ChatGPT

`PyCF_ALLOW_TOP_LEVEL_AWAIT`: Usage, Design, and Implications

Origin and Purpose of `PyCF_ALLOW_TOP_LEVEL_AWAIT`

Python's compiler flag `PyCF_ALLOW_TOP_LEVEL_AWAIT` was introduced in **Python 3.8** to enable **top-level** asynchronous code (using `await`, `async for`, and `async with`) in contexts where it was previously invalid syntax [1]. This flag was added as a direct response to the needs of interactive environments (e.g. IPython/Jupyter) to **simplify asynchronous experimentation**. Traditionally, `await` could only appear inside `async def` coroutines (per PEP 492), meaning top-level `await` would raise a syntax error. By passing the `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` flag to Python's built-in `compile()` function, developers can compile code containing top-level async constructs without error [2]. The primary purpose is to treat a chunk of code as if it were wrapped in an implicit `async` function, thus making top-level `await` feasible.

When this flag is used, the compiler *modifies its normal behavior*: the resulting code object is marked as a **coroutine**. In fact, the official docs state that with this flag, the compiled code object will have the `CO_COROUTINE` flag set in its `co_flags`, meaning it behaves like a coroutine that must be awaited or run in an event loop [2]. For example, compiling `source = "await asyncio.sleep(1)"` with `compile(source, '<input>', 'exec', ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)` produces a code object that, when evaluated, yields a coroutine object instead of immediately executing to a result [2]. This coroutine can then be awaited (e.g. via `await eval(code_object)`) to actually execute the top-level async code [2]. The **origin** of this feature can be traced to Python core developers collaborating with IPython contributors: at the 2019 Python Language Summit, they discussed making async code "convenient in the shell" and subsequently updated CPython's compiler to allow these top-level async constructs (when an appropriate event loop is provided) [3].

Implementation note: Initially, `PyCF_ALLOW_TOP_LEVEL_AWAIT` was assigned a value that collided with an existing flag. This was resolved in Python 3.8.3 – the flag's constant was moved to a higher bit (0x1000000) to avoid clashing with `CO_FUTURE_DIVISION` and other `__future__` flags [4]. Thus, modern Python releases use a unique flag value for top-level await support.

Implementation in CPython's Compiler Internals

Enabling top-level `await` required changes to Python's **compilation pipeline**. The flag is checked during the compilation of an AST into bytecode, specifically in CPython's `compile.c`. When the compiler is processing a module (top-level code) and sees that `c_flags.cf_flags` includes `PyCF_ALLOW_TOP_LEVEL_AWAIT`, it adjusts how it classifies the code. Internally, the symbol table entry (`PySTEntryObject`) for the module is marked as a **coroutine** (`ste_coroutine = 1`) if an `await` or other async construct is present at top level [5] [6]. This is controlled by a macro `IS_TOP_LEVEL_AWAIT(c)` which checks that the flag is set and the current compilation unit is a module

(not a normal function) [7]. If so, the compiler knows to permit `await` and friends in what would normally be an illegal location.

During **AST -> bytecode generation**, when assembling the code object, the compiler uses the `ste_coroutine` flag to set the appropriate code object flags. Specifically, if the module's symbol table is marked as a coroutine, and it's not also a generator, the compiler will set the code object's `CO_COROUTINE` flag (distinguishing it as a coroutine code) [8]. In effect, the entire module's code is treated like an async function's body. As noted in the documentation, "when the bit `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` is set, the returned code object has `CO_COROUTINE` set in `co_code`" [2]. This has a crucial impact: **executing** such a code object (via `exec` or `eval`) will not immediately run the code to completion but instead produce a **coroutine object** that must be awaited to drive the execution [2].

Under the hood, CPython's evaluator recognizes that the code object is a coroutine. If you call `eval()` on it in a normal context, you'll get a coroutine object (of type `coroutine`) rather than a result, as the code hasn't actually been run yet. If you never await this object, Python will issue a warning that "coroutine '<module>' was never awaited" [9]. This design ensures that top-level awaits don't execute until an event loop schedules them – matching the behavior of `async def` functions, which return coroutine objects that need an event loop to run.

REPL vs. script execution: In standard script execution (running a `.py` file with the regular interpreter), top-level `await` is still a syntax error unless special handling is in place. The compiler flag is mainly leveraged by interactive environments or explicitly via `compile()`. The *built-in* CPython REPL (when started normally) does **not** automatically enable top-level await. However, CPython provides an **asynchronous REPL mode**: running `python -m asyncio` will launch an interactive session where each input is compiled with `PyCF_ALLOW_TOP_LEVEL_AWAIT` and an asyncio event loop is run to execute the result [3] [10]. In this mode, one can directly type `await some_coroutine()` at the prompt and get the result, as asyncio is imported and a loop is handling the awaiting behind the scenes [10]. For normal modules (scripts), there isn't yet a built-in mechanism to automatically run an event loop for a top-level `await`. A proposed workaround is using a shebang or CLI invocation to use the asyncio runner – for example, a shebang `#!/usr/bin/env -S python3 -m asyncio` as discussed on Python's forums [11] – which effectively executes the script in that async-enabled REPL environment. Otherwise, to use this in a script, one would manually compile and execute the code via an event loop (or use an interactive shell that supports it). In summary, CPython's internals allow top-level await *when explicitly requested* by a flag, and the typical usage is in interactive settings or specialized runners rather than in standard module imports.

Impact on Third-Party Interpreters and REPLs

The availability of `PyCF_ALLOW_TOP_LEVEL_AWAIT` had a significant impact on **interactive Python tools**. Environments like IPython and Jupyter Notebook quickly adopted this flag to provide a smoother async experience:

- **IPython (Interactive Python)** – Starting in IPython 7.0 (with Python 3.6+), IPython introduced an *"autoawait"* feature to let users `await` coroutines at the top level [12]. Before Python 3.8, IPython achieved this by transforming code (e.g. wrapping it in an async function behind the scenes or using custom AST tricks). With Python 3.8, IPython can directly use `compile(code, '<input>', 'exec', PyCF_ALLOW_TOP_LEVEL_AWAIT)` to compile a user's input. After compilation, IPython

inspects the code object's flags to see if `CO_COROUTINE` is set (indicating the input was or contained an `await`) [2]. If so, IPython knows the execution should be asynchronous – it will **not** just exec the code normally, but instead schedule it on an event loop and `await` the result. In practice, IPython uses a helper like `inspect.CO_COROUTINE & code.co_flags` to detect this [13]. This allows constructs such as:

```
In [1]: import aiohttp; session = aiohttp.ClientSession()
In [2]: response = await session.get('https://api.github.com')
In [3]: data = await response.json()
```

to work seamlessly in IPython, printing the result of the `await` without requiring the user to define an async function or call `asyncio.run` [14] [15]. IPython's autoawait system is pluggable – by default it uses asyncio's event loop, but users can switch to other frameworks (like Trio or Curio) using the `%autoawait` magic [16]. The underlying mechanism remains the same: compile with the flag, detect coroutine code, and dispatch to the appropriate event loop runner. This decoupling was a key reason the flag was designed to be **agnostic to the async library** (it just marks the code; IPython decides *how* to run it) [17].

- **Jupyter Notebook / IPykernel** – The Jupyter IPython kernel builds on IPython's capabilities. When you run a cell in a Jupyter notebook that contains an `await` at top level, the kernel uses the same compile-and-detect approach to handle it. One difference in many notebook setups is that an **event loop is already running** (for example, Jupyter notebooks often have an asyncio loop running to handle I/O with the browser). In these cases, the kernel cannot simply call `asyncio.run()` (as that would try to start a second event loop); instead it uses techniques like `nest_asyncio` (which allows re-entering an already running loop) or runs the coroutine in the existing event loop (via `loop.create_task` and related mechanisms) [18]. The `PyCF_ALLOW_TOP_LEVEL_AWAIT` flag is crucial here because it lets the kernel **accept** code with top-level `await` and receive a coroutine object to schedule. From the end-user's perspective, they can write async code in a notebook cell just as they would in a script (minus the boilerplate), and the notebook will execute it and display the result once the awaitable is done. This has become a key feature for data science workflows (for example, querying an API asynchronously in a Jupyter cell).

- **Asyncio REPL and other shells** – As mentioned, CPython's standard library gained an **async REPL** mode via `python -m asyncio`. This is essentially a thin wrapper around the default REPL that uses the compilation flag and runs an event loop for each command [3]. It was noted as "currently it works only as REPL" in discussions, with suggestions to perhaps extend it to running files in the future [19]. Other third-party shells like **ptpython** or **bpython** could similarly leverage the flag to support top-level await. The Xonsh shell (a Python-powered shell) and IDLE could also incorporate it, though the adoption varies.

- **Pyodide and Web Assemblies** – A notable use-case of top-level await is **Pyodide**, which is Python compiled to WebAssembly running in the browser. In a browser environment, you typically want to avoid blocking the main thread. Pyodide provides an API `pyodide.runPythonAsync(code)` or `pyodide.code.eval_code_async(source)` which uses `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` under the hood to compile the provided Python code [20]. This returns a coroutine that Pyodide can hook into JavaScript's event loop (e.g., converting it to a JavaScript `Promise`). In effect, it allows one to `await` Python code execution from JavaScript. The top-level Python code can contain `await`

expressions (perhaps awaiting JavaScript `awaitable` proxies or just Python `asyncio` tasks), and the Pyodide runtime will handle it asynchronously. This bridges Python's async with JavaScript's async. For example, one could write in a Pyodide-powered web app:

```
# Python code running in browser via Pyodide
import some_async_api
data = await some_async_api.fetch_data()   # top-level await in Python
```

and have that not freeze the UI thread – instead, it yields control until the data is fetched. The implementation compiles the code with the flag and uses the coroutine object to tie into the browser event loop (via an awaitable JS promise) [20].

- **Web frameworks and others** – While typical web frameworks (Django, Flask) do not directly use `PyCF_ALLOW_TOP_LEVEL_AWAIT` (since web request handlers are functions, not top-level code), some specialized frameworks or tools might. For instance, **ASGI** frameworks like FastAPI or Starlette require an event loop for request handling but they usually manage coroutine execution explicitly (not via compile flags). However, one could imagine framework-specific management commands or shells using it. The **Piccolo ORM blog** highlighted top-level await as a convenient tool for **scripts that need async setup**, cautioning that only one loop can run at a time [21]. In general, this flag finds its use in **applications that embed Python interpreters or provide interactive consoles**, rather than in production server code. Tools like **Trio's REPL** or any environment that wants to evaluate user-provided code (like plugin systems or educational sandboxes) can use top-level await to simplify asynchronous code execution.

In summary, `PyCF_ALLOW_TOP_LEVEL_AWAIT` has empowered interactive Python environments to handle async code more naturally. IPython and Jupyter can be cited as major beneficiaries (no more `RuntimeError: loop is closed` confusion or needing to wrap code in `async def`), and even cross-language scenarios (Python in browser) leverage it. This has improved the **usability of asyncio**, helping to dispel the notion that "async is hard" by removing boilerplate in the contexts where it matters most [22].

Proposals, PEPs, and Key Discussions

The journey to top-level await support touches on several Python Enhancement Proposals and discussions:

- **PEP 492 (Python 3.5)** – Introduced `async def` and `await` syntax (native coroutines) but explicitly restricted `await` to **coroutine function bodies** and prohibited it at the module level. The design treated `await` similar to `yield` in that respect – only valid in specific contexts. This meant any top-level use of `await` was a syntax error, a rule enforced by the compiler.

- **PEP 525 (Python 3.6)** – Added asynchronous generators. Like PEP 492, the usage of `async for` and `yield` in async generators was confined to async function contexts. This PEP did not change the restriction on top-level code, but it added more async capabilities that similarly couldn't be used at module scope.

- **PEP 530 (Python 3.6)** – Enabled asynchronous comprehensions (e.g., `[i async for i in ait()]`) and the use of `await` inside comprehensions [23] [24]. However, PEP 530 specified that asynchronous comprehensions were **only allowed inside an** `async def` in Python 3.6 [23]. (This restriction was partly due to `async`/`await` being soft keywords in 3.6; by 3.7 they became hard keywords, theoretically allowing broader use, but the syntax was still not opened up to top level.) So by the end of 3.7, Python's grammar had a variety of async syntax forms – all disallowed at the top level of a module or normal function.

- **Early Ideas and Issue 34616** – The push for top-level await came from outside the core team initially. In mid-2018, IPython developer Matthias Bussonnier opened issue 34616 on the Python bug tracker titled *"implement Async exec"*, proposing first-class support for asynchronous execution in the REPL [25]. He outlined the challenges encountered in IPython's implementation of a pseudo-async REPL, such as: (1) the inability to compile top-level `await` to bytecode (in 3.7 one could parse it into an AST, but code generation failed without a flag) [26], (2) difficulty determining at runtime whether a code block needs to be awaited or executed normally (hence a desire for an indicator on the code object) [27], and (3) ensuring the solution wasn't tied to `asyncio` specifically (to allow other event loops) [17]. This issue sparked design discussions among core developers and the community on Python-Dev and Python-Ideas. The consensus was that it didn't make sense to *automatically* allow top-level await in all modules (due to questions like which loop to use, how to handle module imports, etc.), but it was very reasonable to support it in controlled environments like interactive shells.

- **Python Language Summit 2019** – At this annual core dev meeting, a lightning talk on the **Async REPL** was presented [28]. Bussonnier demonstrated IPython's approach and urged that core Python adopt some of these ideas for the default interpreter [22]. Core developer Yury Selivanov (author of PEP 492/525/530) collaborated on a solution. The outcome was an agreement to implement a compile flag for top-level await instead of introducing new compile modes. Within days, a series of pull requests were merged into CPython (e.g., PR #13148, #13472, #13484) to add this support [3]. The solution integrated smoothly: the `compile()` builtin was extended to accept the new flag, and the flag would signal the altered compilation behavior described earlier. This approach satisfied the need for an "async exec" without changing the fundamental exec/eval API – it's opt-in via a flag.

- **PEP 597?** – (Not directly related, but worth noting: there isn't a specific PEP solely for top-level await in Python modules. Any future proposal to allow `await` at module scope by default would likely require a PEP, but as of 2025, no such PEP has been approved. The current solution remains opt-in and primarily targeted at interactive use.)

- **Core Python Discussions** – On the Python-ideas forum, users periodically suggest "why not allow `await` at the top level of scripts by default?" A 2021 discussion, for instance, weighed the idea of letting the `asyncio` module be used to execute a script with top-level await (which is essentially the `python -m asyncio` idea) [29] [11]. Core developers pointed out complications: an async main script can't be imported by another module easily (since import is synchronous and would need to somehow know to await the module's top-level code), and choosing a default event loop runner is non-trivial in a general-purpose language [30] [31]. For now, the consensus is to **not** change the default semantics of module execution – instead, provide tools (like the flag and the `-m asyncio` runner) for those who need it. There was also a suggestion (by Petr Viktorin) that using a shebang with `python3 -m asyncio` is an acceptable solution for scripts, since the shebang already tells

the user how to run the script [11]. This indicates that *explicit is better than implicit* in this case – developers should clearly opt in to running a script with an event loop.

- **Bugfixes and Tweaks** – Following the initial release of the flag in 3.8, a few issues were discovered and fixed in minor releases:

- **Async comprehensions at top level:** Initially, even with the flag, using an *asynchronous comprehension* at top level (e.g., `result = [x async for x in agen()]`) could fail or misbehave. This was addressed in bpo-39562, which updated the compiler to allow asynchronous comprehensions when the flag is set [32]. (The fix ensured that the presence of an async comprehension would mark the code as needing to be a coroutine, similar to an `await` or `async for` statement would.)
- **Incorrect `await` in non-async function:** Another bug (bpo-39965) was that if the flag was globally set for compilation, an `await` *inside a regular function* could slip through without a SyntaxError in some cases. For example, compiling

```
def f():
    await foo()
```

with `PyCF_ALLOW_TOP_LEVEL_AWAIT` should still be an error (because `await` is not in an async function). A patch by Pablo Galindo in Python 3.8.2 fixed the compiler to **raise SyntaxError** in this scenario [33], ensuring the flag doesn't "mis-enable" await in illegal contexts. Essentially, the flag only affects the *module-level* context, not the rules inside function definitions.
- **List comprehensions marking `CO_COROUTINE` improperly:** There was a subtle case where any list comprehension at top level might cause the code to be flagged as a coroutine, even if it wasn't async. This is because internally list comprehensions create a generator-frame (they set `ste_generator = 1` for the comprehension scope). Initially, the compiler logic saw a generator at top-level *with* the allow-await flag and conservatively marked it as a coroutine too. This was corrected in Python 3.9's beta: now, **only** comprehensions that actually use `async for` or `await` get the coroutine flag [34]. Plain list comprehensions no longer cause a coroutine result when compiled with the flag. This fix prevented unintended "coroutine '<module>' was never awaited" warnings in code that, for example, used a list comprehension to populate a list at top level.

These proposals and discussions underscore that `PyCF_ALLOW_TOP_LEVEL_AWAIT` was carefully scoped. Rather than a blanket change to Python's syntax rules, it is a targeted tool for specific scenarios. PEP 530's notion that async comprehensions might one day be allowed at top level (once `async` became a keyword) did not automatically make it so – it ultimately took this compiler flag to realize that possibility in a controlled way. The Python core team's approach has been incremental: allow advanced usage with flags and CLI options, gather experience, and perhaps consider broader language changes in the future if warranted. As of now, **top-level await in the general case remains opt-in**, but that opt-in is readily available and widely used in the intended domains (interactive computing, education, prototyping, etc.).

Examples of Real-World Usage

To illustrate how `PyCF_ALLOW_TOP_LEVEL_AWAIT` is used in practice, let's look at a few scenarios and projects:

- **Interactive Console Example:** Suppose we want to evaluate a snippet of code that queries an API asynchronously. Using the compile flag, we can do this programmatically:

```
import asyncio, ast

code_str = "import aiohttp\nsession = aiohttp.ClientSession()\nresp = await
session.get('https://api.github.com')\ndata = await resp.json()
\nprint(data['current_user_url'])"
code_obj = compile(code_str, "<snippet>", "exec",
flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT)
result = eval(code_obj)        # This returns a coroutine object
asyncio.run(result)           # Run the coroutine to execute the code
```

In this example, `compile(..., PyCF_ALLOW_TOP_LEVEL_AWAIT)` produces a code object with `CO_COROUTINE`. The `eval(code_obj)` call returns a coroutine (it does *not* print anything by itself). We then explicitly run the coroutine in an event loop with `asyncio.run`, causing the awaited calls to execute. This pattern demonstrates how an application (or shell) might internally handle top-level async code. In an actual REPL like IPython, the loop management is handled for you, but under the hood it's doing something similar: checking if `result` is a coroutine and scheduling it in the event loop.

- **Jupyter Notebook (IPython)**: In a Jupyter notebook, one can simply do:

```
import asyncio
await asyncio.sleep(1)
print("Awake now!")
```

and the notebook will output "Awake now!" after a 1-second pause, without any extra ceremony. Behind this simplicity lies the IPython machinery described earlier (which itself uses the flag). For reference, IPython's documentation explicitly notes that constructs which are syntax errors in the standard Python REPL (like top-level `await` or `async with`) *"can be used seamlessly in IPython."* and that IPython will **automatically allow** `await` in the REPL when it detects such constructs [12] [15]. The `%autoawait` magic command can toggle this on or off [35] (it's on by default). Real-world projects, especially in scientific computing and education, rely on this feature; for instance, **machine learning notebooks** often use `await` to fetch data or results asynchronously, and tutorials can introduce async IO without needing to digress into event loop boilerplate.

- **Pyodide and PyScript**: PyScript (which builds on Pyodide) allows using Python in web applications. Consider a PyScript snippet:

```
<py-script>
  from js import fetch
  response = await fetch("https://api.github.com")
  text = await response.text()
  print(text[:100])
</py-script>
```

This looks almost like JavaScript (where top-level await is allowed in modules) but is actually Python code running in the browser. Under the hood, Pyodide's `eval_code_async` is compiling that `<py-script>` content with `PyCF_ALLOW_TOP_LEVEL_AWAIT` [20]. The first `await fetch(...)` yields control to the browser's event loop (allowing other operations or UI updates), and when the promise resolves, Pyodide resumes the Python coroutine to get the result. This is a powerful example of how the flag enables high-level scripting use-cases. Without top-level await, one would have to break that code into callbacks or define an async function and schedule it from JS – much less straightforward. Projects like **PyScript** use this to make Python a first-class citizen in the web async world.

- **Web Framework Startup**: While not as common, some application code uses the flag to run **initialization code asynchronously**. For example, a framework might allow users to specify an async setup script. By compiling it with this flag, the framework can load configuration (maybe calling await on database connections or external services) before the server starts. Most frameworks tend to use patterns like `asyncio.run(main())` in `if __name__ == "__main__":` for that, but the flag provides an alternative. One open-source example is an **async importer** in a blog post by Tony Fast, which demonstrates using `PyCF_ALLOW_TOP_LEVEL_AWAIT` to import notebooks as modules that can contain top-level awaits [36] [37]. This kind of dynamic import mechanism reads a file, compiles it with the flag, and executes the coroutine to completion (effectively simulating an "async module import"). It's more of an experiment than common practice, but it showcases creative uses of the flag outside the REPL.

- **Testing frameworks**: Tools like `pytest` have gained the ability to test async code. Normally, `pytest` does this by detecting `async def` test functions and running an event loop to execute them. Top-level await isn't directly used in test modules (as test files are modules that get imported), because again the import mechanism isn't async. However, one could imagine a variant of pytest or a plugin that allows writing async setup code outside of any function. Such a plugin might utilize this flag to compile a test file's setup section. For now, though, the prevalent approach is using `pytest.mark.asyncio` or similar to indicate which tests to run in a loop. So this remains an area where top-level await is *not* typically used – it's more for interactive or runtime-evaluated code rather than importable modules.

In summary, real-world usage of `PyCF_ALLOW_TOP_LEVEL_AWAIT` centers on making *asynchronous code more accessible*. From interactive notebooks to browser-based Python and beyond, it allows developers and users to mix `await` into top-level code with minimal friction. This has become so standard in those contexts that many users of IPython may not even realize a special compiler flag is at work – they just know that `await` "magically works" in their interactive sessions. That "magic" is precisely the controlled alteration of the compiler provided by `PyCF_ALLOW_TOP_LEVEL_AWAIT`.

(One more concrete illustration: as of Python 3.11, the built-in `code` *module's interpreter will also respect this flag. If you embed a Python interpreter via the C API or* `code.InteractiveInterpreter`*, you can pass in this flag to allow awaits. This is essentially how the asyncio REPL and IPython function internally. It underscores that the feature is general-purpose – any tool embedding Python or executing dynamic code can opt in.)*

Architectural Design Implications

The introduction of top-level await via a compiler flag brings along several design considerations and implications for Python's architecture and for developers using it:

Compiler Flag Propagation and Code Introspection

`PyCF_ALLOW_TOP_LEVEL_AWAIT` is a **compile-time flag**, meaning it only affects the parsing/compiling of a given code snippet. It does not propagate at runtime. For example, if you compile module A with the flag, and inside module A you use `exec()` on some code string, that inner `exec` won't automatically have top-level await enabled (you'd have to pass the flag again explicitly). This is by design – the flag is contained to the compilation context in which it's used, preventing unintentional bleed-over of async syntax into code that isn't expecting it.

One observable artifact is in **code object introspection**. As discussed, code objects compiled with the flag may have the `CO_COROUTINE` bit set in their `co_flags`. Tools like the `inspect` module can reveal this. For instance, `inspect.iscoroutinefunction` checks a function object's code for the `CO_COROUTINE` flag to decide if it's a native coroutine. In the case of a module compiled with top-level await, there is no function object, but if you have the code object, `bool(code.co_flags & inspect.CO_COROUTINE)` will be True [13]. This can be used programmatically to decide how to execute a code object. IPython's own `is_async_code` utility essentially does this check to see if a given compiled code needs to be awaited.

Another implication is how **future flags and compiler constants** interact. As noted earlier, there was a collision with `CO_FUTURE_DIVISION`. Technically, the `flags` parameter in `compile()` can include both future directive flags and this top-level await flag. Python's implementation had to ensure that the numeric values for these don't overlap. The resolution (assigning a higher bit) means the set of valid compiler flags (`PyCF_*`) grew. The compiler uses a mask (`PyCF_COMPILE_MASK`) to filter out irrelevant bits; `PyCF_ALLOW_TOP_LEVEL_AWAIT` is included in this mask, so it's recognized as a valid compile flag [38]. For introspection, this means if you see a code object's `co_flags`, bits like `0x2000` or `0x1000000` (depending on version) correspond to these features, although typically one would use the names from the `inspect` or `ast` modules rather than hardcoding numbers. It's worth noting that core developers had the foresight to adjust the **future** mechanism to avoid conflicts, indicating that this flag is expected to coexist with others cleanly going forward [4].

One corner case was how the flag affected inner scopes like comprehensions. Initially, as mentioned, simply having a list comprehension at top level triggered the coroutine flag because comprehensions use an implicit generator function. The fix means the compiler is smarter: it looks at whether the comprehension itself is asynchronous. If not, it won't mark the whole module as coroutine just because a generator is present [34]. This nuance ensures that the presence of *synchronous* generators or comprehensions at top level (e.g., a plain list comp or a normal generator expression) doesn't force the module to become async.

Only actual async usage does. In effect, the compiler flag **enables** top-level await where needed, but tries to **minimize side-effects** on code that doesn't need it.

Execution Model and Event Loop Interaction

Perhaps the biggest design implication is that top-level `await` blurs the line between normal sequential execution and event-loop-driven execution. In a normal script, when you run it, the Python interpreter executes statements one by one until completion (or blocking on I/O as needed). With top-level await, the execution model becomes **two-phased**: first the code is compiled and a coroutine object is obtained, then that coroutine must be run to actually do the work. This means an **event loop** must be in play to drive the coroutine.

For environments like IPython, this required careful integration. In a terminal IPython, the approach is to run an event loop **iteration per user input**. Essentially, IPython starts an asyncio event loop, runs the awaited code to completion (using `asyncio.get_event_loop().run_until_complete(coroutine)` internally), then closes or stops the loop until the next input. This ensures that each `await` block executes and returns control to the user promptly [39]. In contrast, in a Jupyter notebook (or any application where an event loop is already running continuously, e.g. for GUI or server purposes), the challenge is different: the loop is running in the background, and you need to schedule the top-level coroutine *on that loop* without stopping it. Libraries like **Nest AsyncIO** allow re-entering an active loop to accomplish this. Essentially, the **code execution model** in these cases is cooperative – the top-level code hands off execution to the event loop when encountering an await, allowing other tasks (or just the loop's normal operations) to proceed until the awaited task is done. This is exactly analogous to what happens inside an async function, but now it's happening at the module level.

One design implication is that any long-running top-level await **blocks the REPL** (in terminal usage) until it's done, because the interpreter is waiting for the coroutine to finish before returning to the prompt. This is usually fine (that's the expected behavior – you don't get the prompt back until the awaited coroutine is done). In notebook environments with a continuous loop, a long-running top-level await doesn't block the UI entirely; other loops (like the GUI or other tasks scheduled) can keep running. However, the user still must wait for the result before that cell is considered finished.

A critical limitation arises from **single-threaded event loop policy**: There can be only one event loop running in a given thread. If you have top-level await enabled and therefore an event loop already active (say, the asyncio REPL or a Jupyter kernel loop), you cannot call another loop starter like `asyncio.run()` or `loop.run_forever()` from within that context without error. For example, if someone tries to call `asyncio.run()` inside IPython (which itself is already running an asyncio loop in the background), they'll get a `RuntimeError` because a loop is already running [21]. The Piccolo ORM blog warned about this specifically: *"If any of your code tries to launch an event loop, perhaps by calling* `asyncio.run`*, you'll get an error"* [21]. The design choice here is to prefer having a *single, global loop* for interactive sessions. Users need to be aware of this to avoid conflicts. Some frameworks detect if they're running in an environment like IPython and adjust accordingly (for instance, not using `asyncio.run` if they see that `asyncio.get_event_loop().is_running()` is True). This interplay means that libraries and user code may need minor adaptations to play nicely with top-level await contexts (e.g., use `await something` rather than `asyncio.run(something)` if already in such an environment).

Another impact on execution: **Cleanup and shutdown**. If a top-level coroutine spawns background tasks (for example, `asyncio.create_task` is called at top level to start something that runs forever), what happens when the top-level coroutine completes? In a normal async function, if you spawn tasks and don't await them, they can keep running even after the function returns (until the loop stops). In an interactive environment, if you do something similar, those tasks will continue running on the loop after the prompt returns. This can be useful (IPython actually exposes this ability; you can schedule background tasks that run between cells on the same loop) [40], but it can also be confusing if not managed. The design leans towards giving control to the environment: IPython, for instance, might keep track of tasks created during the execution of a cell. Some environments could choose to cancel any leftover tasks once the top-level await is done to avoid surprises. Debugging such scenarios can be tricky, since an error in a background task might not surface immediately to the user. This is more of an application-level design consideration than a CPython implementation detail, but it stems from the fact that top-level await invites a more **asynchronous execution model with potential concurrency** (multiple tasks) in what used to be a strictly linear execution (one statement after another).

Error Handling, Exceptions, and Debuggability

Error semantics with top-level await remain largely consistent with normal async code, but there are a few points to highlight:

- If an exception is raised *before* the first await (i.e., in the synchronous portion of the top-level code), it will behave like a normal exception in module execution. Since the code is compiled as a coroutine, an exception at the top level will actually cause the coroutine to fail *immediately* when it's awaited. For instance, if you had:

  ```
  x = 1 / 0
  await asyncio.sleep(1)
  ```

  when you `await eval(code_obj)` (or the REPL tries to run it), it will raise `ZeroDivisionError` without ever hitting the sleep. The traceback in an interactive session would point to the line in the top-level code just as it would in a script, but might show something like in `<module>` context (which is effectively the coroutine's name). The flag doesn't change how tracebacks are captured apart from the fact that the code is one level deeper (inside a coroutine). In practice, IPython and similar environments print the traceback as if you ran it normally – they hide the internal coroutine machinery so that the top line isn't an obtuse "during handling of coroutine...".

- If an exception is raised *during* an await (i.e., the awaited coroutine fails), it will propagate out just like an exception from an `await` inside an async function. This means the coroutine object will end in an exception state. If you're awaiting it in an interactive loop, that loop will catch the exception and typically print the traceback. In IPython, for example, an unhandled exception in top-level await code will be caught by IPython's normal exception handling mechanism and displayed to the user. Debuggers like `pdb` can be used, but one needs to remember that the code is running under an event loop. Setting breakpoints in async code (using `breakpoint()` or `pdb.set_trace()`) does work – when the breakpoint is hit, you drop into the debugger, even if it was inside an awaited function.

- One tricky scenario is **partial execution**: If an error happens after some `await` has completed, some of your top-level code ran and some didn't. For example:

```
data = await fetch_data()    # suppose this succeeds
result = process(data)
await asyncio.sleep(5)       # suppose this raises CancelledError or so
save(result)
```

  If the second await raises (maybe the task was cancelled), the code never reaches `save(result)`. In a script, that means `result` would never be saved – same in the top-level await case. The difference is that in a regular script, if an error happens at some point, you know nothing after that ran. In an async REPL, it's possible the first await already did something (like network I/O) before the error occurred. This isn't new to async, but users have to be mindful that side effects before an error *will* have happened (e.g., data fetched and `process(data)` run), whereas anything after the failing await will not. Proper error handling (try/except around awaited sections) can be done at the top level as well. Interestingly, you can write top-level `try/except` around an await in a REPL and it works as expected.

- **Debuggability**: Traditional debugging tools have been updated to handle async, but using them at the REPL requires the environment's cooperation. For instance, `pdb` stepping through asynchronous code is possible (each await is like a yield point). In a top-level context, one might do:

```
await asyncio.sleep(1)  # set breakpoint() here
```

  The breakpoint will drop to pdb when hit. Since the top-level code is actually running under `asyncio.run` (for example), the debugger is invoked within that event loop's context. From a user perspective, it doesn't feel much different – you can inspect variables, etc. However, if one tries to manually step *over* an await (which involves suspending the coroutine and resuming later), pdb will treat the await kind of like a function call that isn't completed immediately. Recent improvements in Python allow `pdb` to step into async functions, but in older versions this was harder. In an interactive environment, the debugger interface might also compete with the event loop (since pdb is typically synchronous, it effectively pauses the loop while you're debugging).

- Another debugging aspect is **unawaited coroutine warnings**. If a user mistakenly calls an async function without awaiting it at top level, they'll get a coroutine object. In a regular script, if you do that and then script ends, you might see `RuntimeWarning: coroutine never awaited`. In IPython, if you do:

```
coro = some_async_func()  # forgot to await
coro
```

  IPython actually will detect that `coro` is a coroutine when trying to display it, and (as a convenience) will run it for you and show the result, *if* you haven't disabled autoawait. This is part of

the “autoawait” functionality – it tries to prevent users from leaving coroutine objects hanging around. But if you assign it to a variable and not evaluate it, you might not notice. Those warnings are printed to stderr typically. Developers using top-level await in non-REPL contexts should ensure they either await or explicitly ignore such coroutines to avoid resource leaks or warnings.

- **Resource cleanup**: Just as in any async code, if your top-level async uses context managers (`async with`) or loops (`async for`), those will ensure proper acquisition/release and iteration. If the top-level coroutine is cancelled (e.g., the user Kernel interrupts in Jupyter, or a TimeoutError is raised), the `async with` block will clean up as needed by raising `GeneratorExit` or `CancelledError` inside the context manager. CPython’s design ensures that `finally` blocks and context managers in the coroutine code are executed on cancellation. So, writing robust top-level async code with `try/finally` or `async with` is just as important – the structure is preserved by the compiler and runtime.

Limitations and Edge Cases

While `PyCF_ALLOW_TOP_LEVEL_AWAIT` is powerful, it comes with **limitations and edge cases** to be aware of:

- **Not for Library Modules**: You typically should **not** use top-level await in library modules that will be imported by others. Python’s import system is synchronous, and there is no built-in mechanism to automatically await a module’s code during import. If you try to import a module containing an `await` at top level (without using the compile flag trick), you’ll get a syntax error. Even if you compile it dynamically and execute it, the import machinery isn’t designed to integrate with an event loop. This is why proposals to allow top-level await in all modules have stalled – it breaks the import model where imports are blocking and sequential [31]. So, use top-level await for scripts and interactive tasks, but not in libraries that others `import` in the normal way.

- **Single Await vs Multiple Awaits**: It’s worth noting that when you compile a block of code with the flag, the entire block becomes one coroutine. You cannot, for example, partially execute some top-level code, then yield to the caller (like a generator could), then resume the rest later – at least not without more complex coroutine juggling. If you want to, say, do one `await`, return control, then do another `await` later, you’d need to break your code into multiple pieces or tasks. The flag doesn’t turn your module into something that can be awaited multiple times; it’s one-shot. As soon as the coroutine finishes (or raises), you can’t “await the module” again (you’d have to recompile or reuse the code). This is generally fine for REPL use (each input is a separate coroutine) but means that **you cannot yield control back to the caller in the middle of module code** except by using an inner construct (like if the module itself defines an async generator and yields). In other words, top-level await doesn’t make your module an async generator – it’s either a full coroutine or nothing. Most use-cases are okay with this, but it’s a design choice to highlight: top-level code is an all-or-nothing coroutine execution.

- **Performance**: There’s a tiny overhead to using the flag. The compiler does a bit more work to handle the async syntax, and the evaluator has to create a coroutine object and then run it, rather than just running the code directly. In REPL scenarios this overhead is negligible and not a concern. In a tight loop of repeatedly compiling and awaiting small code snippets, it might be a bit slower than

equivalent synchronous execution. However, since any awaited operation usually involves I/O or a context switch that dominates runtime, this overhead is rarely noticed.

- **Compatibility**: The flag was added in Python 3.8. Code that relies on it will not work on earlier versions (3.7 and below). For example, IPython 7.x running on Python 3.7 had to maintain a fallback where it emulated top-level await by source transformation (it actually wrapped code in `async def _pseudo(): ...` and then used `asyncio.get_event_loop().run_until_complete(_pseudo())`). With 3.8+, IPython switched to using the compile flag directly. So, developers targeting multiple Python versions should either conditionally use the flag or stick to patterns that work without it. Fortunately, checking for the flag's availability is simple (it exists in `ast` module for 3.8+).

- **CO_FUTURE Flags Interaction**: Since the flag lives in the same namespace as future flags, an edge case to consider is combining it with future statements. For example, if you compile code that has `from __future__ import annotations` and also pass `PyCF_ALLOW_TOP_LEVEL_AWAIT`, it should work – and it does, because the future flags bits are combined with compile flags. The Python compiler will handle each accordingly. The key is to use `dont_inherit=True` in compile if you don't want outer scope's futures to apply. This is more relevant for tools building on compile() than end-users, but it's an edge detail: the `flags` parameter is a bitfield, and `PyCF_ALLOW_TOP_LEVEL_AWAIT` is just one bit in it. In documentation it's noted under "compiler flags" alongside others like `PyCF_ONLY_AST` [41]. One should OR the flags properly if using multiple (e.g., `flags=ast.PyCF_ALLOW_TOP_LEVEL_AWAIT | ast.PyCF_ONLY_AST` to both allow await and get an AST).

- **Threading and event loops**: Top-level await is typically used in the main thread. If you try to run an asyncio event loop in a non-main thread, it can be done (with the `asyncio.new_event_loop()` and setting it in that thread's event loop policy). The compile flag itself is agnostic to thread – it just produces a code object. But if someone were to attempt to use it to run async code in multiple threads, they'd have to manage separate event loops per thread. This is advanced usage and not common, but worth mentioning: the constraint "one loop per thread" still applies. So you could have two different threads each using top-level await on their own event loop, but they won't share state easily. More practically, most GUI frameworks (like asyncio's integration with Tkinter or Qt) run the event loop in the main thread. So interactive async usage generally sticks to that.

- **Cleanup of loop**: Environments like `python -m asyncio` have to ensure that when you exit the REPL or on keyboard interrupt, the event loop is closed cleanly and any pending tasks are cancelled. This is handled internally (for instance, pressing Ctrl+C in an asyncio REPL will cancel the currently running top-level coroutine, similar to how it interrupts normal code). If writing your own loop to execute compiled top-level-await code, you should mirror this behavior: cancel the coroutine on interrupt to avoid leaving it hanging.

- **Documentation and learning curve**: While not a technical limitation, it's worth noting that top-level await can confuse new users if they don't realize it's a special case in interactive use. For example, a beginner might wonder why `await something` works in Jupyter but not when they put the same code in a script. This is an **educational edge case**: instructors and documentation often need to clarify that "in a script, you must call `asyncio.run()` or otherwise initiate the event loop; in a notebook/REPL, you might be able to use await directly because the environment is doing that for

you." The presence of this flag in Python's API is mostly aimed at tool authors, but its effect is user-visible in that way.

In conclusion, the architectural design of `PyCF_ALLOW_TOP_LEVEL_AWAIT` strikes a balance between flexibility and explicit control. It opens the door for more intuitive asynchronous code in specific scenarios, without altering Python's fundamental single-threaded, synchronous nature when it comes to module execution. The flag-based approach means there's **no magic happening unless you ask for it**, which aligns with Python's philosophy. At the same time, when you *do* ask for it (as IPython, asyncio REPL, etc. do), Python's compiler and runtime cooperate to provide a robust async experience at the top level. It's a great example of an internal design change (in the compiler) enabling higher-level usability improvements, all done in a way that maintains backward compatibility and developer intent (nothing "async" happens unless you explicitly opt in).

Sources:

- Python 3.8 release notes (What's New) – introduction of `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` for compile() [1], and notes on flag value collision fix [4].
- Official Python documentation for `compile()` – details on the flag's effect (CO_COROUTINE flag and usage in eval) [2].
- Python Language Summit 2019 report – discussion of async REPL and the subsequent CPython implementation by core developers [3].
- Bug tracker discussions (Issue 34616) – challenges identified by IPython team and design considerations for async exec [42] [17].
- IPython documentation – explanation of autoawait and usage of top-level async in IPython 7.0+ [12] [15].
- Pyodide documentation – use of `PyCF_ALLOW_TOP_LEVEL_AWAIT` in `eval_code_async` to run code in an async event loop [20].
- Piccolo ORM Blog – notes on using `python -m asyncio` and caveats about event loops with top-level await [21].
- Python bugs and commits – bpo-39562 and bpo-39965 changelog entries (allowing async comprehensions at top level; fixing incorrect await usage handling) [43].
- CPython compiler source (compile.c) – implementation of flag check and setting `CO_COROUTINE` for module blocks [2] [5].

---

[1] [4] 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

[3] [14] [22] [28] Python Software Foundation News: Python Language Summit Lightning Talks, Part 1
https://pyfound.blogspot.com/2019/06/python-language-summit-lightning-talks.html

[5] [6] [9] 1817710 – python-gmpy2 fails to build with Python 3.9 (doctest/asyncio REPL regression in Python)
https://bugzilla.redhat.com/show_bug.cgi?id=1817710

7 8 Python: Python/compile.c Source File - doxygen documentation | Fossies Dox
https://fossies.org/dox/Python-3.12.11/compile_8c_source.html

10 21 Top level await in Python - Piccolo Blog
https://piccolo-orm.com/blog/top-level-await-in-python/

11 19 29 30 31 Add support for top-level await - Ideas - Discussions on Python.org
https://discuss.python.org/t/add-support-for-top-level-await/11684

12 15 16 35 39 Asynchronous in REPL: Autoawait — IPython 9.0.0 documentation
https://ipython.readthedocs.io/en/9.0.0/interactive/autoawait.html

13 mysite/venv/Lib/site-packages/IPython/core/async_helpers.py - GitLab
https://git.imp.fu-berlin.de/tolgayurt/packing_polygons_pycharm/-/blob/353cc3426e3d15a2a24432c995d0a287c00db8cb/mysite/venv/Lib/site-packages/IPython/core/async_helpers.py

17 18 25 26 27 40 42 Message 324900 - Python tracker
https://bugs.python.org/msg324900

20 36 pyodide.code — Version 0.28.2
https://pyodide.org/en/stable/usage/api/python-api/code.html

23 24 PEP 530 – Asynchronous Comprehensions | peps.python.org
https://peps.python.org/pep-0530/

32 33 43 ActivePython 3.8.2 Documentation
https://docs.activestate.com/activepython/3.8/python/whatsnew/changelog.html

34 external/python/Misc/NEWS.d/3.9.0b5.rst · master ... - ICHEC Gitlab
https://git.ichec.ie/ciaran.orourke/wflow/-/blob/master/external/python/Misc/NEWS.d/3.9.0b5.rst

37 asynchronous imports in python - tonyfast
https://tonyfast.github.io/tonyfast/xxii/2022-11-12-async-import.html

38 Diff - refs/tags/ndk-r23-beta1^! - platform/prebuilts/python/linux-x86
https://android.googlesource.com/platform/prebuilts/python/linux-x86/+/refs/tags/ndk-r23-beta1%5E%21/

41 ast — Abstract Syntax Trees — Python 3.13.7 documentation
https://docs.python.org/3/library/ast.html