m.Call(args=[m.ZeroOrOne(m.Arg(m.Integer()))])
```

This will match any function call that has two or three arguments, and the first and last arguments are strings:

```
m.Call(args=[m.Arg(m.SimpleString()), m.ZeroOrOne(), m.Arg(m.SimpleString())])
```

CODEMODS

LibCST defines a codemod as an automated refactor that can be applied to a codebase of arbitrary size. Codemods are provided as a framework for writing higher-order transforms that consist of other, simpler transforms. It includes provisions for quickly creating a command-line interface to execute a codemod.

14.1 Codemod Base

All codemods derive from a common base, *Codemod*. This class includes a context, automatic metadata resolution and multi-pass transform support. Codemods are intended to be executed using the *transform_module()* interface.

class libcst.codemod.**Codemod**

> Abstract base class that all codemods must subclass from. Classes wishing to perform arbitrary, non-visitor-based mutations on a tree should subclass from this class directly. Classes wishing to perform visitor-based mutation should instead subclass from *ContextAwareTransformer*.
>
> Note that a *Codemod* is a subclass of *MetadataDependent*, meaning that you can declare metadata dependencies with the *METADATA_DEPENDENCIES* class property and while you are executing a transform you can call *get_metadata()* to retrieve the resolved metadata.
>
> **should_allow_multiple_passes**() → bool
>> Override this and return `True` to allow your transform to be called repeatedly until the tree doesn't change between passes. By default, this is off, and should suffice for most transforms.
>
> **warn**(*warning: str*) → None
>> Emit a warning that is displayed to the user who has invoked this codemod.
>
> **property module**
>> Reference to the currently-traversed module. Note that this is only available during the execution of a codemod. The module reference is particularly handy if you want to use *libcst.Module.code_for_node()* or *libcst.Module.config_for_parsing* and don't wish to track a reference to the top-level module manually.
>
> **abstract transform_module_impl**(*tree:* *libcst._nodes.module.Module*) → libcst._nodes.module.Module
>> Override this with your transform. You should take in the tree, optionally mutate it and then return the mutated version. The module reference and all calculated metadata are available for the lifetime of this function.
>
> **transform_module**(*tree: libcst._nodes.module.Module*) → libcst._nodes.module.Module
>> Transform entrypoint which handles multi-pass logic and metadata calculation for you. This is the method that you should call if you wish to invoke a codemod directly. This is the method that is called by *transform_module()*.

class libcst.codemod.**CodemodContext**

> A context holding all information that is shared amongst all transforms and visitors in a single codemod invocation. When chaining multiple transforms together, the context holds the state that needs to be passed between transforms. The context is responsible for keeping track of metadata wrappers and the filename of the file that is being modified (if available).
>
> **warnings**
>> List of warnings gathered while running a codemod. Add to this list by calling *warn()* method from a class that subclasses from *Codemod*, *ContextAwareTransformer* or *ContextAwareVisitor*.
>
> **scratch**
>> Scratch dictionary available for codemods which are spread across multiple transforms. Codemods are free to add to this at will.
>
> **filename**
>> The current filename if a codemod is being executed against a file that lives on disk. Populated by *libcst.codemod.parallel_exec_transform_with_prettyprint()* when running codemods from the command line.
>
> **full_module_name**
>> The current module if a codemod is being executed against a file that lives on disk, and the repository root is correctly configured. This Will take the form of a dotted name such as foo.bar.baz for a file in the repo named foo/bar/baz.py.
>
> **wrapper**
>> The current top level metadata wrapper for the module being modified. To access computed metadata when inside an actively running codemod, use the *get_metadata()* method on *Codemod*.
>
> **metadata_manager**
>> The current repo-level metadata manager for the active codemod.
>
> **property module**
>> The current top level module being modified. As a convenience, you can use the *module* property on *Codemod* to refer to this when inside an actively running codemod.

As a convenience, LibCST-compatible visitors are provided which extend the feature-set of *Codemod* to LibCST visitors and transforms. Remember that *ContextAwareTransformer* is still a *Codemod*, so you should still execute it using *transform_module()*.

class libcst.codemod.**ContextAwareTransformer**

> A transformer which visits using LibCST. Allows visitor-based mutation of a tree. Classes wishing to do arbitrary non-visitor-based mutation on a tree should instead subclass from *Codemod* and implement *transform_module_impl()*. This is a subclass of *MatcherDecoratableTransformer* so all features of matchers as well as *CSTTransformer* are available to subclasses of this class.

class libcst.codemod.**ContextAwareVisitor**

> A visitor which visits using LibCST. Allows visitor-based collecting of info on a tree. All codemods which wish to implement an information collector should subclass from this instead of directly from *MatcherDecoratableVisitor* or *CSTVisitor* since this provides access to the current codemod context. As a result, this class allows access to metadata which was calculated in a parent *Codemod* through the *get_metadata()* method.
>
> Note that you cannot directly run a *ContextAwareVisitor* using *transform_module()* because visitors by definition do not transform trees. However, you can instantiate a *ContextAwareVisitor* inside a codemod and pass it to the *visit* method on any node in order to run information gathering with metadata and context support.
>
> Remember that a *ContextAwareVisitor* is a subclass of *MetadataDependent*, meaning that you still need to declare your metadata dependencies with *METADATA_DEPENDENCIES* before you can retrieve metadata using *get_metadata()*, even if the parent codemod has listed its own metadata dependencies. Note

also that the dependencies listed on this class must be a strict subset of the dependencies listed in the parent codemod.

warn (*warning: str*) → None
> Emit a warning that is displayed to the user who has invoked this codemod.

property module
> Reference to the currently-traversed module. Note that this is only available during a transform itself.

It is often necessary to bail out of a codemod mid-operation when you realize that you do not want to operate on a module. This can be for any reason such as realizing the module includes some operation that you do not support. If you wish to skip a module, you can raise the *SkipFile* exception. For codemods executed using the *transform_module()* interface, all warnings emitted up to the exception being thrown will be preserved in the result.

class libcst.codemod.**SkipFile**
> Raise this exception to skip codemodding the current file.
>
> The exception message should be the reason for skipping.

Finally, its often easier to test codemods by writing verification tests instead of running repeatedly on your project. LibCST makes this easy with *CodemodTest*. Often you can develop the majority of your codemod using just tests, augmenting functionality when you run into an unexpected edge case when running it against your repository.

class libcst.codemod.**CodemodTest**
> Base test class for a *Codemod* test. Provides facilities for auto-instantiating and executing a codemod, given the args/kwargs that should be passed to it. Set the *TRANSFORM* class attribute to the *Codemod* class you wish to test and call *assertCodemod()* inside your test method to verify it transforms various source code chunks correctly.
>
> Note that this is a subclass of UnitTest so any *CodemodTest* can be executed using your favorite test runner such as the unittest module.
>
> **TRANSFORM = Ellipsis**
>
> **classmethod addClassCleanup** (*function*, **args*, ***kwargs*)
>> Same as addCleanup, except the cleanup items are called even if setUpClass fails (unlike tearDownClass).
>
> **assertCodeEqual** (*expected: str*, *actual: str*) → None
>> Given an expected and actual code string, makes sure they equal. This ensures that both the expected and actual are sanitized, so its safe to use this on strings that may have come from a triple-quoted multi-line string.
>
> **assertCodemod** ()
>> Given a before and after code string, and any args/kwargs that should be passed to the codemod constructor specified in *TRANSFORM*, validate that the codemod executes as expected. Verify that the codemod completes successfully, unless the expected_skip option is set to True, in which case verify that the codemod skips. Optionally, a *CodemodContext* can be provided. If none is specified, a default, empty context is created for you. Additionally, the python version for the code parser can be overridden to a valid python version string such as *"3.6"*. If none is specified, the version of the interpreter running your tests will be used. Also, a list of warning strings can be specified and *assertCodemod()* will verify that the codemod generates those warnings in the order specified. If it is left out, warnings are not checked.
>
> **assertNoLogs** (*logger=None*, *level=None*)
>> Fail unless no log messages of level *level* or higher are emitted on *logger_name* or its children.
>>
>> This method must be used as a context manager.
>
> **classmethod doClassCleanups** ()
>> Execute all class cleanup functions. Normally called for you after tearDownClass.

static make_fixture_data(*data: str*) → str
> Given a code string originting from a multi-line triple-quoted string, normalize the code using `dedent` and ensuring a trailing newline is present.

14.2 Execution Interface

As documented in the Codemod Base section above, codemods are meant to be programmatically executed using *transform_module()*. Executing in this manner handles all of the featureset of codemods, including metadata calculation and exception handling.

`libcst.codemod.`**transform_module**()
> Given a module as represented by a string and a *Codemod* that we wish to run, execute the codemod on the code and return a *TransformResult*. This should never raise an exception. On success, this returns a *TransformSuccess* containing any generated warnings as well as the transformed code. If the codemod is interrupted with a Ctrl+C, this returns a *TransformExit*. If the codemod elected to skip by throwing a *SkipFile* exception, this will return a *TransformSkip* containing the reason for skipping as well as any warnings that were generated before the codemod decided to skip. If the codemod throws an unexpected exception, this will return a *TransformFailure* containing the exception that occured as well as any warnings that were generated before the codemod crashed.

`libcst.codemod.`**TransformResult**
> alias of `typing.Union`

class `libcst.codemod.`**TransformSuccess**
> A *TransformResult* used when the codemod was successful. Stores all the information we might need to display to the user upon success, as well as the transformed file contents.

> **warning_messages**
> > All warning messages that were generated during the codemod.

> **code**
> > The updated code, post-codemod.

class `libcst.codemod.`**TransformFailure**
> A *TransformResult* used when the codemod failed. Stores all the information we might need to display to the user upon a failure.

> **warning_messages**
> > All warning messages that were generated before the codemod crashed.

> **error**
> > The exception that was raised during the codemod.

> **traceback_str**
> > The traceback string that was recorded at the time of exception.

class `libcst.codemod.`**TransformSkip**
> A *TransformResult* used when the codemod requested to be skipped. This could be because it's a generated file, or due to filename blacklist, or because the transform raised *SkipFile*.

> **skip_reason**
> > The reason that we skipped codemodding this module.

> **skip_description**
> > The description populated from the *SkipFile* exception.

> **warning_messages = ()**
> > All warning messages that were generated before the codemod decided to skip.

class libcst.codemod.**SkipReason**

An enumeration of all valid reasons for a codemod to skip.

GENERATED = 'generated'

The module was skipped because we detected that it was generated code, and we were configured to skip generated files.

BLACKLISTED = 'blacklisted'

The module was skipped because we detected that it was blacklisted, and we were configured to skip blacklisted files.

OTHER = 'other'

The module was skipped because the codemod requested us to skip using the *SkipFile* exception.

class libcst.codemod.**TransformExit**

A *TransformResult* used when the codemod was interrupted by the user (e.g. KeyboardInterrupt).

warning_messages = ()

An empty list of warnings, included so that all *TransformResult* have a warning_messages attribute.

14.3 Command-Line Support

LibCST includes additional support to facilitate faster development of codemods which are to be run at the command-line. This is achieved through the *CodemodCommand* class and the codemod utility which lives inside libcst.tool. The *CodemodCommand* class provides a codemod description and an interface to add arguments to the command-line. This is translated to a custom help message and command-line options that a user can provide when running a codemod at the command-line.

For a brief overview of supported universal options, run the codemod utility like so:

```
python3 -m libcst.tool codemod --help
```

The utility provides support for gathering up and parallelizing codemods across a series of files or directories, auto-formatting changed code according to a configured formatter, generating a unified diff of changes instead of applying them to files, taking code from stdin and codemodding it before returning to stdout, and printing progress and warnings to stderr during execution of a codemod.

Help is auto-customized if a codemod class is provided, including any added options and the codemod description. For an example, run the codemod utility like so:

```
python3 -m libcst.tool codemod noop.NOOPCommand --help
```

A second utility, list, can list all available codemods given your configuration. Run it like so:

```
python3 -m libcst.tool list
```

Finally, to set up a directory for codemodding using these tools, including additional directories where codemods can be found, use the initialize utility. To see help for how to use this, run the initialize utility like so:

```
python3 -m libcst.tool initialize --help
```

The above tools operate against any codemod which subclasses from *CodemodCommand*. Remember that *CodemodCommand* is a subclass of *Codemod*, so all of the features documented in the *Codemod Base* section are available in addition to command-line support. Any command-line enabled codemod can also be programmatically instantiated and invoked using the above-documented *transform_module()* interface.

class libcst.codemod.**CodemodCommand**

A *Codemod* which can be invoked on the command-line using the libcst.tool codemod utility. It behaves like any other codemod in that it can be instantiated and run identically to a *Codemod*. However, it provides support for providing help text and command-line arguments to libcst.tool codemod as well as facilities for automatically running certain common transforms after executing your *transform_module_impl()*.

The following list of transforms are automatically run at this time:

- *AddImportsVisitor* (adds needed imports to a module).

- *RemoveImportsVisitor* (removes unreferenced imports from a module).

DESCRIPTION = 'No description.'

An overrideable description attribute so that codemods can provide a short summary of what they do. This description will show up in command-line help as well as when listing available codemods.

static add_args(*arg_parser: argparse.ArgumentParser*) → None

Override this to add arguments to the CLI argument parser. These args will show up when the user invokes libcst.tool codemod with --help. They will also be presented to your class's __init__ method. So, if you define a command with an argument 'foo', you should also have a corresponding 'foo' positional or keyword argument in your class's __init__ method.

abstract transform_module_impl(*tree:* *libcst._nodes.module.Module*) →
libcst._nodes.module.Module

Override this with your transform. You should take in the tree, optionally mutate it and then return the mutated version. The module reference and all calculated metadata are available for the lifetime of this function.

Additionally, a few convenience classes have been provided which take the boilerplate out of common types of codemods:

class libcst.codemod.**VisitorBasedCodemodCommand**

A command that acts identically to a visitor-based transform, but also has the support of *add_args()* and running supported helper transforms after execution. See *CodemodCommand* and *ContextAwareTransformer* for additional documentation.

class libcst.codemod.**MagicArgsCodemodCommand**

A "magic" args command, which auto-magically looks up the transforms that are yielded from *get_transforms()* and instantiates them using values out of the context. Visitors yielded in *get_transforms()* must have constructor arguments that match a key in the context *scratch*. The easiest way to guarantee that is to use *add_args()* to add a command arg that will be parsed for each of the args. However, if you wish to chain transforms, adding to the scratch in one transform will make the value available to the constructor in subsequent transforms as well as the scratch for subsequent transforms.

abstract get_transforms()

A generator which yields one or more subclasses of *Codemod*. In the general case, you will usually yield a series of classes, but it is possible to programmatically decide which classes to yield depending on the contents of the context *scratch*.

Note that you should yield classes, not instances of classes, as the point of *MagicArgsCodemodCommand* is to instantiate them for you with the contents of *scratch*.

14.4 Command-Line Toolkit

Several helpers for constructing a command-line interface are provided. These are used in the codemod utility to provide LibCST's de-facto command-line interface but they are also available to be used directly in the case that circumstances demand a custom command-line tool.

libcst.codemod.**gather_files**()

> Given a list of files or directories (can be intermingled), return a list of all python files that exist at those locations. If `include_stubs` is `True`, this will include `.py` and `.pyi` stub files. If it is `False`, only `.py` files will be included in the returned list.

libcst.codemod.**exec_transform_with_prettyprint**()

> Given an instantiated codemod and a string representing a module, transform that code by executing the transform, optionally invoking the formatter and finally printing any generated warnings to stderr. If the code includes the generated marker at any spot and `include_generated` is not set to `True`, the code will not be modified. If `format_code` is set to `False` or the instantiated codemod does not modify the code, the code will not be formatted. If a `python_version` is provided, then we will parse the module using this version. Otherwise, we will use the version of the currently executing python binary.
>
> In all cases a module will be returned. Whether it is changed depends on the input parameters as well as the codemod itself.

libcst.codemod.**parallel_exec_transform_with_prettyprint**()

> Given a list of files and an instantiated codemod we should apply to them, fork and apply the codemod in parallel to all of the files, including any configured formatter. The `jobs` parameter controls the maximum number of in-flight transforms, and needs to be at least 1. If not included, the number of jobs will automatically be set to the number of CPU cores. If `unified_diff` is set to a number, changes to files will be printed to stdout with `unified_diff` lines of context. If it is set to `None` or left out, files themselves will be updated with changes and formatting. If a `python_version` is provided, then we will parse each source file using this version. Otherwise, we will use the version of the currently executing python binary.
>
> A progress indicator as well as any generated warnings will be printed to stderr. To supress the interactive progress indicator, set `hide_progress` to `True`. Files that include the generated code marker will be skipped unless the `include_generated` parameter is set to `True`. Similarly, files that match a supplied blacklist of regex patterns will be skipped. Warnings for skipping both blacklisted and generated files will be printed to stderr along with warnings generated by the codemod unless `hide_blacklisted` and `hide_generated` are set to `True`. Files that were successfully codemodded will not be printed to stderr unless `show_successes` is set to `True`.
>
> To make this API possible, we take an instantiated transform. This is due to the fact that lambdas are not pickleable and pickling functions is undefined. This means we're implicitly relying on fork behavior on UNIX-like systems, and this function will not work on Windows systems. To create a command-line utility that runs on Windows, please instead see *exec_transform_with_prettyprint()*.

class libcst.codemod.**ParallelTransformResult**

> The result of running *parallel_exec_transform_with_prettyprint()* against a series of files. This is a simple summary, with counts for number of successfully codemodded files, number of files that we failed to codemod, number of warnings generated when running the codemod across the files, and the number of files that we skipped when running the codemod.
>
> **successes**
> > Number of files that we successfully transformed.
>
> **failures**
> > Number of files that we failed to transform.
>
> **warnings**
> > Number of warnings generated when running transform across files.
>
> **skips**
> > Number of files skipped because they were blacklisted, generated or the codemod requested to skip.

libcst.codemod.**diff_code**()

> Given two strings representing a module before and after a codemod, produce a unified diff of the changes with `context` lines of context. Optionally, assign the `filename` to the change, and if it is not available,

assume that the change was performed on stdin/stdout. If no change is detected, return an empty string instead of returning an empty unified diff. This is comparable to revision control software which only shows differences for files that have changed.

14.5 Library of Transforms

LibCST additionally includes a library of transforms to reduce the need for boilerplate inside codemods. As of now, the list includes the following helpers.

class libcst.codemod.visitors.**GatherImportsVisitor**

Gathers all imports in a module and stores them as attributes on the instance. Intended to be instantiated and passed to a *Module visit()* method in order to gather up information about imports on a module. Note that this is not a substitute for scope analysis or qualified name support. Please see *Scope Analysis* for a more robust way of determining the qualified name and definition for an arbitrary node.

After visiting a module the following attributes will be populated:

> **module_imports** A sequence of strings representing modules that were imported directly, such as in the case of `import typing`. Each module directly imported but not aliased will be included here.

> **object_mapping** A mapping of strings to sequences of strings representing modules where we imported objects from, such as in the case of `from typing import Optional`. Each from import that was not aliased will be included here, where the keys of the mapping are the module we are importing from, and the value is a sequence of objects we are importing from the module.

> **module_aliases** A mapping of strings representing modules that were imported and aliased, such as in the case of `import typing as t`. Each module imported this way will be represented as a key in this mapping, and the value will be the local alias of the module.

> **alias_mapping** A mapping of strings to sequences of tuples representing modules where we imported objects from and aliased using `as` syntax, such as in the case of `from typing import Optional as opt`. Each from import that was aliased will be included here, where the keys of the mapping are the module we are importing from, and the value is a tuple representing the original object name and the alias.

> **all_imports** A collection of all *Import* and *ImportFrom* statements that were encountered in the module.

class libcst.codemod.visitors.**GatherExportsVisitor**

Gathers all explicit exports in a module and stores them as attributes on the instance. Intended to be instantiated and passed to a *Module visit()* method in order to gather up information about exports specified in an `__all__` variable inside a module.

After visiting a module the following attributes will be populated:

> **explicit_exported_objects** A sequence of strings representing objects that the module exports directly. Note that when `__all__` is absent, this attribute does not store default exported objects by name.

For more information on `__all__`, please see Python's Modules Documentation.

class libcst.codemod.visitors.**AddImportsVisitor**

Ensures that given imports exist in a module. Given a *CodemodContext* and a sequence of tuples specifying a module to import from as a string. Optionally an object to import from that module and any alias to assign that import, ensures that import exists. It will modify existing imports as necessary if the module in question is already being imported from.

This is one of the transforms that is available automatically to you when running a codemod. To use it in this manner, import *AddImportsVisitor* and then call the static *add_needed_import()* method, giving it the current context (found as `self.context` for all subclasses of *Codemod*), the module you wish to import from and optionally an object you wish to import from that module and any alias you would like to assign that import to.

For example:

```
AddImportsVisitor.add_needed_import(self.context, "typing", "Optional")
```

This will produce the following code in a module, assuming there was no typing import already:

```
from typing import Optional
```

As another example:

```
AddImportsVisitor.add_needed_import(self.context, "typing")
```

This will produce the following code in a module, assuming there was no import already:

```
import typing
```

Note that this is a subclass of *CSTTransformer* so it is possible to instantiate it and pass it to a *Module visit()* method. However, it is far easier to use the automatic transform feature of *CodemodCommand* and schedule an import to be added by calling *add_needed_import()*

static add_needed_import()
> Schedule an import to be added in a future invocation of this class by updating the `context` to include the `module` and optionally `obj` to be imported as well as optionally `alias` to alias the imported `module` or `obj` to. When subclassing from *CodemodCommand*, this will be performed for you after your transform finishes executing. If you are subclassing from a *Codemod* instead, you will need to call the *transform_module()* method on the module under modification with an instance of this class after performing your transform. Note that if the particular `module` or `obj` you are requesting to import already exists as an import on the current module at the time of executing *transform_module()* on an instance of *AddImportsVisitor*, this will perform no action in order to avoid adding duplicate imports.

class libcst.codemod.visitors.**RemoveImportsVisitor**
> Attempt to remove given imports from a module, dependent on whether there are any uses of the imported objects. Given a *CodemodContext* and a sequence of tuples specifying a module to remove as a string. Optionally an object being imported from that module and optionally an alias assigned to that imported object, ensures that that import no longer exists as long as there are no remaining references.

> Note that static analysis is able to determine safely whether an import is still needed given a particular module, but it is currently unable to determine whether an imported object is re-exported and used inside another module unless that object appears in an __any__ list.

> This is one of the transforms that is available automatically to you when running a codemod. To use it in this manner, import *RemoveImportsVisitor* and then call the static *remove_unused_import()* method, giving it the current context (found as `self.context` for all subclasses of *Codemod*), the module you wish to remove and optionally an object you wish to stop importing as well as an alias that the object is currently assigned to.

> For example:

```
RemoveImportsVisitor.remove_unused_import(self.context, "typing", "Optional")
```

> This will remove any `from typing import Optional` that exists in the module as long as there are no uses of `Optional` in that module.

As another example:

```
RemoveImportsVisitor.remove_unused_import(self.context, "typing")
```

This will remove any `import typing` that exists in the module, as long as there are no references to `typing` in that module, including references such as `typing.Optional`.

Additionally, *RemoveImportsVisitor* includes a convenience function *remove_unused_import_by_node()* which will attempt to schedule removal of all imports referenced in that node and its children. This is especially useful inside transforms when you are going to remove a node using *RemoveFromParent()* to get rid of a node.

For example:

```python
def leave_AnnAssign(
    self, original_node: cst.AnnAssign, updated_node: cst.AnnAssign,
) -> cst.RemovalSentinel:
    # Remove all annotated assignment statements, clean up imports.
    RemoveImportsVisitor.remove_unused_import_by_node(self.context, original_node)
    return cst.RemovalFromParent()
```

This will remove all annotated assignment statements from a module as well as clean up any imports that were only referenced in those assignments. Note that we pass the `original_node` to the helper function as it uses scope analysis under the hood which is only computed on the original tree.

Note that this is a subclass of *CSTTransformer* so it is possible to instantiate it and pass it to a *Module visit()* method. However, it is far easier to use the automatic transform feature of *CodemodCommand* and schedule an import to be added by calling *remove_unused_import()*

static remove_unused_import()
> Schedule an import to be removed in a future invocation of this class by updating the `context` to include the `module` and optionally `obj` which is currently imported as well as optionally `alias` that the imported `module` or `obj` is aliased to. When subclassing from *CodemodCommand*, this will be performed for you after your transform finishes executing. If you are subclassing from a *Codemod* instead, you will need to call the *transform_module()* method on the module under modification with an instance of this class after performing your transform. Note that if the particular `module` or `obj` you are requesting to remove is still in use somewhere in the current module at the time of executing *transform_module()* on an instance of *AddImportsVisitor*, this will perform no action in order to avoid removing an in-use import.

static remove_unused_import_by_node(*context: libcst.codemod._context.CodemodContext,*
> *node: libcst._nodes.base.CSTNode*) → None
> Schedule any imports referenced by `node` or one of its children to be removed in a future invocation of this class by updating the `context` to include the `module`, `obj` and `alias` for each import in question. When subclassing from *CodemodCommand*, this will be performed for you after your transform finishes executing. If you are subclassing from a *Codemod* instead, you will need to call the *transform_module()* method on the module under modification with an instance of this class after performing your transform. Note that all imports that are referenced by this `node` or its children will only be removed if they are not in use at the time of exeucting *transform_module()* on an instance of *AddImportsVisitor* in order to avoid removing an in-use import.

class libcst.codemod.visitors.**ApplyTypeAnnotationsVisitor**
> Apply type annotations to a source module using the given stub mdules. You can also pass in explicit annotations for functions and attributes and pass in new class definitions that need to be added to the source module.

> This is one of the transforms that is available automatically to you when running a codemod. To use it in this manner, import *ApplyTypeAnnotationsVisitor* and then call the static *store_stub_in_context()* method, giving it the current context (found as `self.context` for all subclasses of *Codemod*), the stub module from which you wish to add annotations.

For example, you can store the type annotation `int` for `x` using:

```
stub_module = parse_module("x: int = ...")

ApplyTypeAnnotationsVisitor.store_stub_in_context(self.context, stub_module)
```

You can apply the type annotation using:

```
source_module = parse_module("x = 1")
ApplyTypeAnnotationsVisitor.transform_module(source_module)
```

This will produce the following code:

```
x: int = 1
```

If the function or attribute already has a type annotation, it will not be overwritten.

To overwrite existing annotations when applying annotations from a stub, use the keyword argument `overwrite_existing_annotations=True` when constructing the codemod or when calling `store_stub_in_context`.

static store_stub_in_context(*context: libcst.codemod._context.CodemodContext, stub: libcst._nodes.module.Module, overwrite_existing_annotations: bool = False, use_future_annotations: bool = False, strict_posargs_matching: bool = True, strict_annotation_matching: bool = False*) → None

Store a stub module in the *CodemodContext* so that type annotations from the stub can be applied in a later invocation of this class.

If the `overwrite_existing_annotations` flag is `True`, the codemod will overwrite any existing annotations.

If you call this function multiple times, only the last values of `stub` and `overwrite_existing_annotations` will take effect.

transform_module_impl(*tree: libcst._nodes.module.Module*) → libcst._nodes.module.Module

Collect type annotations from all stubs and apply them to `tree`.

Gather existing imports from `tree` so that we don't add duplicate imports.

class libcst.codemod.visitors.**GatherUnusedImportsVisitor**

Collects all imports from a module not directly used in the same module. Intended to be instantiated and passed to a *libcst.Module visit()* method to process the full module.

Note that imports that are only used indirectly (from other modules) are still collected.

After visiting a module the attribute `unused_imports` will contain a set of unused *ImportAlias* objects, paired with their parent import node.

unused_imports

Contains a set of (alias, parent_import) pairs that are not used in the module after visiting.

filter_unused_imports()

Return the imports in `candidates` which are not used.

This function implements the main logic of this visitor, and is called after traversal. It calls *is_in_use()* on each import.

Override this in a subclass for additional filtering.

is_in_use(*scope: libcst.metadata.scope_provider.Scope*, *alias: libcst._nodes.statement.ImportAlias*)
→ bool
Check if `alias` is in use in the given `scope`.

An alias is in use if it's directly referenced, exported, or appears in a string type annotation. Override this in a subclass for additional filtering.

class libcst.codemod.visitors.**GatherCommentsVisitor**
Collects all comments matching a certain regex and their line numbers. This visitor is useful for capturing special-purpose comments, for example `noqa` style lint suppression annotations.

Standalone comments are assumed to affect the line following them, and inline ones are recorded with the line they are on.

After visiting a CST, matching comments are collected in the `comments` attribute.

comments
Dictionary of comments found in the CST. Keys are line numbers, values are comment nodes.

class libcst.codemod.visitors.**GatherNamesFromStringAnnotationsVisitor**
Collects all names from string literals used for typing purposes. This includes annotations like `foo: "SomeType"`, and parameters to special functions related to typing (currently only *typing.TypeVar*).

After visiting, a set of all found names will be available on the `names` attribute of this visitor.

names
The set of names collected from string literals.

HELPERS

Helpers are higher level functions built for reducing recurring code boilerplate. We add helpers as method of `CSTNode` or `libcst.helpers` package based on those principles:

- `CSTNode` method: simple, read-only and only require data of the direct children of a CSTNode.

- `libcst.helpers`: node transforms or require recursively traversing the syntax tree.

15.1 Construction Helpers

Functions that assist in creating a new LibCST tree.

`libcst.helpers.`**`parse_template_module`**(*template:* *str*, *config:* *libcst._parser.types.config.PartialParserConfig* *=* *PartialParserConfig(),* ***template_replacements*) → libcst._nodes.module.Module

Accepts an entire python module template, including all leading and trailing whitespace. Any *CSTNode* provided as a keyword argument to this function will be inserted into the template at the appropriate location similar to an f-string expansion. For example:

```
module = parse_template_module("from {mod} import Foo\n", mod=Name("bar"))
```

The above code will parse to a module containing a single `FromImport` statement, referencing module `bar` and importing object `Foo` from it. Remember that if you are parsing a template as part of a substitution inside a transform, its considered *best practice* to pass in a `config` from the current module under transformation.

Note that unlike *parse_module()*, this function does not support bytes as an input. This is due to the fact that it is processed as a template before parsing as a module.

`libcst.helpers.`**`parse_template_expression`**(*template:* *str*, *config:* *libcst._parser.types.config.PartialParserConfig* *=* *PartialParserConfig(),* ***template_replacements*) → libcst._nodes.expression.BaseExpression

Accepts an expression template on a single line. Leading and trailing whitespace is not valid (there's nowhere to store it on the expression node). Any *CSTNode* provided as a keyword argument to this function will be inserted into the template at the appropriate location similar to an f-string expansion. For example:

```
expression = parse_template_expression("x + {foo}", foo=Name("y")))
```

The above code will parse to a *BinaryOperation* expression adding two names (`x` and `y`) together.

Remember that if you are parsing a template as part of a substitution inside a transform, its considered *best practice* to pass in a `config` from the current module under transformation.

libcst.helpers.**parse_template_statement**(*template:* *str*, *config:*
libcst._parser.types.config.PartialParserConfig
= *PartialParserConfig()*, ***tem-*
plate_replacements) →
Union[libcst._nodes.statement.SimpleStatementLine,
libcst._nodes.statement.BaseCompoundStatement]

Accepts a statement template followed by a trailing newline. If a trailing newline is not provided, one will be added. Any *CSTNode* provided as a keyword argument to this function will be inserted into the template at the appropriate location similar to an f-string expansion. For example:

```
statement = parse_template_statement("assert x > 0, {msg}", msg=SimpleString('"Uh␣
↪oh!"'))
```

The above code will parse to an assert statement checking that some variable x is greater than zero, or providing the assert message `"Uh oh!"`.

Remember that if you are parsing a template as part of a substitution inside a transform, its considered *best practice* to pass in a `config` from the current module under transformation.

15.2 Transformation Helpers

Functions that assist in transforming an existing LibCST node.

libcst.helpers.**insert_header_comments**()

Insert comments after last non-empty line in header. Use this to insert one or more comments after any copyright preamble in a *Module*. Each comment in the list of `comments` must start with a # and will be placed on its own line in the appropriate location.

15.3 Traversing Helpers

Functions that assist in traversing an existing LibCST tree.

libcst.helpers.**get_full_name_for_node**()

Return a dot concatenated full name for str, *Name*, *Attribute*. *Call*, *Subscript*, *FunctionDef*, *ClassDef*, *Decorator*. Return None for not supported Node.

libcst.helpers.**get_full_name_for_node_or_raise**(*node:* *Union[str,*
libcst._nodes.base.CSTNode]) →
str

Return a dot concatenated full name for str, *Name*, *Attribute*. *Call*, *Subscript*, *FunctionDef*, *ClassDef*. Raise Exception for not supported Node.

libcst.helpers.**ensure_type**()

Takes any python object, and a LibCST *CSTNode* subclass and refines the type of the python object. This is most useful when you already know that a particular object is a certain type but your type checker is not convinced. Note that this does an instance check for you and raises an exception if it is not the right type, so this should be used in situations where you are sure of the type given previous checks.

EXPERIMENTAL APIS

These APIs may change at any time (including in minor releases) with no notice. You probably shouldn't use them, but if you do, you should pin your application to an exact release of LibCST to avoid breakages.

16.1 Reentrant Code Generation

class libcst.metadata.**ExperimentalReentrantCodegenProvider**

An experimental API that allows fast generation of modified code by recording an initial code-generation pass, and incrementally applying updates. It is a performance optimization for a few niche use-cases and is not user-friendly.

This API may change at any time without warning (including in minor releases).

This is rarely useful. Instead you should make multiple modifications to a single syntax tree, and generate the code once. However, we can think of a few use-cases for this API (hence, why it exists):

- When linting a file, you might generate multiple independent patches that a user can accept or reject. Depending on your architecture, it may be advantageous to avoid regenerating the file when computing each patch.

- You might want to call out to an external utility (e.g. a typechecker, such as pyre or mypy) to validate a small change. You may need to generate and test lots of these patches.

Restrictions:

- For safety and sanity reasons, the smallest/only level of granularity is a statement. If you need to patch part of a statement, you regenerate the entire statement. If you need to regenerate an entire module, just call *libcst.Module.code()*.

- This does not (currently) operate recursively. You can patch an unpatched piece of code multiple times, but you can't layer additional patches on an already patched piece of code.

class libcst.metadata.**CodegenPartial**

Provided by *ExperimentalReentrantCodegenProvider*.

Stores enough information to generate either a small patch (get_modified_code_range()) or a new file (get_modified_code()) by replacing the old node at this position.

start_offset

end_offset

has_trailing_newline

get_original_module_code() → str

Equivalent to *libcst.Module.bytes()* on the top-level module that contains this statement, except that it uses the cached result from our previous code generation pass, so it's faster.

get_original_module_bytes() → bytes
> Equivalent to *libcst.Module.bytes()* on the top-level module that contains this statement, except that it uses the cached result from our previous code generation pass, so it's faster.

get_original_statement_code() → str
> Equivalent to *libcst.Module.code_for_node()* on the current statement, except that it uses the cached result from our previous code generation pass, so it's faster.

get_modified_statement_code(*node: libcst._nodes.statement.BaseStatement*) → str
> Gets the new code for `node` as if it were in same location as the old statement being replaced. This means that it inherits details like the old statement's indentation.

get_modified_module_code(*node: libcst._nodes.statement.BaseStatement*) → str
> Gets the new code for the module at the root of this statement's tree, but with the supplied replacement `node` in its place.

get_modified_module_bytes(*node: libcst._nodes.statement.BaseStatement*) → bytes
> Gets the new bytes for the module at the root of this statement's tree, but with the supplied replacement `node` in its place.

INDICES AND TABLES

- genindex

- modindex

- search

EIGHTEEN

PRIVACY POLICY AND TERMS OF USE

- Privacy Policy
- Terms of Use

Symbols

A

B