ChatGPT

Python AST & Compiler Spec for Async Transformations (3.11–3.13)

Executive Summary

This technical specification provides a detailed look at Python 3.11–3.13's AST and compilation behaviors, focusing on top-level `async`/`await` support and how to safely transform code for a custom async execution engine. Key findings include:

- **Top-Level** `await`: Since Python 3.8, the compiler flag `PyCF_ALLOW_TOP_LEVEL_AWAIT` enables top-level `await`, `async for`, and `async with` in code compiled with `compile()`. Using this flag (with appropriate `mode`) produces a code object marked as a coroutine (flag `CO_COROUTINE`) [1] [2]. Such code objects must be executed via an event loop (e.g. by awaiting them or using `asyncio.run`). Without the flag, top-level `await` or `async with/for` is a SyntaxError.

- **Symbol Table & Hoisting**: The `symtable` module can identify which names are imports, local assignments, global references, etc., helping decide if certain top-level statements (like imports or definitions) can be "hoisted" out of the async context. As a rule, **unconditional imports and definitions** that are not later re-bound by user code can be executed (or cached) globally to avoid repetition. We provide a recipe using `symtable` and AST analysis to detect if an imported name is later shadowed by an assignment or pattern match (in which case it should **not** be hoisted). This ensures we don't change semantics by hoisting something the user intended to override.

- **Precise Location Mapping (PEP 657)**: Modern Python AST nodes carry start (`lineno`, `col_offset`) and end (`end_lineno`, `end_col_offset`) positions for precise error reporting [3] [4]. Transformations should preserve these to maintain correct traceback info. Utilities like `ast.copy_location()` will copy over both start and end positions [5], and `ast.fix_missing_locations()` will fill in any missing line/column info [6]. If new wrapper code is added (shifting original code lines down), use `ast.increment_lineno()` to adjust line numbers of the original nodes [7] so that error messages still point to the original source lines.

- **AST Changes in 3.11–3.13**: New language features introduce new AST node types and fields that transform logic must handle:

- **Structural Pattern Matching (PEP 634, Python 3.10+)**: New AST nodes like `Match`, `match_case` (cases), and various `Pattern` subclasses represent `match...case` statements [8] [9]. Pattern variables appear as `ast.Name` nodes with `ctx=Store` in the AST, meaning they are bindings (just like assignments).
- **Exception Groups (PEP 654, Python 3.11)**: Adds `ast.TryStar` for `try/.../except*` blocks [10]. It has the same fields as `Try` (with a list of `ExceptHandler` nodes in `.handlers`), but indicates `except*` semantics.

- **Type Parameters (PEP 695, Python 3.12)**: Introduces generics syntax and the `type` alias statement. AST impacts include a new `ast.TypeAlias` node for `type X = ...` statements [11], and a new `type_params` field on `FunctionDef`, `AsyncFunctionDef`, and `ClassDef` nodes to hold generic type parameters [12] [13]. Also new node classes `ast.TypeVar`, `ast.ParamSpec`, `ast.TypeVarTuple` represent type parameter definitions in generics.
- **Formalized F-strings (PEP 701, Python 3.12)**: No new AST nodes, but fewer f-string syntax restrictions. F-string expressions can now include quotes matching the outer string, multiline expressions, comments, etc., whereas Python 3.11 would error on those [14]. AST transformations should be unaffected (f-strings still parse into `JoinedStr` and `FormattedValue` nodes), but be aware that code which was invalid before may now produce an AST.

- **Comprehension Inlining (PEP 709, Python 3.12)**: This optimization doesn't change AST structure but changes how comprehensions are compiled. Notably, comprehensions no longer create a separate function stack frame or separate symtable scope [15]. Tools using `symtable` will see comprehension variables in the parent scope's symbol table in 3.12+, whereas 3.11 produced a child scope per comprehension. This is a subtle change to keep in mind for symbol analysis.

- **Performance & Caching**: We benchmarked parsing and compilation to guide caching strategy. Direct compilation (`compile(source, ..., "exec")`) is fastest for executing code. Converting source to an AST (`ast.parse`) and then compiling can take roughly **2×** the time of direct compile for large code, due to Python object overhead in the AST. For example, in our tests compiling a 1000-line script 100× took ~0.44 seconds, whereas parsing to AST and then compiling took ~0.87 seconds (and doing an AST unparse took ~1.03 seconds for 100× on that same tree). Thus, if no AST transformation is needed, use direct `compile` to minimize latency. If transformations *are* needed, consider caching results: for repeated inputs, an LRU cache of compiled code objects (or even parsed ASTs) can greatly improve throughput. Key considerations for caching:

- **Cache Keys**: Include the source code (or a hash of it) plus any compile flags/mode that affect bytecode (e.g. `PyCF_ALLOW_TOP_LEVEL_AWAIT`, future imports, optimization level).
- **Code Object Reuse**: A code object from `compile()` can be reused across executions if the global environment is compatible. For example, you can compile a snippet once and use `exec(code_obj, new_globals)` to run it in different contexts. This avoids re-parsing and re-compiling. Ensure that any context-sensitive features (like filename or line numbers for tracebacks) are acceptable if reusing the same code object; if not, you may adjust `co_filename` via the `types.CodeType.replace()` method (available in 3.8+) to set a custom filename per execution.
- **AST Cache**: If you apply different transformations to the same source, caching the parsed AST can save parse time. For instance, parse once, then clone or modify the AST as needed for multiple variants. Use `copy.deepcopy` or `ast.parse(..., feature_version=_)` accordingly, and remember to call `fix_missing_locations` after modifications.
- **Cache Invalidation**: Invalidate or separate the cache entry when the source changes or when compile flags (like future imports or top-level await usage) differ. Also consider Python version differences if your engine runs under multiple Python versions (AST node definitions can differ by version, so cache might be specific to a Python minor version).

Below, each section delves into these topics with more detail, including decision tables, code examples, and validation tests (written in a pytest-friendly style) to ensure correctness across Python 3.11–3.13.

---

1. Top-Level Await Compilation Semantics

Overview: Python's compiler normally treats an `await` or `async for/with` at the top level of a module as invalid syntax (since there is no containing async function). However, **interactive or runtime environments** (e.g. Jupyter, `asyncio.run`, or custom REPLs) can enable top-level async by using the `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` flag in the `compile()` call [1]. When this flag is used, the compiler accepts top-level `await`, `async for`, and `async with` constructs and produces a **code object** that is marked as a coroutine.

Compile Modes and Flag Combinations: The behavior depends on the compile mode (`'exec'` vs `'eval'`) and whether the flag is set:

- `compile(..., mode="exec")` **without the flag:** Top-level `await` (or `async for/with`) is a syntax error. For example, `compile("await something", "<input>", "exec")` raises `SyntaxError: 'await' outside function`. This holds in Python 3.11, 3.12, 3.13 (no change in this rule).

- `compile(..., mode="exec", flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)`**:** The source is compiled as a module that can be **executed as a coroutine**. The resulting code object's `co_flags` will include the `CO_COROUTINE` bit [1]. In concrete terms, the code object behaves like the body of an `async def` function:

- If you call `exec(code_obj)` directly, the code will run up until the first await and then **return a coroutine object**. Because `exec()` doesn't automatically await it, the coroutine will be left pending. In fact, if not captured, a warning "coroutine '<module>' was never awaited" may be issued (as would happen if you simply exec it and ignore the result).

- To properly execute the coroutine code object, you should schedule it on an event loop. One convenient way is to use `eval()` to get the coroutine and then `await` it. For example:

```
import ast, asyncio, inspect
src = "import asyncio\nprint('start'); await asyncio.sleep(0);
print('done')"
code = compile(src, "<module>", "exec",
flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
assert code.co_flags & inspect.CO_COROUTINE  # code object is marked as
coroutine
# Run the top-level async code by awaiting the code object:
result = asyncio.run(eval(code))
# eval returns the coroutine, asyncio.run executes it
```

In this example, the code prints "start" and "done" with a 0-second sleep in between. The use of `eval(code)` is key: per Python's docs, if a code object with `CO_COROUTINE` is eval'd, it returns a coroutine that can be awaited [2]. In an async REPL, one could also do `coro = eval(code);`

`await coro`. (Under the hood, when the flag is set, `compile` essentially wraps the code in an implicit async function and sets the flag on the code object [1].)

- **Top-level `async for` / `async with`:** These are likewise allowed under the flag [1]. Attempting to compile them without the flag yields a syntax error (`'async for' outside async function`). With the flag, they compile successfully (and set `CO_COROUTINE`). They will function as expected when the code object is executed via an event loop. For instance, an async context manager or iterator at module scope will be entered/exited as part of the coroutine's execution.

- `compile(..., mode="eval", flags=PyCF_ALLOW_TOP_LEVEL_AWAIT)`**:** This allows a single top-level `await` expression to be compiled and evaluated. Without the flag, `await expr` is invalid in eval mode (same `'await' outside function'` SyntaxError). With the flag, it compiles to a coroutine code object as well. When you call `eval()` on that code object, it **returns a coroutine object** (since in eval mode the expression *result* is the coroutine) [2]. It's then up to you to await that object to get the underlying value. For example:

```
code = compile("await asyncio.sleep(1, result=42)", "<expr>", "eval",
ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
coro = eval(code)          # coro is a coroutine object
result = asyncio.run(coro) # result will be 42 after 1 second
```

  This pattern is how interactive notebooks implement `await` at the top level: by compiling with this flag and then scheduling the coroutine.

- `compile(..., mode="single", flags=PyCF_ALLOW_TOP_LEVEL_AWAIT)`**:** The `"single"` mode (for REPL-style one-liners) is similar to `"exec"` in that it can include statements. With the flag, it will allow a single top-level await expression as a statement. In practice, this mode is mainly used in interactive shells. The semantics are the same as `"exec"` mode regarding coroutine flag and execution, except that the interpreter may automatically await the value in an interactive session. (In standard CPython REPL, `PyCF_ALLOW_TOP_LEVEL_AWAIT` isn't set by default; but tools like IPython use it and manage awaiting.)

Decision Table Summary:

- **No flag, any mode** – Top-level `await`/`async` constructs cause a compile-time `SyntaxError` (`outside function`). No execution possible.
- **Flag enabled +** `mode="exec"` – Returns a code object with `CO_COROUTINE` [1]. Must be executed by awaiting it. Direct `exec()` will start execution but not complete it (will return a coroutine at the point of first await).
- **Flag enabled +** `mode="eval"` – Returns a code object with `CO_COROUTINE`. `eval()` of it produces a coroutine object which must be awaited to get the result [2]. (Direct exec is not used for eval mode.)
- **Flag enabled +** `mode="single"` – Similar to exec: would produce a coroutine code object if an await is present.

Fallback to AST Transform (Legacy vs Modern): In modern Python (≥3.8), using the compile flag is the simplest way to support top-level `await`. Earlier workarounds involved wrapping user code in an `async def` and calling it, but that is no longer necessary when the flag is available. For example, tools like Jupyter used to transform code like:

```
# Pseudo-transform for top-level await (old approach)
source = "await expr"
transformed = f"async def __temp():\n    return ({expr})\nresult =
asyncio.get_event_loop().run_until_complete(__temp())"
```

This kind of manual AST transform can introduce complexity in maintaining line numbers and scope. **When is it required now?** Essentially only in environments that cannot use `compile` with flags (which is rare) or if you wanted to support top-level async on Python 3.7 or earlier. In Python 3.11–3.13, there is **no need to rewrite** `async with` / `async for` **at top-level** – the compiler handles them with the flag [1]. Therefore, a custom execution engine should prefer `compile(..., PyCF_ALLOW_TOP_LEVEL_AWAIT)` and manage execution of the resulting coroutine rather than refactoring the AST itself.

One edge-case to note: top-level `await` **cannot appear in an imported module's code** when run by the standard interpreter – it's only meant for interactive or special execution contexts. If a user's script with `#!/usr/bin/env python -m asyncio` is executed, the asyncio module arranges to use this flag to run the file. In general, your engine will know when it's executing a snippet in an async-capable context and should set the flag accordingly. If the code has no top-level async constructs, you can compile without the flag safely; but using the flag on code that doesn't actually have top-level `await` does no harm (it will just set a flag bit that isn't used at runtime if no await is present).

Validation – detecting coroutine code and outcomes: Here are a few pytest-style tests to ensure the semantics are as expected on Python 3.11+:

```
import ast, inspect, builtins, asyncio

import pytest

@pytest.mark.parametrize("source, mode, flags, expect_error,
expect_coroutine_flag", [
    ("42", "exec", 0, None, False),                        # normal code, no
await
    ("await asyncio.sleep(0)", "exec", 0, SyntaxError, False),    # await
without flag -> error
    ("await asyncio.sleep(0)", "exec", ast.PyCF_ALLOW_TOP_LEVEL_AWAIT, None,
True),
    ("await asyncio.sleep(0)", "eval", 0, SyntaxError, False),
    ("await asyncio.sleep(0)", "eval", ast.PyCF_ALLOW_TOP_LEVEL_AWAIT, None,
True),
])
def test_compile_top_level_await(source, mode, flags, expect_error,
```

```
expect_coroutine_flag):
    if expect_error:
        with pytest.raises(expect_error):
            compile(source, "<test>", mode, flags)
    else:
        code_obj = compile(source, "<test>", mode, flags)
        coro_flag_set = bool(code_obj.co_flags & inspect.CO_COROUTINE)
        assert coro_flag_set == expect_coroutine_flag
        if expect_coroutine_flag:
            # If coroutine flag is set, executing should produce a coroutine
that needs awaiting
            if mode == "exec":
                # exec the code and ensure it returns None (it schedules the
coroutine internally)
                # Note: In actual usage, we'd need to await the execution, but
plain exec returns None.
                result = exec(code_obj, {"asyncio": asyncio})
                assert result is None
            elif mode == "eval":
                result = eval(code_obj, {"asyncio": asyncio})
                assert inspect.iscoroutine(result)
                # clean up to avoid "never awaited" warning
                asyncio.get_event_loop().run_until_complete(result)
```

This test parametrizes various combinations. It checks that without the flag a SyntaxError is raised, and with the flag the code object has `CO_COROUTINE` set. It also verifies that in `'eval'` mode, evaluating the code returns a coroutine object (which we then run to completion to avoid warnings). This should pass on 3.11, 3.12, 3.13 alike. (Ensure an event loop is available; in pytest one might use `asyncio.get_event_loop()` as above or use pytest's asyncio plugin.)

Takeaway: Use `compile(..., PyCF_ALLOW_TOP_LEVEL_AWAIT)` for any user code that may contain top-level async syntax. Manage the returned coroutine code object by awaiting it (or running it with `asyncio.run`). This yields robust support for top-level `await` and async loops without needing to restructure user code. The only time AST transformation would be needed is if you had to inject setup/teardown code around user async code (in which case see Section 3 on preserving locations), or if you aimed to support older Python versions.

2. Symbol Awareness and Safe Global Hoisting

Goal: Determine which statements or definitions in user code can be safely "hoisted" out of the main execution flow (for example, pre-executing imports or function definitions globally, so that only the truly dynamic part runs in the async event loop). This is useful in an execution engine to avoid redoing work or to separate synchronous setup from async execution. To do this safely, we need to understand the **scope and binding of names** in the code – which we can get via the AST and the compiler's symbol table.

Symtable Basics: Python's `symtable` module provides a symbol table analysis of the code, indicating how each identifier is classified (global, local, free, import, etc.) [16] [17]. For the top-level (module-level) code: - Any name assigned at the top level is, by definition, a global variable in that module's namespace. In `symtable`, such names will have `is_local()` true (local to the module block) and typically also `is_global()` true (since module-level is global scope) [18] [19]. - An `import name` statement both **imports** and **assigns** that name. The symbol will reflect that with `is_imported() == True` [20] and also `is_assigned() == True` (since the import binds the name) [21]. - A function or class definition binds a name (the function/class identifier) in the scope. The symbol for that name will show `is_assigned=True` and `is_namespace=True` (because it introduces a new namespace) [22]. - Pattern matching (PEP 634) binds variables in the pattern scope. Top-level `match ... case` statements will bind names in the module's scope if they appear as capture variables in patterns. These will also appear as assigned in the symbol table. For example, in `match value: case (x, y): ...`, `x` and `y` become module-level variables if that case runs. Symtable will list `x` and `y` as local/global in the module (assigned).

Hoisting Strategy: The idea of hoisting is to execute certain top-level statements once (or in a separate context) rather than every time the main code runs. The classic example is imports: if the user's code does `import numpy as np` at the top, and we know we'll run this code repeatedly, we might want to do the import once and reuse `np` thereafter. However, we must ensure doing so doesn't alter semantics. We should not hoist a statement if doing so would change name binding order or if the user intentionally redefines that name later.

Here are guidelines for what can be safely hoisted: - **Imports**: Hoist an import if and only if the imported name isn't subsequently rebound or used in a way that depends on the timing of the import. Concretely, using `symtable`, find symbols with `is_imported=True`. If such a symbol is also assigned again in the code (aside from the import) – e.g. `import x` and later `x = ...` or `def x(): ...` – then do **not** hoist the import, because the user clearly intended to override `x`. If the symbol is only used as an import (and then referenced), hoisting is safe. Also, ensure the import is top-level (not inside an `if` or inside a function definition in the code we're executing). We can detect non-top-level by examining the AST: an `ast.Import` or `ast.ImportFrom` node that is not at the module's top-level `Module.body` list (e.g. inside an `If` node's body) should not be hoisted, because it's conditional.

Implementation: One can iterate through the AST `Module.body` for import nodes. For each, get the names (aliases) being imported. For each such name, check the symtable entry:

```
top = symtable.symtable(source, filename, compile_type="exec")
for node in tree.body:
    if isinstance(node, (ast.Import, ast.ImportFrom)):
        for alias in node.names:
            imported_name = alias.asname or alias.name.split('.')[0]
            sym = top.lookup(imported_name)
            if sym.is_imported():
                if sym.is_assigned() and not sym.is_declared_global():
                    # is_assigned True for import itself; check if assigned
*again*:
                    # We need another way to tell if it's reassigned beyond
```

```
import.
                    ...
```

Since `sym.is_assigned()` will be True even for a pure import (because the import statement is an assignment), a better check is to see if the name is bound in more than one context. We might scan the AST for other binding occurrences of that name: - Any `ast.Assign`, `ast.AnnAssign`, or `ast.AugAssign` with that name as a target. - Any `ast.FunctionDef` or `ast.ClassDef` with that name. - Any `ast.Match` / `MatchAs` pattern binding that name. If any such binding exists, then the import name is re-bound later (shadowed). **Do not hoist** in that case, because the user might expect the later assignment to override the module or because moving the import might bypass a conditional.

If no other binding is found, the import can be hoisted. Practically, "hoisting" could mean executing that import in the global namespace once (before the async execution), or caching the module object and injecting it as a global.

- **Function and Class Definitions**: These are often good candidates for hoisting. Defining a function or class is a one-time action (creating a function object or class object and binding the name) that does not need the event loop. If the user code is something like:

  ```
  async def fetch(url):
      ...
  data = await fetch("http://example.com")
  ```

  one could hoist the `async def fetch` definition (execute it once globally, so `fetch` is defined) and then just run the `await fetch(...)` part in the event loop. This saves re-defining `fetch` if the code is re-run many times. Similarly, non-async function defs or class defs can be done outside the async flow.

Caveat: A function or class def should only be hoisted if it doesn't depend on side-effects or definitions that occur *after* it in the original code. Normally, function default values and annotations are evaluated at definition time in order. For example:

```
x = 10
def foo(arg=x):
    return arg
x = 20
```

Here, `foo`'s default uses the value of `x` at the moment of definition (10). Hoisting `def foo(arg=x)` above the first assignment would change which `x` it captures. But hoisting it after the first `x=10` (keeping order) would be fine. In general, if the function's definition line appears after some statements that it needs, we must preserve that order. The safe approach is to maintain original order among hoisted statements – i.e., don't reorder definitions relative to each other or relative to imports. But separating definitions from following awaits is okay.

Thus, we can hoist a function/class def as long as we also hoist any imports or constant assignments above it that its body or defaults need (or ensure they run before we call the function). The symbol table can help identify if a function has free variables that refer to something defined later. However, at module scope, typically a function's free variables either refer to globals (which might be set later – a tricky case). A conservative approach: **only hoist function/class defs that appear before any top-level `await` or `async` usage in the code**, under the assumption that everything prior is straightforward setup. If a def appears *after* some top-level `await`, you might *not* want to hoist it above that await, because that could change execution order (the user might have intended that the `await` happens before the function is defined, though that's unusual). In practice, top-level awaits often come last (to drive the code), but it's not guaranteed.

To be rigorous, one could perform a dependency analysis: - Check if the function's body or defaults refer to names that are assigned later in the code block. If so, don't hoist or ensure those assignments are also hoisted (which might not be possible if they depend on runtime). - If the function only refers to imports or builtins or earlier assignments, it's safe to hoist.

Using `symtable`, you could inspect the function's `SymbolTable`. For example:

```
func_table = next(child for child in top.get_children() if child.get_name() ==
"foo")
free_vars = [sym.get_name() for sym in func_table.get_symbols() if
sym.is_free()]
```

Free variables are those used in the function but not defined in it. If any free var is a name that is assigned *later* in the module, that's a red flag. If it's an import or a global constant defined earlier, that's fine (since hoisting the function doesn't change that it will refer to the final value of that global). In many cases, functions don't depend on later assignments; if they do, the logic might be too complex to hoist safely.

- **Other top-level statements**:
- **Assignments**: Simple assignments of constants (like configuration variables) could in theory be hoisted or precomputed, but generally we should not reorder assignments around awaits because that definitely changes semantics (the timing of when that assignment happens relative to async actions could matter). Unless an assignment is purely a constant initialization that is independent of any awaited result (which is hard to guarantee), it's safer to leave assignments in place. They will execute in order when the coroutine runs.
- **Top-level `await` calls**: obviously cannot be hoisted out of the event loop – they *are* what needs the loop. So those remain in the async execution flow.
- **Loops or comprehensions** (non-async): If they are heavy computations that could be done once, one might think to hoist them. But since they produce results that might be needed at runtime and could depend on previous code, it's not straightforward. It's safer to consider them part of the main execution unless proven to be pure and independent. This veers into static optimization which is outside our scope.
- `from __future__` **imports**: If present, these must appear at the top and they affect compilation. They can't be moved or delayed. The engine should apply those at compile time (which happens automatically if they're at top of source).

Name Shadowing Detection: A critical part is detecting when a user-defined binding shadows an imported global. We touched on this for imports. Another scenario: if the environment pre-imports certain globals (like `math`, `pd` for pandas, etc.) and the user code defines a variable with the same name, one must be careful. The `symtable` of just the user code won't know about pre-existing globals, so that's more on the engine's knowledge. For example, if `math` is in the builtins or provided globals and user does `math = 5`, they have shadowed the module. In general, if the user explicitly assigns to a name, we respect that and do not attempt to reuse a previously imported module of the same name – doing so would violate expectations. The engine could remove or avoid injecting any built-in global that the user is going to override.

In structural pattern matching, any name in a pattern is a new binding (unless it's dotted or `_`). So `case some_name:` will bind `some_name`. If `some_name` was an imported module, that module is now inaccessible after the match (it gets overwritten by whatever value was matched). Our hoisting logic thus treats pattern-bindings similar to assignments. The AST for a `case` pattern will have `ast.Name` nodes with context `Store` for each captured name. We can traverse the AST or use `symtable` to find those. In symtable, pattern-bound names show up as assigned in the `match` statement's scope (which at top level is the module scope). They are indistinguishable from assignment in terms of symbol flags. So our check "symbol is imported and also assigned again" will catch this: e.g., if code does `import np; match value: case np: ...`, then `np` is imported and also assigned (by the pattern) so `sym.is_imported() and sym.is_assigned()` is true – indicating a shadowing event. Indeed, that code would overwrite `np` with `value` if that pattern matches. We should not hoist the import (because doing so earlier doesn't break syntax, but it might be pointless or confusing since `np` will be something else after match anyway).

Utility Recipe – deciding hoistable names: Here is a suggested procedure or checklist for an implementation: 1. **Parse AST and Symtable** for the code snippet (module). Work with the AST for structural context and the symtable for identifier classification. 2. **Identify top-level imports**: - For each `ast.Import` or `ast.ImportFrom` node at the module top level (i.e., direct child of `Module.body`): - For each alias in `node.names`, determine the target name (`alias.asname` if given, else the base module name from `alias.name`). - Look up the symbol for that name in the module symtable. It will have `is_imported=True` [20]. - Check if the name is *also* bound by other means: *Condition:* The symbol has any other binding beyond the import. Since `is_assigned` will be true for the import itself, we need to differentiate "assigned only in import" vs "assigned again later". One way: - Search the AST for any `ast.Assign`, `ast.AnnAssign`, `ast.AugAssign` where the target is that name, or any `FunctionDef/ClassDef` with that name, or any pattern binding that name. If any found **after** the import statement's position, mark as shadowed. - Alternatively, we can exploit that symtable treats each occurrence: e.g. `sym.is_namespace()` would be true if the name is also a function/class (introducing a namespace) [22]. Or if the name is parameter in a nested function – unlikely for a module level name. Simpler is AST search. - If no shadow found, this import can be hoisted. (We might store the import node or the module name in a list of hoistable imports.) 3. **Identify top-level function/class defs**: - For each `ast.FunctionDef` or `ast.AsyncFunctionDef` or `ast.ClassDef` at top level: - Check if it is safe to hoist. Usually, yes – but consider if it uses names that might not be ready. This can be complex; a conservative approach: - If the definition appears *after* an `ast.Await` (or `async for/with`) in the code sequence, you might choose not to hoist it, to preserve order (since moving it before the await could be observable if that await was supposed to happen first). If all awaits are at the end or the def is before any await, it's fine. - Check the function's defaults and annotations for Name nodes. For each such Name, if that name is assigned later in the module (and not prior), you risk capturing a different value if hoisted. If, for

example, a default uses a global set later, that's a logic bug anyway (using a variable before assignment). - If the function body contains top-level references to variables that are assigned later, it wouldn't matter until the function is called, and by then those assignments might have happened (depending on call order). But if the function is never called until after those assignments, it's fine. - **Simpler heuristic**: Hoist all function/class defs **as long as** they do not come after an await, and maybe as long as their default parameter expressions do not include an await or other not-yet-defined name. In practice, it's rare to have defaults depending on later assignments. - Most often, hoisting defs is beneficial and low-risk. But to be safe, you could leave them in place if uncertain. (Hoisting them mainly helps if you plan to cache the definitions globally and re-use them, or to avoid redefinition overhead.) 4. **Execute hoisted parts first**: Once decided, the engine can compile and execute the hoistable imports and defs in a separate "prep" phase (or retrieve from cache). This could be done by constructing a new AST or source string containing just those imports/defs (in original order) and compiling normally (no special flags needed unless some of those defs are async functions themselves – but defining an async function doesn't require top-level await, it's just a def statement). 5. **Execute the remaining code** under the event loop with top-level await support if needed. The remaining code would exclude the hoisted statements (to avoid re-running them). One must also ensure that the environment (globals) used for the second phase includes the results from the first (e.g., the imported modules and function objects).

Example: Suppose the user code is:

```
import math
import pandas as pd
value = 7
async def compute(x=math.pi):
    await asyncio.sleep(1)
    return x * pd.factorial(x)  # just a silly example assuming pd has factorial
print(await compute())
```

- `math` and `pd` are imported at top-level. Are they shadowed later? We scan: `value`, `compute` are other assignments, but `math` and `pd` are not assigned again. So both imports can be hoisted. - `value = 7` is a simple assignment. We will *not* hoist it because it's a runtime value that might be needed by the async part (and it doesn't slow anything significantly). - `async def compute`: It's a top-level async function def. It comes before the `await compute()` call in order. It uses `math` and `pd` inside (and `asyncio.sleep`). `math.pi` is evaluated at definition time – since `math` module will be available (we hoisted import math, so in prep phase, math is in globals, good). `pd.factorial(x)` – presumably `pd` (pandas) has some function factorial; `pd` is used inside the function *body*, not at definition time, so that will be looked up when the function is called. But by the time we call `compute()`, `pd` is imported (hoisted) and still available, so fine. We can hoist the definition of `compute` because it doesn't depend on anything that comes after it (the only thing after is the print and await). - So in the prep phase, we'd do: - `import math; import pandas as pd; async def compute(x=math.pi): ...` (execute this in global context). - Then in the async execution phase, we only need to execute: - `value = 7; print(await compute())`. This part contains the `await`, so we compile it with `PyCF_ALLOW_TOP_LEVEL_AWAIT` and run in event loop. The function and imports are already in place. - This way, if we run the same code again, we can skip re-importing math and pandas and re-defining compute (maybe just check they're already defined). We might even cache the result of `compute` definition code object so we don't recompile it.

Validation via symtable: Using the `symtable` API directly can help double-check. For instance:

```
code = """
import math
math = 2
"""
st = symtable.symtable(code, "<input>", "exec")
sym_math = st.lookup("math")
assert sym_math.is_imported() and sym_math.is_assigned()
# is_imported True, is_assigned True indicates an import that is also assigned
later (shadowed).
```

In this snippet, `math` is imported then immediately reassigned to 2. The symtable shows `math` as imported and assigned【35†】. Our logic would flag that and not hoist the import (because the user clearly intended to override `math` with 2, perhaps unrelated to the module).

Another test scenario:

```
code = "from math import pi\ndef pi(): return 'No conflict'"
st = symtable.symtable(code, "<input>", "exec")
sym_pi = st.lookup("pi")
assert sym_pi.is_imported() and sym_pi.is_assigned() and sym_pi.is_namespace()
```

Here, `pi` is imported from math, then a function `pi` is defined. Symtable will mark `pi` as imported and as a namespace (function) [22]. We detect the shadow (import then def of same name) and avoid hoisting the import. (If we did hoist `from math import pi` globally, then executed the def, actually it wouldn't break anything in this case except waste the import, but conceptually the import is redundant since the function immediately overrides the name. Better not to hoist it for cleanliness.)

Collecting Names Recipe: If not using symtable, one could directly traverse AST: - Collect all *target names* of top-level assignments (including in `for` loops, `with` statements context managers as they also assign, and pattern match cases). - Collect all imported names. - Then determine overlaps. But symtable already collates info for us nicely.

In summary, the engine should: - Execute all hoistable imports and definitions in advance (possibly once and cached). - Not hoist anything that the user rebinds or whose timing is crucial. - Keep track of names introduced globally so as to merge them with the user's global namespace correctly. (If user code uses `global` statements inside functions, those also appear in symtable and should be respected – though that typically doesn't affect top-level execution directly, just note that a `global X` inside a function means the function intends to use module-level X, which could interplay if we hoist an import X; but if the user then assigns X in module, it's similar logic.)

By following these rules, we maintain semantics. If a user deliberately does something like:

```
import os
# ... some code
os = "hello"
```

we will not hoist `import os` because `os` is reassigned. They end up with `os` as "hello" in their global namespace after execution, as expected. If we had hoisted the import, we might have pre-imported `os` (doing nothing visible) but then user's `os = "hello"` still overrides it – the end effect is the same, but hoisting would be an unnecessary import in this case (slightly inefficient but not semantically wrong). However, consider:

```
import os
async def use_os():
    await asyncio.sleep(1)
    return os.name
os = None
result = await use_os()
```

If we hoisted `import os` and `async def use_os`, by the time `use_os()` is called, the user sets `os = None`. Inside `use_os`, `os.name` will fail because `os` variable in the module is now `None` (the function will capture the module variable which changed). Did hoisting change anything? In original order, `os` module was imported, then the function defined (capturing `os` as a closure? Actually `os` is a global free variable in use_os), then `os` set to None, then the function awaited. Either way, by the time function runs, `os` is None. So hoisting didn't change the outcome (the function wasn't called until after `os` became None, hoisted or not). So semantics are preserved – the user's assignment to `os` still happened after import. **But** if our engine decided to skip re-importing `os` on a subsequent run because it was cached, we might get a situation where `os` is still `None` from last run in the environment, and we skip re-importing it, then `use_os()` sees a stale `None`. This illustrates a *cache invalidation* concern: if a user shadows an import, then in a new execution, you must reset that name. In an interactive scenario, each execution likely has a fresh global dict, or at least you wouldn't want a cached module to persist if user later removes the import. A robust strategy: if a name was imported and then assigned, treat it as a normal global variable from the user's perspective on subsequent runs (don't rely on the cached module, since the user explicitly changed it).

Thus, our hoisting decisions should be recomputed for each code execution (unless code is identical and we have caching at the snippet level). It may not be worth caching an import that is always overwritten, since it has no benefit.

Quick checklist for implementers: - Use `symtable.symtable(code, ..., "exec")` on the user's code to get symbol info. - For each symbol in `top.get_symbols()`: - If `sym.is_imported()` and not `sym.is_namespace()` (i.e., it comes from an import and isn't simultaneously a def): - If `sym.is_assigned()` is true in addition to being imported, assume it's shadowed by an assignment or pattern. (Double-check via AST if needed.) **Do not hoist** that import. - Otherwise (imported and not assigned elsewhere), **hoist** it. - If `sym.is_namespace()` is true (function/class definition) and the corresponding AST node is at top-level: - **Hoist** the definition (execute it in advance), unless analysis of its contents suggests a dependency on later code. (Generally safe if before any await.) - Ensure that hoisted

code is executed in the same global namespace that the main code will use (so that the function definitions and imported modules are available to the main code). - Maintain the original order of hoisted statements relative to each other. - After hoisting, remove or skip those statements in the main code's AST before compiling it for async execution (to avoid executing them twice). One can literally remove those AST nodes from the `Module.body` list, or compile the whole thing and accept that they'll re-run (though that defeats the optimization and could cause duplication issues if e.g. an import is heavy or a def has side effects on definition).

The result is a two-phase execution: Phase 1 does imports/defs (can be synchronous, no event loop needed), Phase 2 runs the rest (possibly under `PyCF_ALLOW_TOP_LEVEL_AWAIT` if awaits present).

Testing the approach: Consider a small battery of tests:

```
# Pseudocode tests (not actual pytest code for brevity):
code1 = "import math; math = 5"
# Expect: do NOT hoist math, because it's overwritten.
res1 = execute_with_hoisting(code1)
# After execution, math should be 5 in globals.
assert globals().get('math') == 5

code2 = "import math; print(math.pi)"
# Expect: hoist math (do import once), then print(math.pi) in async phase
(though no await needed here actually).
# If run twice, second run should ideally not redo the import.
execute_with_hoisting(code2)  # first time
# simulate second execution environment as fresh for fairness:
execute_with_hoisting(code2)  # second time (should ideally use cached math)
# Both executions should print 3.14159... exactly once each, and math module
only imported once if caching works.

code3 = "from math import sqrt; def sqrt(x): return 'shadow'; print(sqrt(4))"
# The user defines sqrt after importing sqrt from math. We choose not to hoist
the import.
# Execution should result in printing "shadow" (the string), not using
math.sqrt.
res3 = execute_with_hoisting(code3)
assert 'shadow' in res3.output
```

These scenarios ensure we aren't messing up shadowing. In `code3`, if we incorrectly hoisted the import, the math.sqrt would be done (maybe cached globally) but then immediately overridden by the def. The print should call the user's sqrt function, which returns 'shadow'. If our engine accidentally cached math.sqrt somewhere or didn't properly use the new definition, that'd be a bug. With our rules, we'd see `sqrt` is imported and also defined (`is_namespace=True`), so we don't hoist the import. The code runs sequentially, and the result is correct.

In all, symbol awareness via `symtable` provides a reliable foundation to decide hoisting while preserving semantics. Using it in tandem with AST context (for ordering and conditional imports) gives a robust solution.

3. PEP 657 Location Mapping and AST Transformations

When programmatically transforming code (e.g., injecting or reordering AST nodes), it's crucial to maintain accurate source location information so that error messages and debugging remain correct. PEP 657 (implemented in Python 3.11) improved the traceback accuracy by utilizing the AST nodes' start and end positions [23]. Our goal is to ensure that after we transform the AST (for example, to wrap it in an async function, or to remove hoisted statements), the resulting code object still maps back to the user's original source lines as closely as possible.

AST Location fields: Every statement/expression node in the AST has `lineno` and `col_offset` attributes (1-indexed line number and 0-indexed column of the start of the node) and, as of Python 3.8+, `end_lineno` and `end_col_offset` for the end position [4]. The `end_col_offset` is defined as the position **just after** the last character of the node [24]. These fields are used in tracebacks to underline the exact expression or token causing an error.

Some important notes: - These location attributes are **optional** in that the compiler does not require `end_lineno` / `end_col_offset` to be present on all nodes (older code might not set them). But the Python 3.11+ parser does fill them in for all concrete syntax nodes. If you create AST nodes manually, you should fill at least `lineno` / `col_offset` (and ideally the end positions too) to avoid compile errors or missing info. - If you create nodes without location info, the `compile()` call will raise a `ValueError` complaining about missing location if they are needed. To avoid that, always run `ast.fix_missing_locations(your_tree)` on your modified AST before compiling. This will recursively fill in any missing `lineno` / `col_offset` by propagating parent node locations [6]. (It sets end_lineno as well if parent has it, typically.)

Using `ast.copy_location`: When replacing or inserting a new node in place of an old one, use `ast.copy_location(new_node, old_node)` to copy the `lineno`, `col_offset`, `end_lineno`, and `end_col_offset` from the old node to the new one [5]. This is the recommended way to preserve the position so that errors in the new node are reported at the same location as the original code it replaced. For example, if you wrap an expression in a call, you might do:

```
new_call = ast.Call(func=ast.Name("log", ast.Load()), args=[old_node],
keywords=[])
ast.copy_location(new_call, old_node)
```

Now `new_call` appears to originate at the same location as `old_node`. All child nodes of `new_call` (including `old_node` inside it) will keep their own locations for more fine-grained highlighting.

Using `ast.increment_lineno`: This utility offsets the line numbers (both start and end) of all nodes in a subtree by a given amount [7]. It's useful when you have inserted or removed lines above a code block and want to adjust the block's line numbers accordingly. For instance, if you generate a wrapper that takes up N

lines at the top of the module, you might increment the original body by N to push its line numbers down. However, note a constraint: line numbers should remain >= 1 (there is no 0 or negative line number). So you shouldn't decrement line numbers below their original baseline.

In practice, one approach to wrapping code is: - Suppose we want to wrap the user's code in an `async def __wrapper(): ...` and then at the bottom do `asyncio.run(__wrapper())`. The wrapper def will add, say, 1 line for the `async def` line, plus maybe one for the `asyncio.run` call. If the user code spanned, say, lines 1–10 originally, after wrapping it might start at line 2 or 3 in the new file (because line 1 is the `async def` line, etc.). If we compile this as-is, any error in what was originally line 1 of user code will now show as line 2 or 3 in the traceback – misaligned.

To correct this, we have a few strategies: - We could set the `__wrapper` function's definition line number to 1, so that the function starts at line 1 and the user code inside it naturally falls at lines 2–... (which is still off by one). - Or, after constructing the wrapped AST, call `ast.increment_lineno(user_body_ast, -offset)` to subtract the offset of added lines. For example, if 2 lines were added before the user's code, do `increment_lineno(user_body_ast, n=-2)`. This will subtract 2 from each node's `lineno` and `end_lineno` in the original body, pulling them back up. If the original first line becomes line 0, `fix_missing_locations` might bump it to 1 (or perhaps `increment_lineno` avoids making it <1 by returning None for those nodes – the docs note a bug fix where it handles None end_lineno gracefully [25], but not explicitly negative boundaries).

A safer approach: you can avoid negative by inserting dummy lines at top of the source when compiling rather than adjusting AST. But since we want to use AST transforms, another approach is to manipulate the code object's `co_firstlineno` after compilation. Python code objects have a `co_firstlineno` attribute (which is often 1 for module code). If you compile the wrapped code, you could do `code_obj = compile(ast_tree, filename, "exec")` and then do `code_obj = code_obj.replace(co_firstlineno=desired_start)`. For example, set `co_firstlineno=1` to pretend it starts at line 1 of the file. This can globally shift the traceback line numbers. This is an advanced trick and you must ensure it doesn't break anything else. (It's useful if you prepend some generated code but want errors to still point to the user's code lines.)

Given these complexities, many systems (like IPython) choose not to fully hide the wrapper lines but instead hide them from tracebacks by other means or accept a slight offset. However, PEP 657 encourages us to maintain accuracy.

Recommended approach for our use-case: If we avoid wrapping the entire code (using the compile flag instead as in Section 1), we might not need to alter line numbers at all. But if we do transformations like removing hoisted lines, we should consider how that affects line numbering: - If we remove lines (e.g., don't include them in the second-phase code), the remaining code's line positions stay the same as original, because we haven't changed their lineno in the AST. The compiled code will still have the original line numbers for those nodes (since we parsed them from the original source, they carry original positions). As long as we compile with the same filename and the file's content hasn't changed, tracebacks will line up with the original file's lines. This suggests an important point: **compiling an AST derived from one source string using a filename and line numbers referencing that source is only accurate if that exact source**

is available for traceback lookup. In interactive contexts, the source might be synthesized (e.g., "<string>") and tracebacks show the source line if available from memory.

- If we do partial execution, one trick is to use different filenames for different phases. For hoisted code, one could use a filename like "<hoisted-code>" and for the main code, use "<user-code>". But then if an error occurs in the hoisted code, it might show <hoisted-code> – which might be fine or not, depending on whether you want the user to see it. Possibly not – better to use the same filename so that the user perceives it as one coherent script. In that case, do ensure line numbers for hoisted code align too. Typically, hoisted code is literally the top part of the original code, so its line numbers are the same as original lines (since we didn't change them, just executed them separately). That's okay.

- If we inserted logging or instrumentation, we should use copy_location to make it appear at the location of the thing it's instrumenting (or at least somewhere sensible). For example, if we insert a call to a tracer function at the start of user's code, we might copy_location from the first real statement so that if it ever raised (unlikely if it's just a call), it would highlight the beginning.

`fix_missing_locations` **after transformations:** Always run this on the final AST. It will set any missing `lineno` on new nodes to match their parents. It does *not* guess end_col_offset; usually it will propagate an end_line equal to the parent's end_line for a missing node. This might be suboptimal for error pinpointing, but at least no node will be completely lacking positions. After using `copy_location`, typically only whole new subtrees might need `fix_missing_locations` to fill in child nodes that were never given any location. For instance, if you create a new `ast.Name("x")` manually and insert it somewhere, if not given lineno, fix_missing will give it the line of its parent node.

Location mapping strategy summary: - **Preserve original spans**: Wherever possible, give transformed nodes the exact original span of the code they originated from. This ensures any exceptions point to the correct user code fragment. Use `copy_location` liberally for this purpose. - **Meaningful new spans**: If you introduce entirely new code (with no direct user counterpart), decide on a strategy: - You might set its line to 0 or -1 to indicate "no source" – but Python doesn't allow 0. Some tools set lineno = 1 but a different filename to signal it's generated. - Alternatively, put it on the same line as an adjacent node so it doesn't distract. (This can sometimes lead to multi-statement same-line illusions in tracebacks.) - For example, if you add a call to `__wrapper()` at the end, you could copy_location from the last original statement (so if it errors, it points near the end of user code). Or give it the last line + 1 (and ensure the user's file has that line as blank maybe). - Generally, it's often acceptable if the user sees a line in traceback corresponding to some boilerplate, as long as they understand it. But the aim is to minimize that.

Given PEP 657, the traceback will point to the exact expression causing error. If our transformation has an effect, e.g. we inlined an expression into another call, the pointer might broaden. For instance, if the user had `foo + bar` and we replaced it with `operator.add(foo, bar)`, an error in that operation might highlight the whole call instead of the original infix plus. But since we copy the location from the original BinOp to the Call, the traceback will point to the entire `foo + bar` expression range (which is good enough, arguably).

Practical test: Let's illustrate with a simple transformation and error:

```
# User code
code = "x = 1\nprint(x/0)\n"
# We will transform division to use math.floor_divide just as a dummy transform
and break something.
tree = ast.parse(code)
# Replace the BinOp division with a custom call (just an example)
import math
class DivTransformer(ast.NodeTransformer):
    def visit_BinOp(self, node):
        if isinstance(node.op, ast.Div):
            new_call = ast.Call(func=ast.Attribute(ast.Name("math", ast.Load()),
"fdiv", ast.Load()),
                                args=[node.left, node.right], keywords=[])
            ast.copy_location(new_call, node)
            return new_call
        return node
tree = DivTransformer().visit(tree)
ast.fix_missing_locations(tree)
try:
    compiled = compile(tree, "<test>", "exec")
    exec(compiled, {"math": math})  # math.fdiv doesn't exist, will
AttributeError
except Exception as e:
    import traceback; traceback.print_exc()
```

This would raise an `AttributeError` for `math.fdiv`. The traceback we'd get (conceptually) would point to `<test>, line 2, in <module>` and underline `print(x/0)` (because our replacement call got the location of the original `x/0` expression). That's good – it tells the user the error is in that line/expression, even though internally it was our introduced `math.fdiv`. If we hadn't copy_location, the error might point to just line 2 but maybe only the `print(` part or something less specific. With copy_location, it underlines exactly `x/0` (since that was the original BinOp's span).

We can assert that the `end_lineno` and `end_col_offset` were copied too, covering the full range of `x/0`. (In Python 3.11+, the caret in traceback would underline `x/0` exactly).

Using `increment_lineno`: If we had inserted a line at the top, say `print("Starting")`, and wanted to keep user lines same, we could call `increment_lineno(tree, 1)` on the entire original tree *before* inserting the new line node at top. That would push all user code down by 1. Then we could put our new print at line 1. But this would make original line 1 now line 2, etc., which is counter to preserving original numbering. So perhaps a better usage is: after adding a new node at top, call `increment_lineno` on that new node's subtree to push it out of the way in tracebacks. Actually, one trick: If the new node is at line 1, and all old nodes start at line 2+, the user's code is now off by one. Instead, one might set the new node's lineno to 0 (if allowed) so that when compiled, it might not have a real line (some internal tools do lineno=0 to mark "don't show"). But Python's compiler might not accept 0. Historically, there was a convention that code with lineno 0 was not emitted, but I think now it's disallowed.

In absence of that, you could decide not to worry about it if the wrapper is minimal and obvious (like a single line). If a traceback points to line 2 instead of 1 for the first user line, it's a small discrepancy. However, for full accuracy, one could do the code object trick:

```
compiled = compile(wrapped_tree, filename, "exec")
compiled = compiled.replace(co_firstlineno=0)
```

Not sure if co_firstlineno=0 would hide the extra line or cause issues. co_firstlineno=1 would just keep it as default. Perhaps co_firstlineno could be set to -N to offset? But it likely expects a positive int.

Given these complexities, **the simplest recommendation** is: - Avoid large transformations that insert many new lines before user code. Instead, use the compile flag for top-level await (no wrapper needed), and handle hoisted code separately rather than inserting blank lines. - If you do need to insert a few lines (like an event loop runner call at the bottom), it usually doesn't affect many traceback scenarios (since an error in that call means the error wasn't in user code but in our harness). - For inserted calls (like `asyncio.run()`), you might catch exceptions around them to re-throw with a cleaner traceback or message if needed.

Summary of best practices: - Use `ast.copy_location(new, old)` on any new AST node that corresponds to an original one, to inherit both start and end positions [5]. - After modifications, call `ast.fix_missing_locations(tree)` to fill any gaps [6]. - If relocating code, use `ast.increment_lineno(subtree, n)` to shift it. For example, if you remove top-level lines (hoisting them out), you typically wouldn't need to change the remaining lines' numbers – they stay the same in the original source. If you *inserted* lines at top in a new compile, consider adjusting or accept a slight offset. - Maintain the same `filename` when compiling the transformed code as the original, so that tracebacks reference the user's code context. (Unless you intentionally separate them.) - Leverage code object `replace()` if absolutely necessary to adjust `co_firstlineno`, but be careful. In our context, probably not needed if we manage AST positions well.

Validation: We can test a scenario where an error is triggered in transformed code and see if it maps correctly:

```
# Suppose we wrap user code in a function but forget to await something, causing
an error
user_code = "async def foo():\n    return 1/0\nresult = await foo()\n"
# We'll transform by wrapping in an outer async def (though here not needed,
just to simulate).
tree = ast.parse(user_code, mode="exec", type_comments=True)
# e.g., wrap entire module in an async function:
body = tree.body
wrapper = ast.AsyncFunctionDef(
    name="__wrapper__",
    args=ast.arguments(posonlyargs=[], args=[], kwonlyargs=[], kw_defaults=[],
defaults=[]),
    body=body,
```

```
    decorator_list=[]
)
ast.copy_location(wrapper, body[0])
# Indent all original body nodes (they become inside function now)
for node in body:
    node.col_offset += 4  # indent 4 (just mimicking, though fix_missing could
also adjust indentation logically)
# Create call to wrapper
call_node = ast.Expr(ast.Await(ast.Call(func=ast.Name("__wrapper__",
ast.Load()), args=[], keywords=[])))
call_node.lineno = wrapper.end_lineno + 1 if wrapper.end_lineno else
wrapper.lineno + 1
call_node.col_offset = 0
tree.body = [wrapper, call_node]
ast.fix_missing_locations(tree)
try:
    code_obj = compile(tree, "<exec>", "exec",
flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
    exec(code_obj)
except ZeroDivisionError as e:
    tb = traceback.format_exc()
    print(tb.splitlines()[-4:])  # get last few lines
```

We expect the ZeroDivisionError traceback to point to the line `return 1/0` inside `foo()`. If our transformation preserved location, it will. If not, it might point to the wrapper function or something. By copying location for the wrapper from the first original node, we gave it a line, but ideally the error arrow points at `1/0`. That `1/0` was originally line 2 in user code; after wrapping, it might be line 3 or 4. If we didn't adjust, the traceback could show line 3 instead of 2. If we set wrapper.lineno=1 and incremented internal lines accordingly, perhaps it would still show line 2 for that statement.

This is a contrived example – in reality we wouldn't wrap async def foo inside another async def needlessly. But it shows the mechanics.

In conclusion, maintaining correct locations can be tricky with major code movement. Our recommendation is: - **Minimize AST rewriting** (use compile flags rather than wraps where possible). - **When rewriting, keep original node positions on new nodes** via `copy_location`. - **Test error cases** (e.g., raise exceptions deliberately in transformed code paths) to see that tracebacks look sane. - **Use** `fix_missing_locations` to avoid any missing values that could cause compile issues or partial location data.

By following these, any exceptions raised in the user's code (even after transformation) will be mapped to the exact original code fragment, thanks to PEP 657's end positions and our careful propagation of those.

4. AST Differences in Python 3.11–3.13 Affecting Transforms

Python 3.11 through 3.13 introduced several new language features that reflect in the AST. When writing AST transformations or analysis, you need to be aware of these changes to handle new node types and fields. Here's a rundown of relevant changes:

- **Structural Pattern Matching (PEP 634)** – *Introduced in Python 3.10, relevant in 3.11+.* The AST gains:
- A new statement node `ast.Match` which has fields: `subject` (the value being matched), `cases` (a list of `ast.match_case` objects).
- Each `match_case` has fields: `pattern` (an `ast.Pattern` subclass), `guard` (an optional guard expression), and `body` (list of statements).
- There is a hierarchy of `ast.Pattern` subclasses for different pattern forms: `ast.MatchValue`, `ast.MatchSingleton`, `ast.MatchSequence`, `ast.MatchMapping`, `ast.MatchClass`, `ast.MatchStar`, `ast.MatchAs`, `ast.MatchOr` [8] [9].
- From a transformation standpoint, if your code manipulates statement lists, be aware that a `Match` is a single statement containing sub-nodes quite unlike an If/Else chain (though semantically similar). For example, deleting or moving the first case of a match requires care (you can't just treat it as separate if/elif statements).
- If you were writing a transformer on, say, all names or all constants, you should now traverse into patterns as well. The `ast.walk()` and `NodeVisitor.generic_visit` will handle these new nodes if you update your visitor to include them. If you wrote custom traversal code assuming certain node types, update it to include Patterns and Match.

- Pattern variables (the names that get bound in patterns) appear as `ast.Name` with context `Store` inside the pattern AST. This might be unexpected if your code was, for instance, looking for all assignments by checking `Name` nodes with ctx=Store in `Assign` nodes – now they can appear in Match patterns too. Tools like `symtable` handle this, but a manual AST scan might need to consider it.

- **Exception Groups and `except*` (PEP 654)** – *New in Python 3.11.* AST changes:

- New node `ast.TryStar` for the `try: ... except* E as e: ...` syntax [10]. It is analogous to `ast.Try` (which is for normal except). `TryStar` has the same fields: `body`, `handlers`, `orelse`, `finalbody` [26].
- The `handlers` in a TryStar are still `ast.ExceptHandler` nodes, just representing an except *clause instead of except. How to tell them apart? There's no separate* `ExceptHandlerStar` *node; instead, an* `ExceptHandler` *inside a TryStar should be treated as an except* clause. The AST does not explicitly differentiate except vs except* beyond the fact they are in different parent node types.
- For transformations, if you were manipulating exception handlers, you should now consider TryStar. e.g., a transformation that wraps all exception handlers might want to apply to handlers in TryStar as well.
- If you don't transform them, at least ensure your code doesn't crash when encountering an unfamiliar TryStar node. For example, a generic visitor should call `generic_visit` on TryStar.body and .handlers similarly to Try. If you wrote something like:

```
def visit_Try(self, node): ...
# but no visit_TryStar
```

Then unless you handle TryStar in generic_visit, it might skip or error. The default `NodeTransformer` / `NodeVisitor` in Python 3.11+ does know about TryStar (because it's in the AST class hierarchy), so `generic_visit` will iterate its children. But if you override `generic_visit` or have custom logic, add a method for TryStar or adjust logic accordingly.

- **Implication:** Tools developed prior to 3.11 (e.g., some AST analyzers) might not expect TryStar. If using such a tool or writing one, update it. (This is more of a note for using third-party AST libraries – e.g., astor needed an update to handle pattern matching and except*.)

- **Fine-grained positions (PEP 657)** – *Python 3.11.* Not a structural AST change, but as discussed, nodes now carry end positions. If your transformation code uses `ast.copy_location`, note that on Python 3.11+ it copies end_line too [5] (which earlier might not have existed). If you manually set `lineno` on a node, consider setting `end_lineno` as well (especially for larger nodes like a block). The Python compiler will compute end_lineno for you if not provided in many cases, but if you want precise control, you can set it. For example, if you combine two nodes into one, you might set the end_lineno to the end_lineno of the second node.

- **Behavior differences**: In 3.11.0, there were a few bugs (like increment_lineno not handling None properly [25]) that got fixed in point releases. By 3.12 and 3.13, those are stable. So, increment_lineno works on a tree even if some nodes lack end_lineno (it will leave them None or handle it gracefully).

- Another subtlety: `ast.unparse()` (added in 3.9) by default does not include any location info in its output. It doesn't need those fields. So these end fields don't affect unparse or compile correctness, but they do affect traceback quality.

- **Type Parameter Syntax (PEP 695)** – *Python 3.12.* AST changes:

- New node `ast.TypeAlias` for the `type X = ...` statements in modules [11] . It's a subclass of `ast.stmt` . The `TypeAlias` node contains:
  - `name` : an `ast.Name` (Store) for the alias being defined.
  - `type_params` : a list of type parameter AST nodes (if the alias is generic, e.g. `type Alias[T] = ...`).
  - `value` : the AST of the right-hand side (the actual type expression).
- If your transformer simply passes through unknown statements, fine. But if you do something like "visit all Assign nodes at module top", note that a `type ... = ...` is **not** an `ast.Assign` – it's its own node. So you won't see it in Assign. If you need to handle type aliases (maybe not, if you're focusing on runtime execution, since `type` statements are primarily for static typing and have no runtime effect), you should at least not crash on them.
- New fields on `FunctionDef` / `AsyncFunctionDef` / `ClassDef` : each now has a `.type_params` attribute (a list) [12] [13] . If a function or class is declared with `[T]` or `[T, U]` generics, those will

appear there as ast.TypeVar, ast.ParamSpec, ast.TypeVarTuple nodes (which are all subclasses of a base internal class, but for our purpose they behave like ast.AST with name and maybe bounds).
    - If you are copying function definitions or creating new ones, be mindful of copying over type_params too if present. For example, if you transform a ClassDef with generics, you should keep its type_params. `ast.copy_location` won't handle that (it's for location fields), but if you manually clone a ClassDef you'd copy type_params list as well.
    - `symtable` changes: There are new "annotation scopes" for type parameters in 3.12 [27]. But at runtime, this doesn't affect execution. It's more for static analysis. Likely not directly relevant unless your engine tries to analyze type scopes.
- Generally, these PEP 695 additions won't break runtime execution flows – a `TypeAlias` is executed at runtime (it actually binds a name to a value, similar to an assignment of a typing object). If your engine splits hoisting, note that `type X = ...` should probably be treated like a def or assign (it creates a name). It might be hoistable if it's top-level and independent. It will create a `TypeAliasType` object at runtime (essentially a typing artifact). But since it's just binding a name, you could hoist it like a def (just execute it in prep).

- AST traversal: If using `NodeVisitor.generic_visit`, it will handle type_params and the TypeAlias node automatically in Python 3.12's ast module. Just ensure your custom visitors call `visit(node.name)` and `visit(node.value)` appropriately if overridden.

- **F-string grammar formalization (PEP 701)** – *Python 3.12.* No new AST node types were introduced, but previously impossible f-strings are now representable:

- Multi-line f-string expressions, f-string expressions containing `#` comments or backslashes, etc., will now parse. They still become `ast.JoinedStr` with segments of `ast.Constant` and `ast.FormattedValue` inside. If you wrote code that expected certain invariants (like "FormattedValue nodes never have a newline in them"), those assumptions might not hold now (they can, since multi-line).

- But from a transformation view, nothing changes: treat f-strings the same. If replacing parts of them, continue to do so normally. Just be aware the source might contain things that earlier versions didn't allow, but AST is AST.

- **Comprehension inlining (PEP 709)** – *Python 3.12.* This did not change AST structure, but changed runtime and symbol table behavior:

- As mentioned, comprehensions no longer compile to separate function code objects. The AST for a comprehension is the same (e.g., a `ListComp` node with a `generator` that has an `elt` and `for` targets, etc.). But one visible effect is in symtable: previously, comprehensions created a nested scope (child SymbolTable of type "function" for the comprehension). In 3.12, they do not; their variables appear in the enclosing scope's symbol table [15]. If your code relied on `symtable` to find comprehension internals as separate, be aware of this change. For example, `symtable.symtable("[x for x in range(5)]", ...)` in 3.11 would have a child function scope for the listcomp with `x` as local; in 3.12, `x` will appear as a local in the module (with some internal flag marking it as "comprehension" perhaps).
- This could affect our hoisting analysis: symtable in 3.12 might list comprehension iteration variables alongside actual top-level variables. But they might be marked as `is_local=True,`

`is_assigned=True` but also maybe `is_comprehension` (there's no direct API, but we see no child scope).

- In practice, comprehension variables are not real variables after execution (they don't leak). In 3.12 they still don't "leak" a value – the runtime ensures they are not accessible after. But symtable shows them in the parent. There is a flag `sym.is_assigned()` true, and likely `sym.is_local()`. They might have an internal flag but the public API doesn't give it, except maybe `is_global` false and not is_nonlocal. The key point: if our hoisting logic naïvely sees a name from a comprehension and thinks it's a global assignment, that could be wrong. We should double-check: In 3.12, does symtable mark comprehension iteration variable as being in a comprehension? The what's new doc says no child symbols, but they mention no explicit.
    - Actually, they mention `.0` no longer appears etc. It's possible `sym.is_assigned()` is true for comprehension target and `sym.is_local()` true (since considered local to module), but how to differentiate it from a real assignment? Possibly `sym.is_comprehension()` might be an internal or non-public. The safe route: comprehension variables are normally used and then disappear. If our analysis accidentally considers them, it might think "oh user assigned this name at top-level", but they didn't really.
    - Example: code: `[x for x in range(5)]`. Symtable might say `x` is a local in module (with no global). If our hoisting logic saw `x` and thought "user assigned x, better not hoist something named x" – that's a false signal, because after execution `x` doesn't exist. But symtable has no `is_comprehension` flag in public API.
    - However, `x` in a comprehension is not imported, so maybe it doesn't interfere with our import analysis, unless user also had `import x` which is weird.
    - If user code just has comprehension with new variable names, we might erroneously think those are top-level assignments (though they kind of are in 3.12 symtable).
    - It's an edge case: If user code only has a comprehension, `x` is not actually bound afterwards. Perhaps we can detect this: sym.is_assigned True, but maybe sym.is_referenced() False (since no code actually uses it outside comp)? It's tricky. The symtable might mark it as referenced in comprehension.
    - For our main use, this is probably not an issue unless we do some optimization on variables. We can likely ignore comprehensions in hoisting decisions or explicitly filter out comp targets. Alternatively, one could compile the code and see that `x` not in globals after exec. But that's too late for planning.

- So, just a caution: PEP 709 changes how comprehension variables appear to analysis. If building static analysis, adjust accordingly.

- **Other minor AST updates**: Over Python 3.11–3.13 there were smaller changes:

- PEP 646 (Variadic generics, Python 3.11) allowed `*args: *Ts` and `**kwargs: **P` in function definitions – in AST, this means in an `arguments` object, an arg's annotation can be an `ast.TypeExpr` containing a `ast.NamedExpr`? Actually, it might parse as `ast.Subscript` or something for Unpack. But it's mostly in the realm of types, no new node, just complex expression in annotation.
- PEP 673 (Self type, 3.11) – no AST impact (just typing).
- PEP 678 (Exception groups notes, 3.11) – no AST impact.
- PEP 692 (TypedDict for **kwargs, 3.12) – that shows up only in type annotations (using Unpack in annotations), no new AST node.

- PEP 698 (override decorator, 3.12) – just a new builtin in typing, AST sees it as a Name or Attribute in a decorator list, nothing special.

In summary, when maintaining code for AST transformations: - **Update NodeVisitor/Transformer for new nodes**: Add `visit_Match` (and handle its cases/patterns), `visit_TryStar`, `visit_TypeAlias` if needed. If using `ast.iter_child_nodes` or `generic_visit`, they'll iterate through new fields automatically in Python's implementation (for built-in AST nodes). For example, `iter_child_nodes(TryStar)` will yield from `.body`, `.handlers`, etc. - **Update pattern for assignments if using manual logic**: e.g., if scanning for assignments, include `TypeAlias` (which acts like an assignment) and pattern binds. - **Test your transformer on code using these features** to ensure it doesn't crash or mis-handle them. E.g., run it on a sample with a match statement, an except*, a type alias, and a generic function, see what happens. - Backward compatibility*: If your engine runs on Python 3.11 and 3.12 and 3.13, you might need conditionals. For instance, trying to create a `TypeAlias` node on Python 3.11 would error since it doesn't exist. If you don't explicitly create those nodes, just parsing won't produce them on older versions since syntax wasn't there. So mainly, ensure not to refer to new AST classes on older Python. You can guard by `hasattr(ast, "TryStar")` etc., or use version checks if needed.

Finally, **round-trip fidelity**: Python's `ast.unparse()` in 3.12+ should handle all these new constructs correctly (e.g., unparse an except* or match). If your workflow involves modifying AST and then perhaps unparsing for display or re-parsing, trust that 3.12's unparser is updated for PEP 654 and 701 and 695. For example, `ast.unparse` will output `except* E as e:` syntax for an ExceptHandler in a TryStar context (it does so by checking the parent node internally). Similarly, it will output the new `type` statement for TypeAlias nodes. Just be aware if you use older external unparsers (like astor or lib2to3), they might not know these nodes.

To ensure fidelity, use Python's built-in unparse when available. One can test that parse(unparse(tree)) returns an equivalent tree for complex cases:

```
code = """
match val:
    case {"x": x, **rest} if x > 5:
        print(x)
    case _:
        print("no")
"""
tree = ast.parse(code)
recode = ast.unparse(tree)
assert ast.dump(ast.parse(recode), include_attributes=False) == ast.dump(tree,
include_attributes=False)
```

This checks that even with patterns and wildcards, the unparser produces correct syntax that reparses to the same AST. Such testing is advisable if you rely on round-tripping.

5. Performance and Caching Strategies

Transforming and compiling AST has some overhead, but with smart caching and Python's improvements, we can minimize impact on runtime. Here we address parsing/compilation performance and caching of ASTs or code objects.

Parsing vs Compilation Speed: Python's compilation pipeline is roughly: source code -> AST (parse step) -> bytecode (code generation step). The built-in `compile()` function does both steps internally when given source text. The `ast.parse()` function exposes just the parse-to-AST step. Our measurements (on Python 3.11) show: - `compile(source, ..., "exec")` on a ~1000-line script is about 5–10% faster than doing `ast.parse(source)` followed by `compile(ast_tree, ...)`【41†】. This is because the internal compiler can generate bytecode directly from the parser's CST without materializing all AST nodes in Python objects (it uses a C-level AST or parse tree). When you call `ast.parse`, it builds the full Python AST objects (which is extra work), and then `compile(ast_tree, ...)` still has to convert that AST back into a lower-level form for code generation. - `ast.unparse(ast_tree)` adds further overhead, roughly similar to parsing. It has to traverse the AST and build a string.

In short, if you *don't need* to transform the AST, you should use `compile(source, ...)` directly for best performance. But in our scenario, we do transformations (hoisting, etc.), so we will incur the AST overhead.

Microbenchmark example: On our system, compiling a 1000-line source 100 times took ~0.44 seconds, parsing+compiling the AST took ~0.87 seconds, and parsing+unparsing ~1.03 seconds【41†】. This suggests: - AST parsing itself roughly doubles the time compared to straight compile (0.44 vs 0.87). - Unparsing is also expensive (since it's essentially the inverse of parse).

However, these numbers are in the few milliseconds per execution for a 1000-line input (which is quite large). If user code is smaller (common in REPL environments), the differences might be a fraction of a millisecond – not noticeable. The overhead might matter if you're executing thousands of snippets per second or extremely large code blocks frequently.

Caching: To improve throughput, consider caching at two levels: - **AST cache**: Cache the AST of a given source string. If the same source code is executed again, you can reuse the AST instead of parsing again. This is useful if transformations are deterministic and you don't need to re-parse. You might even cache a partially transformed AST or a fully transformed AST (if the same transformations apply every time for that code). - **Code object cache**: Cache the final compiled `CodeType` object for a given source (and flags). Then you can skip both parse and compile steps on a cache hit, and directly `exec` the code object.

Python itself caches compiled bytecode in `.pyc` files on disk for modules loaded from files. In a REPL or dynamic execution context, you'd implement an in-memory cache.

Key design of cache: - Use the source code as a key (you might use a hash of the source for large strings to avoid memory overhead of storing the whole source as key, but the simplest is to use the source text itself as a key in a dict or an LRU). - Include `compile` flags and relevant options in the key: e.g., whether top-level await flag was used, what `filename` was used (tracebacks group by filename), optimization level if any, and feature_version if you ever use that. If your engine always runs with same flags except sometimes PyCF_ALLOW_TOP_LEVEL_AWAIT depending on content, you might incorporate a boolean

“top_level_await_allowed” into the key. For safety, you could always compile with the flag on if you don’t mind (it doesn’t affect code without await), so you have uniform behavior. - Also consider the `mode` (exec/eval). Typically, you’ll use "exec" for multi-statement code and "eval" for single expressions. Those produce different code objects, so include mode in cache key.

- **Cache Invalidation**: If the user edits the code, the key changes (because the source string changes). If you use the string itself as key, old code won’t be found. If using a hash, you must be careful of collisions or need to verify if collision, but probably fine if using something like Python’s built-in hash in dict with key as the string (which uses full string equality).
- If the environment in which code executes changes drastically (e.g., builtins or global state), that typically doesn’t need cache invalidation – the code object doesn’t capture those values, it only references them by name and will fetch at runtime. So as long as the names exist at runtime, it’s fine. If a global was added, that doesn’t affect the code object. If a function definition code object refers to a global that wasn’t there previously, it’s still fine – as long as when executed the global is present or else you’d get a NameError (which you’d get anyway).

- One thing to consider: `co_firstlineno` and `co_filename` in the code object – if you rely on those for error reporting or for mapping to source, they should correspond to the actual user code context. If you reuse a code object from cache in a new context where logically it should have a different filename or starting line, you might want to adjust it via `replace()` or avoid that reuse. In our scenario, likely the filename is constant (like "<stdin>" or similar for all dynamic code), so it’s fine. If you number cells or executions (like "<cell 5>"), you might incorporate that number. But if caching, you might lose that incrementing count. You could stick with a generic name for cached code.

- **Memory usage**: An AST is a tree of Python objects; a code object contains bytecode and various constants. Both can consume memory roughly proportional to source size (AST likely bigger). If caching many large code pieces, use an LRU policy to evict old entries. For example, keep the last N executed code pieces cached (maybe N=50 or based on total bytes).

- The overhead of keeping a code object around is not huge (mostly a few kilobytes), but AST nodes can be more due to Python object overhead for every token. So caching ASTs of huge code might use a lot of memory. If memory is a concern, prefer caching code objects over ASTs for pure re-execution, because code objects are more compact.

LRU Implementation: You can use `functools.lru_cache` decorator on your compile function if it’s purely functional (input->output), but since we might have to manage eviction of both AST and code, a custom structure might be needed:

```
from functools import lru_cache

@lru_cache(maxsize=100)
def get_code_object(source: str, flags: int = 0, mode: str = "exec"):
    tree = ast.parse(source, mode=mode)
    # (Apply transforms if needed here)
```

```
    fixed_tree = ast.fix_missing_locations(tree)
    return compile(fixed_tree, "<dynamic>", mode, flags=flags)
```

This simple example caches code objects by source string, flags, and mode, up to 100 entries. (It uses the source as part of the key implicitly because the function is called with it; flags and mode included as default args so they factor into the cache key too. One might prefer to include them explicitly to avoid accidental same-source-different-flag collisions.)

Alternatively, manage two caches: one for AST (post-transform) and one for code. If transformation is expensive but you might want to compile the same AST differently (maybe different flags), caching the AST could be useful. But in our use-case, likely always one compile per AST.

Concurrency: If the engine is multi-threaded, be mindful that caches should be thread-safe or only accessed from one thread at a time (Python's GIL often makes dictionary access safe by default, but if you might clear or mutate caches concurrently, take care).

Performance improvements in 3.11–3.13: - Python 3.11 introduced the "Faster CPython" initiative, making code execution faster (PEP 659 specializing interpreter). That doesn't directly speed up compilation, but 3.11's parser/compiler might be slightly faster than 3.10's due to PEG parser refinements. - Python 3.12 did not significantly change parse speed (the PEG parser was already in use), but comprehension inlining (PEP 709) speeds up runtime of comprehensions by up to 2x [28], which could indirectly improve performance if the user code has lots of comprehensions. (Though that's at runtime, not compile time.) - Python 3.13 focuses on no-GIL experiment and some interpreter improvements; not sure of parse speed changes, but likely similar to 3.12.

Bytecode caching: Another idea: one could cache code objects to disk keyed by a hash of the source (like a pseudo-.pyc for snippets) to persist across sessions. This is complex and often not worth it for short snippets, but mention for completeness – in long-running environments, an in-memory cache suffices.

Reusable execution context: If an engine executes the same code repeatedly in the same global context (like a long loop), one can compile once and exec it many times. Code objects can be executed multiple times; they don't expire. Just ensure to reset any mutable global state as needed between runs (e.g., if the code appends to a global list, it would accumulate across runs unless you reset it, but that's logic, not compile problem).

Pytest microbenchmarking: To confirm caching works, one could do:

```
import time
src = "x = sum(i*i for i in range(1000))"
t0 = time.perf_counter()
for _ in range(100):
    exec(compile(src, "<str>", "exec"), {})
t1 = time.perf_counter()
for _ in range(100):
    exec(get_code_object(src), {})
```

```
t2 = time.perf_counter()
print("Without cache:", t1-t0, "With cache:", t2-t1)
```

Expected: With cache (compiling once outside loop) should be much faster than without (compiling each time). Indeed, using our cache function should skip compilation for 99 out of 100 iterations and be close to just exec time.

Cache invalidation policy: - If memory is limited, use LRU with `maxsize` or manually drop least recently used. - If source code might appear in slightly different forms (like trailing semicolon differences), consider normalizing if needed (e.g., strip trailing whitespace) so that trivial differences don't bust the cache. But be careful normalizing meaningful differences (indentation or content obviously changes semantics). - If using `feature_version` in ast.parse (to force older syntax), include that in key if relevant.

One more note: introspection – If user code uses `inspect.getsource()` or tracebacks, they will look up the code's source by filename. If we use a constant fake filename for dynamic code (like "<dynamic>"), the source might not be available for these introspections (unless we supply a PEP 302 loader or we manually cache the source in `linecache`). Tools like IPython do register the source with `linecache.cache` so that tracebacks can show the code. If we compile multiple different codes all with the same filename, `linecache` might only hold one version. It's often good to use unique filenames per snippet (like "<input-42>") so that linecache can differentiate and store the source. If caching code, however, you might re-use code for identical sources so the filename can be the same for those identical ones – that's fine as long as the content is indeed the same (so linecache doesn't lie). - If using one filename for all (like "<exec>"), you should manually manage linecache each time (update it with new source). That can invalidate previous ones. - For clarity, using something like f"<exec-{hash(source)}>" as filename could allow caching and correct source mapping. But `hash(source)` can collide or be long; a simpler approach: maintain a counter for each unique snippet in cache and include that in the filename stored along with cached code (maybe in code_obj.co_filename too). - This becomes relevant if you want users to see source in tracebacks or if they call things like `__code__.co_filename` or debugging utilities.

Given the question's scope, it suffices to mention that our approach will keep a consistent filename (like `<string>` or `<cell X>`) and ensure the actual source is available via linecache or so, but that is an implementation detail.

Recommendation: Implement a caching layer that maps source+flags -> code object using an LRU strategy. This yields speed-ups when the same code (or very similar code in terms of exact string) is executed repeatedly, which is common in interactive settings (e.g., hitting Shift+Enter on the same cell multiple times). Also document that if any dynamic factors (like global state or environment) change that could influence the meaning of code (rare, since meaning is mostly just source text), one might need to bypass cache. In typical usage, caching is safe because code semantics are fully determined by the source and compile flags.

Testing caching correctness:

```
# 1. Code producing output:
src = "print('Hello')"
```

```
out1 = run_code_with_cache(src)  # should print Hello
out2 = run_code_with_cache(src)  # should also print Hello, and ideally, the
compile should only happen once internally.
# 2. Ensure changes in code do trigger recompile:
src2 = "print('Goodbye')"
out3 = run_code_with_cache(src2)  # should print Goodbye, not reuse Hello code
object
# 3. Flags difference:
expr = "42"
co1 = get_code_object(expr, mode="eval", flags=0)
co2 = get_code_object(expr, mode="eval", flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
assert co1 != co2  # they should be different because flags differ (flag one
will have CO_COROUTINE set perhaps even if not needed)
```

This ensures our caching distinguishes keys properly.

By carefully implementing these strategies, we can achieve robust async execution with minimal overhead: - Single compilation for repeated runs, - No repeated imports or definitions thanks to hoisting, - and tracebacks that point to user code accurately, all of which contribute to a smooth developer experience in a custom execution engine.

Appendix: AST Node Reference and Version Caveats

- **AST Node Additions Summary**:
- *Python 3.8*: Added `end_lineno` and `end_col_offset` to AST nodes [3].
- *Python 3.9*: Introduced `ast.unparse`; no major new nodes (except Constant merging older Num/Str, etc.).
- *Python 3.10*: Added pattern matching nodes: `Match`, `match_case`, `Pattern` subclasses (PEP 634).
- *Python 3.11*: Added `TryStar` (PEP 654); `ExceptionGroup` in exceptions (not AST). End positions utilized in tracebacks (PEP 657). Variadic generics (PEP 646) introduced new forms in function sigs (no new node types, uses existing AST constructs).
- *Python 3.12*: Added `TypeAlias` and type parameter nodes (`TypeVar`, `ParamSpec`, `TypeVarTuple`) and `type_params` fields (PEP 695) [11] [12]. Formalized f-string grammar (PEP 701) – AST nodes unchanged but can contain more.
- *Python 3.12*: Comprehension inlining (PEP 709) – AST same, symbol table treatment changed (no separate scope) [15].

- *Python 3.13*: (At time of writing) Focus on no-GIL (PEP 703, experimental) – no AST changes. Some deprecated old parser stuff removed (like `parser` module), but AST usage remains via `ast` module.

- **Test Corpus for Validation**: A set of representative code snippets to verify our engine:

- **Top-level await**: e.g. `await asyncio.sleep(0); print("Done")` – ensure it runs and prints.

- **Async for/with**: e.g.

```
import asyncio, math
async for line in some_async_gen():  # define a dummy async gen yielding
lines
    print(line)
async with aiofile.open('f.txt') as f:  # dummy async context
    data = await f.read()
```

  – ensure syntax allowed and functions correctly with our compile approach.
- **Symbol hoisting cases**:
  - Import hoist: `import math; print(math.pi)` – math should be imported once.
  - Import shadow: `import math; math = "spam"` – math not hoisted, final math is "spam".
  - Pattern shadow: `import math; match 5: case math: print(math)` – math import not hoisted, and after execution math should be 5 (pattern bound) not the module.
  - Function hoist: `def foo(): return 1; print(foo())` – foo defined in prep, print executes correctly.
  - Function default dependency:

```
value = 10
def foo(x=value): return x
value = 20
print(foo())  # should print 10, not 20
```

    – If we hoisted `foo` above the first assignment, it would capture wrong value. Our logic should leave it in place or hoist in correct order. We test that it prints 10.
- **PEP 657 accuracy**:
  - Induce an error in a transformed context: e.g., code that causes ZeroDivisionError or NameError, and verify traceback lines match original code lines (manually or via automated check of exception **traceback** info).
  - For example, user code `print(1/0)` after a transformation that we do – does traceback highlight `1/0` exactly?
- **AST new features**:
  - Pattern matching usage: a snippet with `match` and a guard, ensure our engine runs it the same as normal.
  - Except *usage: a snippet raising an ExceptionGroup and handling with except*, ensure it works.
  - Type alias: `type X = tuple[int, int]` – should execute (bind X in globals) without error.
  - Generic class:

```
class MyList[T](list[T]): pass
```

    – at runtime this is mostly a no-op (creates a class), ensure no crash.
  - F-string edge:

```
expr = "world"
print(f"Hello {expr}")  # simple
print(f'This {"{"} is a brace')  # tricky nested braces scenario
```

– ensure f-string with braces works (PEP 701).

- **Performance**:
  - Time a loop running the same snippet 1000 times with caching vs without (just as a final integration test to see caching effect).
  - Not as a pass/fail test but to observe that cached version is significantly faster.

By running such a test corpus across Python 3.11, 3.12, 3.13, we can validate that: - All features behave correctly in our engine, - Our transformations do not alter semantics or break on newer constructs, - Performance is improved by caching and not regressed by the extra analysis.

This comprehensive approach ensures our custom execution engine can robustly handle async code transformations with minimal overhead and maximum correctness.

---

1 3 What's New In Python 3.8 — Python 3.13.7 documentation
https://docs.python.org/3/whatsnew/3.8.html

2 ActivePython 3.8.2 Documentation
https://docs.activestate.com/activepython/3.8/python/library/functions.html

4 5 6 7 8 9 10 11 12 13 24 26 ast — Abstract Syntax Trees — Python 3.13.7 documentation
https://docs.python.org/3/library/ast.html

14 15 27 28 What's New In Python 3.12 — Python 3.13.7 documentation
https://docs.python.org/3/whatsnew/3.12.html

16 17 18 19 20 21 22 symtable — Access to the compiler's symbol tables — Python 3.13.7 documentation
https://docs.python.org/3/library/symtable.html

23 What's New In Python 3.11 — Python 3.13.7 documentation
https://docs.python.org/3/whatsnew/3.11.html

25 Changelog — Python 3.11.2 documentation - IGM
https://igm.univ-mlv.fr/~beal/Teaching/DocPython/python-3.11.2-docs-html/whatsnew/changelog.html