Python support for free threading

- Installation
- Identifying free-threaded Python
- The global interpreter lock in free-threaded Python
- Thread safety
- Known limitations
  - Immortalization
  - Frame objects
  - Iterators
  - Single-threaded performance
- Behavioral changes
  - Context variables
  - Warning filters
  - Increased memory usage
    - All interned strings are immortal
    - Non-GC objects have a larger object header
    - QSBR can delay freeing of memory
    - mimalloc allocator vs pymalloc
    - Free-threaded reference counting can cause objects to live longer
    - Per-thread reference counting can delay freeing objects

Starting with the 3.13 release, CPython has support for a build of Python called `free threading` where the `global interpreter lock` (GIL) is disabled. Free-threaded execution allows for full utilization of the available processing power by running threads in parallel on available CPU cores. While not all software will benefit from this automatically, programs designed with threading in mind will run faster on multi-core hardware.

Some third-party packages, in particular ones with an `extension module`, may not be ready for use in a free-threaded build, and will re-enable the `GIL`.

This document describes the implications of free threading for Python code. See `freethreading-extensions-howto` for information on how to write C extensions that support the free-threaded build.

`703` – Making the Global Interpreter Lock Optional in CPython for an overall description of free-threaded Python.

Installation

Starting with Python 3.13, the official macOS and Windows installers optionally support installing free-threaded Python binaries. The installers are available at https://www.python.org/downloads/.

For information on other platforms, see the Installing a Free-Threaded Python, a community-maintained installation guide for installing free-threaded Python.

When building CPython from source, the `--disable-gil` configure option should be used to build a free-threaded Python interpreter.

Identifying free-threaded Python

To check if the current interpreter supports free-threading, `python -VV <-V>` and `sys.version` contain "free-threading build". The new `sys._is_gil_enabled` function can be used to check whether the GIL is actually disabled in the running process.

The `sysconfig.get_config_var("Py_GIL_DISABLED")` configuration variable can be used to determine whether the build supports free threading. If the variable is set to `1`, then the build supports free threading. This is the recommended mechanism for decisions related to the build configuration.

The global interpreter lock in free-threaded Python

Free-threaded builds of CPython support optionally running with the GIL enabled at runtime using the environment variable `PYTHON_GIL` or the command-line option `-X gil`.

The GIL may also automatically be enabled when importing a C-API extension module that is not explicitly marked as supporting free threading. A warning will be printed in this case.

In addition to individual package documentation, the following websites track the status of popular packages support for free threading:

- https://py-free-threading.github.io/tracking/
- https://hugovk.github.io/free-threaded-wheels/

Thread safety

The free-threaded build of CPython aims to provide similar thread-safety behavior at the Python level to the default GIL-enabled build. Built-in types like `dict`, `list`, and `set` use internal locks to protect against concurrent modifications in ways that behave similarly to the GIL. See `threadsafety` for the guarantees provided by built-in types.

> [!NOTE] It's recommended to use the `threading.Lock` or other synchronization primitives instead of relying on the internal locks of built-in types, when possible.

Known limitations

This section describes known limitations of the free-threaded CPython build.

Immortalization

In the free-threaded build, some objects are `immortal`. Immortal objects are not deallocated and have reference counts that are never modified. This is done to avoid reference count contention that would prevent efficient multi-threaded scaling.

As of the 3.14 release, immortalization is limited to:

- Code constants: numeric literals, string literals, and tuple literals composed of other constants.
- Strings interned by `sys.intern`.

Frame objects

It is not safe to access `frame.f_locals` from a `frame <frame-objects>` object if that frame is currently executing in another thread, and doing so may crash the interpreter.

Iterators

It is generally not thread-safe to access the same iterator object from multiple threads concurrently, and threads may see duplicate or missing elements.

Single-threaded performance

The free-threaded build has additional overhead when executing Python code compared to the default GIL-enabled build. The amount of overhead depends on the workload and hardware. On the pyperformance benchmark suite, the average overhead ranges from about 1% on macOS aarch64 to 8% on x86-64 Linux systems.

Behavioral changes

This section describes CPython behavioural changes with the free-threaded build.

Context variables

In the free-threaded build, the flag `~sys.flags.thread_inherit_context` is set to true by default which causes threads created with `threading.Thread` to start with a copy of the `~contextvars.Context()` of the caller of `~threading.Thread.start`. In the default GIL-enabled build, the flag defaults to false so threads start with an empty `~contextvars.Context()`.

Warning filters

In the free-threaded build, the flag `~sys.flags.context_aware_warnings` is set to true by default. In the default GIL-enabled build, the flag defaults to false. If the flag is true then the `warnings.catch_warnings` context manager uses a context variable for warning filters. If the flag is false then `~warnings.catch_warnings` modifies the global filters list, which is not thread-safe. See the `warnings` module for more details.

Increased memory usage

The free-threaded build will typically use more memory compared to the default build. There are multiple reasons for this, mostly due to design decisions.

All interned strings are immortal

For modern Python versions (since version 2.3), interning a string (e.g. with `sys.intern`) does not cause it to become immortal. Instead, if the last reference to that string disappears, it will be removed from the interned string table. This is not the case for the free-threaded build and any interned string will become immortal, surviving until interpreter shutdown.

Non-GC objects have a larger object header

The free-threaded build uses a different `PyObject` structure. Instead of having the GC related information allocated before the `PyObject` structure, like in the default build, the GC related info is part of the normal object header. For example, on the AMD64 platform, `None` uses 32 bytes on the free-threaded build vs 16 bytes for the default build. GC objects (such as dicts and lists) are the same size for both builds since the free-threaded build does not use additional space for the GC info.

QSBR can delay freeing of memory

In order to safely implement lock-free data structures, a safe memory reclamation (SMR) scheme is used, known as quiescent state-based reclamation (QSBR). This means that the memory backing data structures allowing lock-free access will use QSBR, which defers the free operation, rather than immediately freeing the memory. Two examples of these data structures are the list object and the dictionary keys object. See `InternalDocs/qsbr.md` in the CPython source tree for more details on how QSBR is implemented. Running `gc.collect` should cause all memory being held by QSBR to be actually freed. Note that even when QSBR frees the memory, the underlying memory allocator may not immediately return that memory to the OS and so the resident set size (RSS) of the process might not decrease.

mimalloc allocator vs pymalloc

The default build will normally use the "pymalloc" memory allocator for small allocations (512 bytes or smaller). The free-threaded build does not use pymalloc and allocates all Python objects using the "mimalloc" allocator. The pymalloc allocator has the following properties that help keep memory usage low: small per-allocated-block overhead, effective memory fragmentation prevention, and quick return of free memory to the operating system. The mimalloc allocator does quite well in these respects as well but can have some more overhead.

In the free-threaded build, mimalloc manages memory in a number of separate heaps (currently four). For example, all GC supporting objects are allocated from their own heap. Using separate heaps means that free memory in one heap cannot be used for an allocation that uses another heap. Also, some heaps are configured to use QSBR (quiescent-state based reclamation) when freeing the memory that backs up the heap (known as "pages" in mimalloc terminology). The use of QSBR creates a delay between all memory blocks for a page being freed and the memory page being released, either for new allocations or back to the OS.

The mimalloc allocator also defers returning freed memory back to the OS. You can reduce that delay by setting the environment variable `!MIMALLOC_PURGE_DELAY` to `0`. Note that this will likely reduce the performance of the allocator.

Free-threaded reference counting can cause objects to live longer

In the default build, when an object's reference count reaches zero, it is normally deallocated. The free-threaded build uses "biased reference counting", with a fast-path for objects "owned" by the current thread and a slow path for other objects. See `703` for additional details. Any time an object's reference count ends up in a "queued" state, deallocation can be deferred. The queued state is cleared from the "eval breaker" section of the bytecode evaluator.

The free-threaded build also allows a different mode of reference counting, known as "deferred reference counting". This mode is enabled by setting a flag on a per-object basis. Deferred reference counting is enabled for the following types:

- module objects
- module top-level functions
- class methods defined in the class scope
- descriptor objects
- thread-local objects, created by `threading.local`

When deferred reference counting is enabled, references from Python function stacks are not added to the reference count. This scheme reduces the overhead of reference counting, especially for objects used from multiple threads. Because the stack references are not counted, objects with deferred reference counting are not immediately freed when their internal reference count goes to zero. Instead, they are examined by the next GC run and, if no stack references to them are found, they are freed. This means these objects are freed by the GC and not when their reference count goes to zero, as is typical.

Per-thread reference counting can delay freeing objects

To avoid contention on the reference count fields of frequently shared objects, the free-threaded build also uses "per-thread reference counting" for a few selected object types. Rather than updating a single shared reference count, each thread maintains its own local reference count array, indexed by a unique id assigned to the object. The true reference count is only computed by summing the per-thread counts when the object's local count drops to zero. Per-thread reference counting is currently used for:

- heap type objects (classes created in Python)
- code objects
- the `__dict__` of module objects

Because the per-thread counts must be merged back to the object before it can be deallocated, objects using per-thread reference counting are typically freed later than they would be in the

default build. In particular, such an object is usually not freed until the thread that referenced it reaches a safe point (for example, in the "eval breaker" section of the bytecode evaluator) or exits. Running `gc.collect` will merge the per-thread counts and allow these objects to be freed.

□□□ #2
□ □□□ □□□ 2026-10-31 16:59:21 UTC
□ □□□ □□□ 2026-10-31 16:59:23 UTC