Python 3.13 Free-Threading Benchmarks

Question: What changed in Python 3.13's free-threaded build, and what do benchmarks show so far?

Python 3.13 introduced an experimental free-threaded build that can run without the global interpreter lock (GIL), per PEP 703 .

DEPTH standard_deep SOURCES 12

disco LATIN · VERB /ˈdɪs.koː/

"I learn; I become acquainted with."

from ***discere*** — to learn

> Python 3.13 introduced an experimental free-threaded build that can run without the global interpreter lock (GIL), per PEP 703 [1] [3]. The design replaces the GIL's global mutual exclusion with per-object locks, biased reference counting, immortalization, and deferred reference counting, and swaps CPython's pymalloc for mimalloc [1].
>
> The shipped 3.13 free-threaded build carries a substantial single-threaded penalty — about 40% on the pyperformance suite — because the specializing adaptive interpreter (PEP 659) is disabled in that build [3]. Benchmarks so far show a consistent, workload-dependent trade-off: parallel workloads on independent data scale well (up to ~4x in a 3.14.2 study, ~3x in an 8-thread FOSDEM 2026 test, 7.2x in one 8-core hash job) [6] [12] [9], while workloads that share mutable containers degrade because per-object locks serialize access [7].
>
> By 3.14 the single-threaded pyperformance penalty had fallen to about 10% (PEP 779) [5], and a FOSDEM 2026 benchmark measured ~1.2% on one 3.14t workload [12].
>
> Ecosystem support is partial and growing [11] [8], and free-threading is not a universal improvement [6] [7].

01

The mechanism: how free-threading replaces the GIL

PEP 703's core claim is that the GIL can be removed by making reference counting thread-safe at low cost and protecting containers with per-object locks. The reference counting change combines three techniques: biased reference counting, immortalization, and a limited form of deferred reference counting [1]. Biased reference counting, first described in 2018 by Choi, Shull, and Torrellas, exploits the observation that most objects are accessed by a single thread even in multi-threaded programs: each object is associated with an owning thread, which modifies a local refcount with non-atomic instructions, while other threads use atomic instructions on a shared refcount [1]. The PyObject header gains fields for the owning thread id, a one-byte per-object mutex, GC bits, a local refcount, and a shared refcount [1].

Immortalization marks objects that live for the program's lifetime (interned strings, small integers, statically allocated type objects, True/False/None) by setting the local refcount to UINT32_MAX, making INCREF/DECREF no-ops and avoiding contention [1]. Deferred reference counting targets objects frequently accessed by many threads — top-level functions, code objects, modules, and methods — by marking them with the two most significant bits of the local refcount and having the interpreter skip refcount operations as they are pushed and popped from the interpreter stack; the true refcount can only be computed when all threads are paused during cyclic garbage collection, so these objects can only be deallocated during GC cycles [1]. The PEP notes these objects already naturally form reference cycles in CPython [1].

For containers, every list, dictionary, and set gets an associated lightweight lock [1]. To avoid deadlocks that per-object locking could introduce, the PEP proposes "Python critical sections" (Py_BEGIN/END_CRITICAL_SECTION macros) that implicitly release per-object locks when a thread would block, and Py_BEGIN_CRITICAL_SECTION2 for two-object operations with lock ordering by memory address [1]. A few dict and list operations — item fetch and iteration — optimistically avoid acquiring locks, with a fast path that falls back to the locked path when another thread is concurrently modifying the container; this is motivated by the need for scalable access to shared module and class dictionaries and by reducing single-threaded overhead [1].

Memory management changes: pymalloc is replaced with mimalloc, a thread-safe allocator [1]; free lists move to per-thread state [1]; and the cyclic garbage collector becomes non-generational, requiring two stop-the-world pauses per cycle [1]. The C API gains new non-borrowed-reference functions (PyList_FetchItem, PyDict_FetchItem, PyWeakref_FetchObject) to replace unsafe borrowed-reference patterns [1]. The reference implementation changes roughly 15,000 lines of CPython code plus about 15,000 lines of mimalloc [1].

02

What Python 3.13 shipped

The GIL remains the default; the free-threaded build is opt-in via the --disable-gil configure flag, which defines Py_GIL_DISABLED and uses an ABI tag ending in "t" [1]. Official macOS and Windows installers optionally install free-threaded binaries [3], run via a separate executable named python3.13t or python3.13t.exe [2]. The build is not ABI-compatible with the standard build or the stable ABI because of the changed object header, so C-API extensions must be rebuilt specifically for it [1]; pip 24.1 or newer is required to install C-extension packages in the free-threaded build [2], and the pkg-config file is named python-3.13t.pc [2].

At runtime, the GIL can be re-enabled via the PYTHON_GIL environment variable or the -X gil command-line option, and it auto-enables (with a warning) when importing a C-API extension not marked free-threading-safe via the Py_mod_gil slot [3] [1]. The PYTHONGIL environment variable forces the GIL off (0) or on (1) [1]. sys._is_gil_enabled() and sysconfig's Py_GIL_DISABLED identify the build and runtime state [3] [2].

The status is explicitly experimental. The 3.13 documentation says to "expect some bugs and a substantial single-threaded performance hit" [2] [3]. The Steering Council accepted PEP 703 with a proviso that the rollout be gradual and that changes could be rolled back — including potentially all of PEP 703 — if too disruptive [1].

Known limitations in 3.13: objects are immortalized when a new thread is first started — function objects declared at module level, method descriptors, code objects, module objects and their dictionaries, and classes — and because immortal objects are never deallocated, applications creating many such objects may see increased memory usage, expected to be

addressed in 3.14 [3]. Frame objects are not safe to access from other threads, making sys._current_frames() generally unsafe [3]. Sharing the same iterator between threads is generally not safe [3].

03

Benchmarks: the single-threaded cost

The single-threaded cost is the clearest documented figure. The 3.13 howto states the free-threaded build has about 40% overhead on the pyperformance suite compared to the GIL-enabled build, with the largest impact from the specializing adaptive interpreter (PEP 659) being disabled in the free-threaded build; programs spending most time in C extensions or I/O see less impact, and the target is 10% or less [3].

This shipped reality diverges from PEP 703's own performance section, which reported execution overhead on pyperformance 1.0.6 of 6% (one thread) and 8% (multiple threads) on Intel Skylake and 5%/7% on AMD Zen 3, measured against a 3.12-era baseline [1]. The gap is explained by the specializing interpreter: the PEP's numbers predate the decision to disable it in the free-threaded build, and multiple independent sources attribute the 3.13 slowdown to that disablement [3] [4] [10].

By 3.14 the picture improved substantially. PEP 779 (Final, June 2025) reports the pyperformance penalty is currently around 10% (except macOS, ~3%), with PRs in flight to get below 10% on Linux and Windows, and about 15–20% higher memory use (geometric mean) [5]. A FOSDEM 2026 talk by Ruben Hias of TechWolf measured a single-threaded workload at 2.341s on 3.14 vs 2.370s on 3.14t — about 1.2% overhead — and described single-threaded overhead as "now minimal (~3% vs Python 3.14)" [12].

One caution: the ~40% figure is for single-threaded pyperformance. A different, multi-threaded workload can show much smaller penalties. johal.in's postmortem measured a 4-thread SHA-256 workload on 3.13.0 free-threaded at 121,890 iterations/sec vs 128,450 on 3.12.1 — about 94% of 3.12 throughput — with memory at 48.2 MB vs 42.1 MB [9]. That same postmortem documents a separate confound: 3.13.0's GIL-enabled build had a regression (30–70% throughput drops for multi-threaded CPU workloads) traced to a change in GIL hold interval from 5ms to 15ms, fixed in 3.13.1 [9]. Any multi-threaded CPU benchmark comparing 3.13.0 free-threaded against 3.13.0 GIL-enabled is comparing against a regressed baseline, since the documented regression was specific to multi-threaded CPU workloads [9].

04

Benchmarks: multi-threaded scaling and real workloads

The multi-threaded results across sources are consistent in pattern: free-threading helps when threads work on independent data and hurts when they contend on shared mutable containers [4] [7] [6] [12].

Benchmark	Build tested	Workload	Result
CodSpeed PageRank	3.13t (GIL off)	PageRank, 16-core ARM64	Multithreaded fastest with GIL off; free-threaded build slower on all other implementations [4]
julian.ac	3.14.0a4	Pure-Python matrix multiply	Terrible scaling; per-object lock contention on shared lists; fixed by separate copies [7]
arXiv 2603.04782	3.14.2	Parallel workloads on independent data	Up to 4x execution-time reduction, proportional energy reduction, higher memory [6]
FOSDEM 2026	3.14t	8-thread CPU-bound sum	~3x speedup vs 3.14 [12]
johal.in	3.13.0 free-threaded	8-core hash job	7.2x speedup vs single-threaded; 5.8x for multiprocessing [9]

CodSpeed's PageRank benchmark (3.12 vs 3.13 vs 3.13t with and without GIL, on 16-core ARM64 runners) found 3.12 and 3.13 (GIL) perform very similarly; the multiprocessing implementation was even slower than single-threaded due to inter-process communication overhead; the multithreaded implementation was fastest on 3.13t with the GIL disabled; but the free-threaded build showed a significant slowdown for all other implementations because the specializing adaptive interpreter is disabled. CodSpeed gives qualitative conclusions rather than concrete speedup numbers, and concludes free-threading is a promising alternative to multiprocessing but not yet production-ready [4].

An independent experiment by Julian Schrittwieser (May 2025, on a 3.14.0a4 free-threaded build) found reasonable scaling in Mandelbrot and prime-number-sieve benchmarks (threads with little interaction) but "absolutely terrible scaling" in pure-Python matrix multiplication where all threads read the same matrices and write to a shared result matrix. The cause is lock contention: Python cannot know the access is read-only, so list accesses acquire per-object critical-section locks; passing separate list copies to each thread restored expected scaling [7]. This is the mechanism from the first section showing up in practice.

A March 2026 arXiv preprint (2603.04782, on Python 3.14.2) found: for parallelizable workloads on independent data, the free-threaded build reduces execution time by up to 4x with proportional energy reduction and effective multi-core utilization, at the cost of increased memory usage (more visible in virtual than physical memory, attributed to per-object locking, thread-safety mechanisms, and the new memory allocator). Sequential workloads show no benefit and a 13–43% increase in energy consumption; workloads with frequent shared-object access show reduced gains or degradation due to lock contention. Energy is proportional to execution time [6]. As a preprint, this study has not been peer-reviewed.

The negative results are equally instructive. A Stack Overflow case (September 2024, 3.13.0rc2) found python3.13t took 56s vs 26s for python3.13 and 25s for 3.12 on a ThreadPoolExecutor randint workload; the answer attributes this to randint's static import sharing a mutex between threads, plus the specializing adaptive interpreter being disabled in free-threaded 3.13 — a comment quotes discuss.python.org confirming the specializing interpreter is switched off, causing significant single-threaded slowdown, "you may have to wait for Python 3.14" [10].

The synthesis across all sources is that the per-object locking design means Python cannot distinguish read-only from read-write access to shared containers, so any shared mutable container becomes a contention point [7]. Free-threading is a clear win only when threads avoid sharing mutable containers [7] [6] [4].

05

Ecosystem compatibility and remaining limitations

Ecosystem support is partial and growing. The py-free-threading tracking page lists packages with active free-threaded support including NumPy (2.1.0), pandas (2.2.3), SciPy (1.15.0), scikit-learn (1.6.0), PyTorch (2.6.0), Cython (3.1.0), pybind11 (2.13), nanobind (2.2.0), and many others, with some still pending (OpenCV, polars, vLLM, Triton) [11]. hugovk's free-threaded-wheels tracker covers the top 360 most-downloaded packages with extensions on PyPI: dark-green packages offer free-threaded wheels (ABI tag ending in t, e.g., cp313t), light-green offer pure-Python wheels, and orange offer no free-threading-ready wheels yet; pure-Python wheels work without changes, and only extension wheels need updating [8].

The C-API burden is the main barrier: extensions must be rebuilt for the free-threaded ABI, extensions relying on the GIL for protection need explicit locking, and unsafe borrowed-reference uses must switch to the new non-borrowed APIs [1]. Memory overhead is a recurring cost: 12–15% per process (johal.in) [9], 15–20% geometric mean on pyperformance (PEP 779) [5], and increased memory more visible in virtual than physical (arXiv) [6]. johal.in recommends free-threaded mode for batch CPU workloads (data ETL, image resizing, scientific computing) but not yet for production web services [9].

One question remains open: what the community discussion of free-threading performance concluded is not available in this evidence pool; the only trace here is the Stack Overflow comment quoting it on the specializing interpreter being switched off [10].

06

Roadmap: from experimental to supported

PEP 703's acceptance defined three phases: Phase I (started early in 3.13 development) made the free-threaded build available but explicitly experimental; Phase II would make it officially supported but optional; Phase III would make it the default [5]. PEP 779 (Final, resolved 16 June 2025) establishes the Phase II criteria: a hard 15% performance target, a 20% memory target (geometric mean on pyperformance), proven stable APIs, and internal documentation; it notes the 3.13 API design is stable — no breaking of 3.13 free-threading APIs in 3.14 — and that more ecosystem support is needed before deciding on default status [5]. PEP 703's own projected timeline anticipated 3.13 in 2024 with the --disable-gil flag and two ABIs [1].

07

Where the evidence lands

The evidence supports a clear judgment: 3.13's free-threaded build is a genuine but experimental capability with a documented cost-benefit trade-off [1] [3] [4]. The single-threaded cost fell sharply from 3.13 (~40% on pyperformance) to 3.14 (~10%, ~1.2–3% in one FOSDEM test) [3] [5] [12], while the multi-threaded benefit pattern held: large gains on independent-data parallel workloads, degradation on shared mutable containers [6] [7]. The decisive uncertainties are ecosystem readiness — how many extension packages ship free-threaded wheels and how fast [11] [8] — and whether the memory overhead and shared-container contention are acceptable for production workloads [5] [9]. Evidence that would change the judgment: broader ecosystem support, further reduction of single-threaded overhead below 10% on Linux and Windows [5], and resolution of the shared-container contention issue [7].

SOURCES (12)

[1] PEP 703 – Making the Global Interpreter Lock Optional in CPython | peps.python.org — https://peps.python.org/pep-0703

[2] What's New In Python 3.13 — Python 3.13.15 documentation — https://docs.python.org/3.13/whatsnew/3.13.html

[3] Python experimental support for free threading — Python 3.13.15 documentation — https://docs.python.org/3.13/howto/free-threading-python.html

[4] State of Python 3.13 Performance: Free-Threading | CodSpeed — https://codspeed.io/blog/state-of-python-3-13-performance-free-threading

[5] PEP 779 – Criteria for supported status for free-threaded Python | peps.python.org — https://peps.python.org/pep-0779/

[6] [2603.04782v1] Unlocking Python's Cores: Hardware Usage and Energy Implications of Removing the GIL — http://arxiv.org/abs/2603.04782v1

[7] Experimenting with free threaded Python — https://www.julian.ac/blog/2025/05/04/experimenting-with-free-threaded-python/

[8] Free-Threaded Wheels — https://hugovk.github.io/free-threaded-wheels/

[9] Postmortem: Python 3.13 GIL Caused Performance Bottleneck for CPU-Intensive Workloads — johal.in — https://johal.in/postmortem-python-313-gil-caused-performance-bottleneck-cpu-intensive-python

[10] gil - Python 3.13 with free-thread is slow - Stack Overflow — https://stackoverflow.com/questions/79009542/python-3-13-with-free-thread-is-slow

[11] Compatibility Status Tracking - Python Free-Threading Guide — https://py-free-threading.github.io/tracking/

[12] The GIL and API Performance — https://fosdem.org/2026/events/attachments/ABJMWD-the_gil_and_api_performance_past_present_and_free-threaded_future/slides/266979/the_gil_a_fw7etsx.pdf