`!symtable` --- Access to the compiler's symbol tables

- Generating Symbol Tables
- Examining Symbol Tables
- Command-Line Usage

symtable

Source code: `Lib/symtable.py`

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. `!symtable` provides an interface to examine these tables.

Generating Symbol Tables

symtable(code, filename, compile_type, *, module=None)

Return the toplevel `SymbolTable` for the Python source *code*. *code* can be a string, a bytes object, or an AST object, as for the builtin `compile`. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to `compile`. The optional argument *module* specifies the module name. It is needed to unambiguous `filter <warning-filter>` syntax warnings by module name.

3.15 Added the *module* parameter.

3.16 *code* can now be an AST object.

Examining Symbol Tables

An enumeration indicating the type of a `SymbolTable` object.

MODULE

Used for the symbol table of a module.

FUNCTION

Used for the symbol table of a function.

CLASS

Used for the symbol table of a class.

INLINED_COMPREHENSION

Used for the symbol table of a list, set or dict comprehension that is inlined into the enclosing code unit (see `709`). A symbol table of this type represents a sub-scope of the enclosing code unit's scope, and it does not correspond to a separate compilation unit.

3.16

The following members refer to different flavors of `annotation scopes <annotation-scopes>`.

ANNOTATION

Used for annotations if `from __future__ import annotations` is active.

TYPE_ALIAS

Used for the symbol table of `type` constructions.

TYPE_PARAMETERS

Used for the symbol table of `generic functions <generic-functions>` or `generic classes <generic-classes>`.

TYPE_VARIABLE

Used for the symbol table of the bound, the constraint tuple or the default value of a single type variable in the formal sense, i.e., a TypeVar, a TypeVarTuple or a ParamSpec object (the latter two do not support a bound or a constraint tuple).

3.13

A namespace table for a block. The constructor is not public.

get_type()

Return the type of the symbol table. Possible values are members of the `SymbolTableType` enumeration.

3.12 Added `'annotation'`, `'TypeVar bound'`, `'type alias'`, and `'type parameter'` as possible return values.

3.13 Return values are members of the `SymbolTableType` enumeration.

The exact values of the returned string may change in the future, and thus, it is recommended to use `SymbolTableType` members instead of hard-coded strings.

get_id()

Return the table's identifier.

get_name()

Return the table's name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (`get_type` returns `'module'`). For type parameter scopes (which are used for generic classes, functions, and type aliases), it is the name of the underlying class, function, or type alias. For type alias scopes, it is the name of the type alias. For `~typing.TypeVar` bound scopes, it is the name of the `TypeVar`.

get_lineno()

Return the number of the first line in the block this table represents.

is_optimized()

Return `True` if the locals in this table can be optimized.

is_nested()

Return `True` if the block is a nested class or function.

has_children()

Return `True` if the block has nested namespaces within it. These can be obtained with `get_children`.

get_identifiers()

Return a view object containing the names of symbols in the table. See the `documentation of view objects <dict-views>`.

lookup(name)

Lookup *name* in the table and return a `Symbol` instance.

get_symbols()

Return a list of `Symbol` instances for names in the table.

get_children()

Return a list of the nested symbol tables.

A namespace for a function or method. This class inherits from `SymbolTable`.

get_parameters()

Return a tuple containing names of parameters to this function.

get_locals()

Return a tuple containing names of locals in this function.

get_globals()

Return a tuple containing names of globals in this function.

get_nonlocals()

Return a tuple containing names of explicitly declared nonlocals in this function.

get_frees()

Return a tuple containing names of `free (closure) variables <closure variable>` in this function.

get_cells()

Return a tuple containing names of `cell variables <closure variable>` in this table.

3.15

A namespace of a class. This class inherits from `SymbolTable`.

An entry in a `SymbolTable` corresponding to an identifier in the source. The constructor is not public.

get_name()

Return the symbol's name.

is_referenced()

Return `True` if the symbol is used in its block.

is_imported()

Return `True` if the symbol is created from an import statement.

is_parameter()

Return `True` if the symbol is a parameter.

is_type_parameter()

Return `True` if the symbol is a type parameter.

3.14

is_global()

Return `True` if the symbol is global.

is_nonlocal()

Return `True` if the symbol is nonlocal.

is_declared_global()

Return `True` if the symbol is declared global with a global statement.

is_local()

Return `True` if the symbol is local to its block.

is_annotated()

Return `True` if the symbol is annotated.

3.6

is_free()

Return `True` if the symbol is referenced in its block, but not assigned to.

is_cell()

Return `True` if the symbol is referenced but not assigned in a nested block.

3.15

is_free_class()

Return *True* if a class-scoped symbol is free from the perspective of a method.

Consider the following example:

```
def f():
    x = 1  # function-scoped
    class C:
        x = 2  # class-scoped
        def method(self):
            return x
```

In this example, the class-scoped symbol `x` is considered to be free from the perspective of `C.method`, thereby allowing the latter to return *1* at runtime and not *2*.

3.14

is_assigned()

Return `True` if the symbol is assigned to in its block.

is_comp_iter()

Return `True` if the symbol is a comprehension iteration variable.

3.14

is_comp_cell()

Return `True` if the symbol is a cell in an inlined comprehension.

3.14

is_namespace()

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()

Return a list of namespaces bound to this name.

get_namespace()

Return the namespace bound to this name. If more than one or no namespace is bound to this name, a `ValueError` is raised.

Command-Line Usage

3.13

The `!symtable` module can be executed as a script from the command line.

```
python -m symtable [infile...]
```

Symbol tables are generated for the specified Python source files and dumped to stdout. If no input file is specified, the content is read from stdin.

□□□ #2
□ □□□ □□□ 2026-10-31 17:20:16 UTC
□ □□□ □□□ 2026-10-31 17:20:17 UTC