PyData Roma Capitale + PyRoma
Meetup @ The Social Hub

The pivotal Python version

Who am I ?

My name is **Alessandro Pappalettera,**

I am a senior software engineer with 13+ years of experience, mistakes, experiments, django and lifelong learning mindset.

https://www.linkedin.com/in/alessandro-pappalettera/

https://github.com/Ardenine

Table of contents

New cool features for the interpreter

- PEP 649 and PEP 749: Deferred evaluation of annotations
- PEP 734: Multiple interpreters in the standard library
- PEP 750: Template strings
- PEP 758: Allow except and except* expressions without brackets
- PEP 765: Control flow in finally blocks
- PEP 768: Safe external debugger interface for CPython
- A new type of interpreter
- Free-threaded mode improvements
- Improved error messages
- Incremental garbage collection

PEP 734 – Multiple Interpreters in the Stdlib

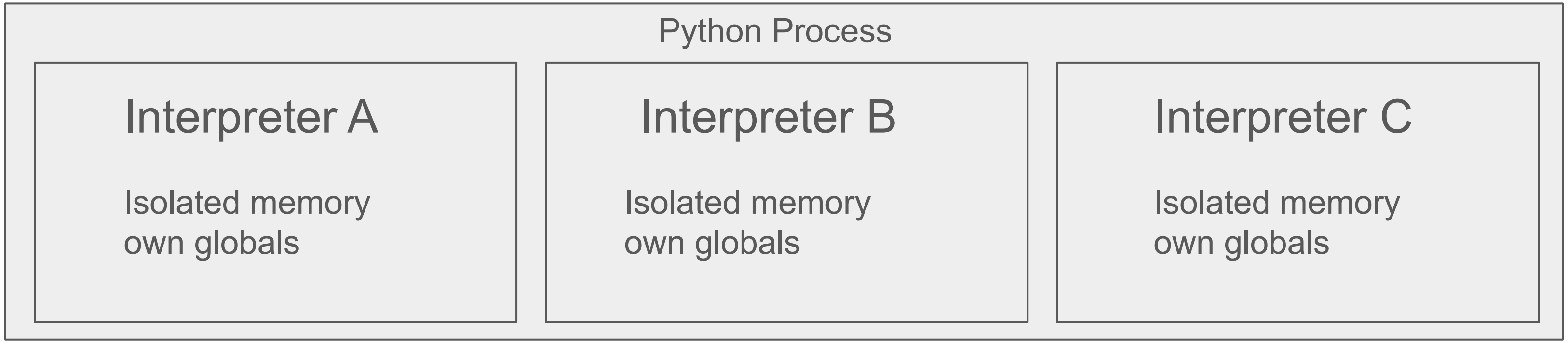
This feature contributes to enrich the “parallelism” within Python:

new type of concurrency which differs from free-threading.

- introduction of “**concurrent.interpreters**” module
- evolution of PEP 684 (sub-interpreters)
- similar to *multiprocessing* but with optimized (lower) memory overhead
- CPU bound safe and isolated

PEP 734 – Multiple Interpreters in the Stdlib

- **isolated** I. but within the same python process
- each I. has **its own memory space** and modules: none object is shared among interpreters
  - communication between I. via “**interpreters.Queue**”
- all the I. use the same libraries installed at OS/env level



Free-threaded mode improvements

What does “free-threaded” mean?

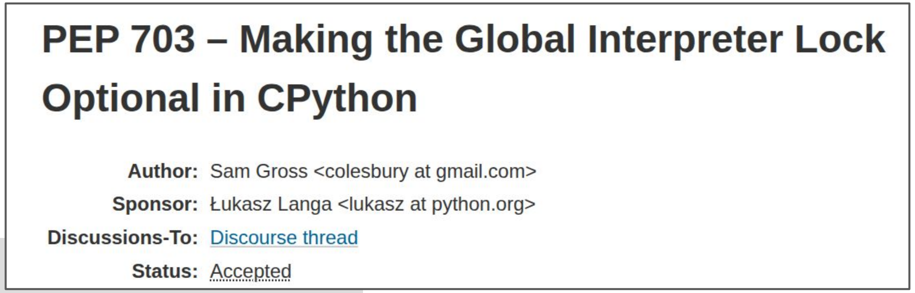
> Python bytecode executed in parallel threads by multiple CPU cores

no-GIL **thread-safety**

Free-threaded mode improvements

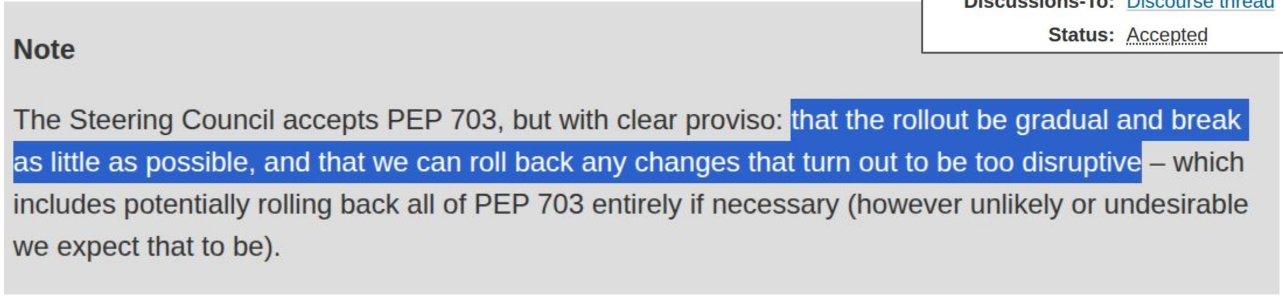
Perfection is a trade-off

- free-threaded is not the default option
- **no-GIL** is still optional and a “compile-time” option



PEP 703 – Making the Global Interpreter Lock Optional in CPython

Author:	Sam Gross <colesbury at gmail.com>
Sponsor:	Łukasz Langa <lukasz at python.org>
Discussions-To:	Discourse thread
Status:	Accepted



Note

The Steering Council accepts PEP 703, but with clear proviso: that the rollout be gradual and break as little as possible, and that we can roll back any changes that turn out to be too disruptive – which includes potentially rolling back all of PEP 703 entirely if necessary (however unlikely or undesirable we expect that to be).

How did we get here

milestones

Python 3.12 :

PEP 683 – Immortal Objects, Using a Fixed Refcount

PEP 684 – A Per-Interpreter GIL

Python 3.13:

PEP 703 – Making the Global Interpreter Lock Optional in CPython (experimental)

Python 3.14:

PEP 703 – Making the Global Interpreter Lock Optional in CPython (finalized)

PEP 659 – Specializing Adaptive Interpreter (revisited + thread-safe)

PEP 779 – Criteria for supported status for free-threaded Python

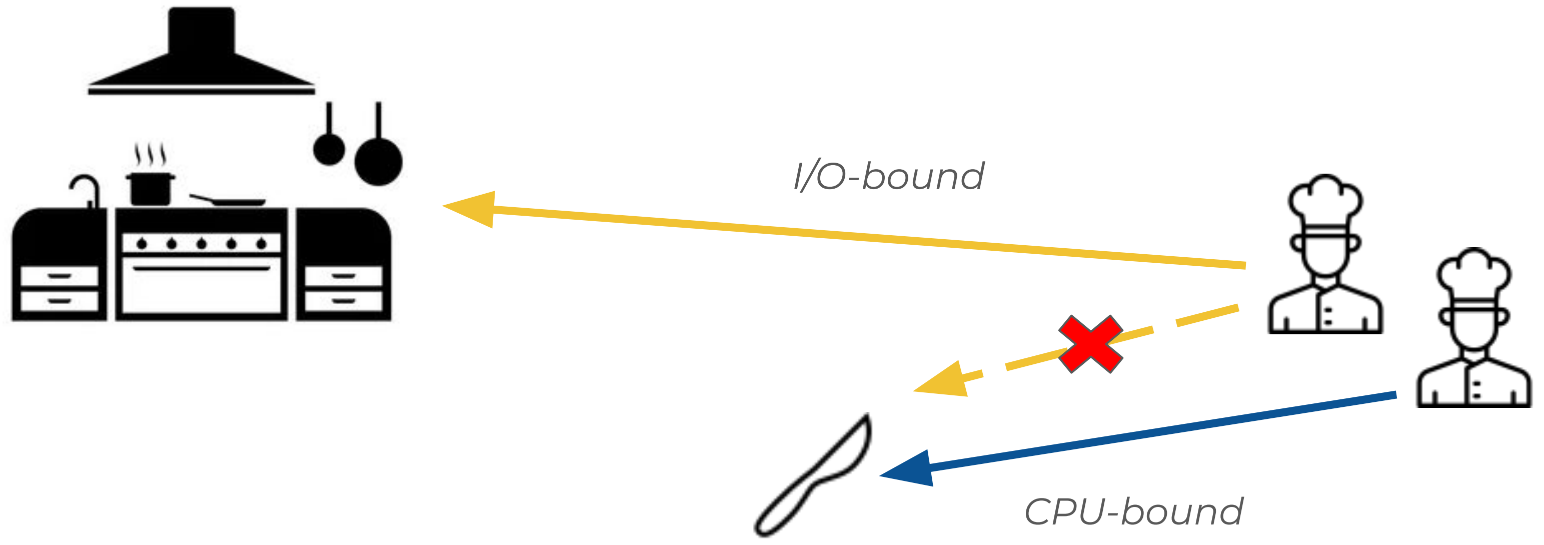
Free-Threading paradigm

GIL or no-GIL, that *was* the question...

...what’s the Global Interpreter Lock then?

“within a single CPython process, only one thread can execute Python bytecode at any given time”

Free-Threading paradigm



Python process

GIL

Threads

Free-Threading paradigm

The issue:

Since multiple threads **share the same memory** and *I/O*, their parallel run might raise “thread racing” problems.

The “solution”:

For decades (early ‘90s)GIL “locked” threads parallel execution in order to give consistency to memory usage: **only one thread to run at one time**. When one thread requests to start working, it will lock the other one.

Therefore, although multiple threads run together, they cannot make use of the multiple CPU cores at the same time.

Free-Threading paradigm

The fix (after several years and some keystones achieved):

- all core objects (lists, dicts, modules) **have been made thread-safe**: internal access is protected
-
- a large portion of global state has been **isolated per thread** or **per interpreter** (PEP 684)
-
- shared **immutable objects** (e.g., None, integers, functions) have **become immortal** (PEP 683) and no locks needed to use them
-
- **GIL can be disabled** (PEP 703) without any consequence (this is a lie)

Free-Threading paradigm

With a no-GIL run we have:

= no major improvements with I/O bound (database is still a bottleneck anyway)

↗ major improvements with CPU-bound: threads execution in parallel, spread to all the available CPU cores; CPU is not anymore a bottleneck when multiple cores can run python code together

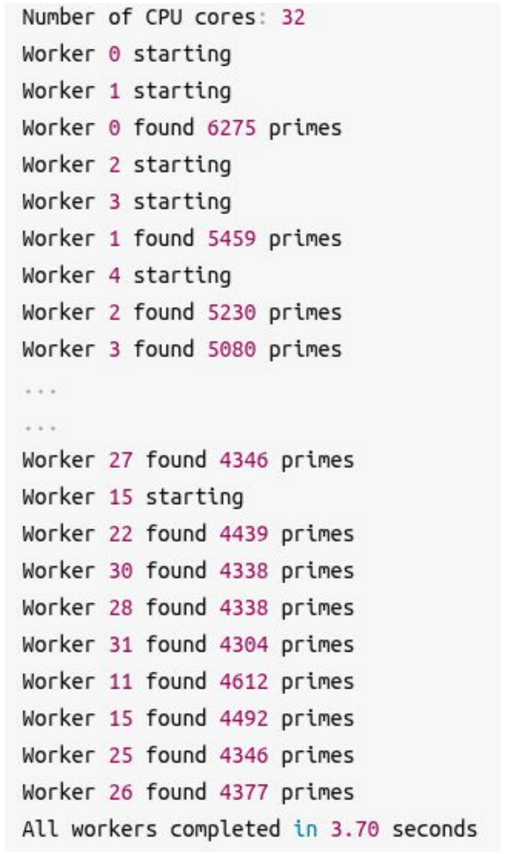
Free-Threading paradigm

Workload	Process	Performance(Free-Threaded 3.14)
CPU-Bound (Pure Python)	Matrix Multiplication (custom)	Speedup ~10x (44s > 4.6s)
CPU-Bound (Pure Python)	Prime Number Calculation	Speedup ~10x (3.7s > 0.35s)
Orchestration (Pandas-like)	Applying a Python Function (row-wise)	Time Reduction of 60-90%
C-Optimized Libraries	numpy.dot()	No Significant Change (already released the GIL)

Free-Threading paradigm

Finding prime numbers:

Regular Python with GIL



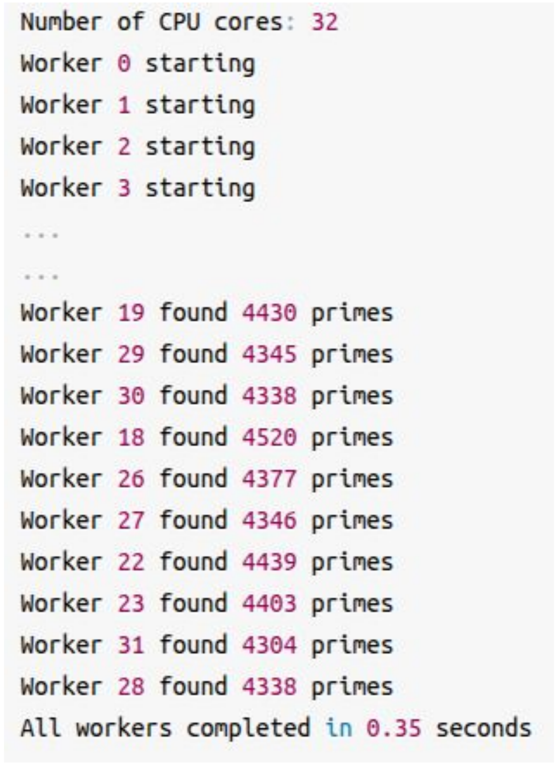
```
Number of CPU cores: 32
Worker 0 starting
Worker 1 starting
Worker 0 found 6275 primes
Worker 2 starting
Worker 3 starting
Worker 1 found 5459 primes
Worker 4 starting
Worker 2 found 5230 primes
Worker 3 found 5080 primes
...
...
Worker 27 found 4346 primes
Worker 15 starting
Worker 22 found 4439 primes
Worker 30 found 4338 primes
Worker 28 found 4338 primes
Worker 31 found 4304 primes
Worker 11 found 4612 primes
Worker 15 found 4492 primes
Worker 25 found 4346 primes
Worker 26 found 4377 primes
All workers completed in 3.70 seconds
```

source: https://towardsdatascience.com/python-3-14-and-the-end-of-the-gil/

Free-Threading paradigm

Finding prime numbers:

Regular Python without GIL



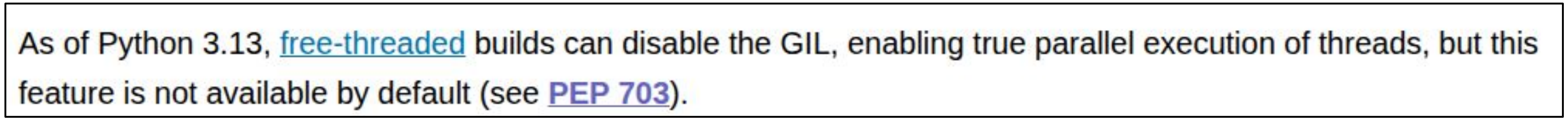
```
Number of CPU cores: 32
Worker 0 starting
Worker 1 starting
Worker 2 starting
Worker 3 starting
...
...
Worker 19 found 4430 primes
Worker 29 found 4345 primes
Worker 30 found 4338 primes
Worker 18 found 4520 primes
Worker 26 found 4377 primes
Worker 27 found 4346 primes
Worker 22 found 4439 primes
Worker 23 found 4403 primes
Worker 31 found 4304 primes
Worker 28 found 4338 primes
All workers completed in 0.35 seconds
```

source: https://towardsdatascience.com/python-3-14-and-the-end-of-the-gil/

Impacts on Threading & Multiprocessing

Threading:

“The threading module operates within a single process, [...] all threads share the same memory space. However, the GIL limits the performance gains of threading when it comes to CPU-bound tasks, as only one thread can execute Python bytecode at a time. Despite this, threads remain a useful tool for achieving concurrency in many scenarios.”



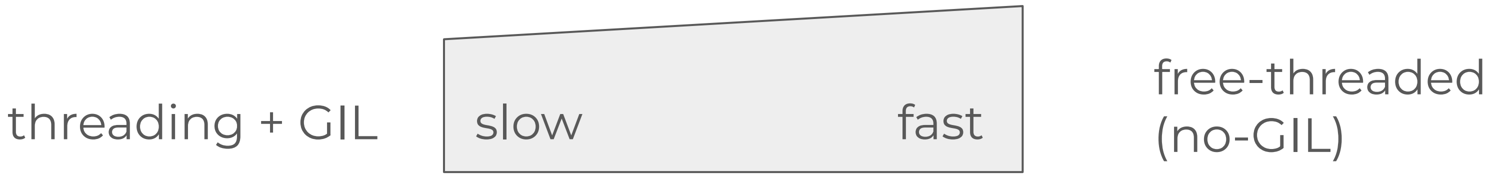
As of Python 3.13, free-threaded builds can disable the GIL, enabling true parallel execution of threads, but this feature is not available by default (see PEP 703).

source: https://docs.python.org/3/library/threading.html

Impacts on Threading & Multiprocessing

Threading is good for I/O operations, but not for CPU-bounds: **GIL still locks threads**.

With “threading” we define an extremely fast interleaving among running threads where “**context switching” appears to users if these are progressing simultaneously** >>> virtual/faked parallelism



Impacts on Threading & Multiprocessing

Multiprocessing:

“Multiprocessing is a package that supports spawning processes [...]. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using sub-processes instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.”

Impacts on Threading & Multiprocessing

Multiprocessing is good for CPU bound operations, having different processes each with a Python interpreter and its own GIL.

Offers real parallelism, but is slower (than threading), higher resource usage, and even less efficient for smaller tasks.

Free-threaded python does not significantly improve multiprocessing performance, but it affects resource consumption. With python 3.14 multiprocessing remains the preferred module when isolation (memory and resources) is a mandatory requirement. *

* remember “**concurrent.interpreters**”

Impact of no-GIL in data science

Can this free-threading help "data" world? Yes, **but...**

- benchmarks show a **10x speedup** on CPU-bounds operations
- pure Python intensive tasks will gain lot of benefits
- all the major "AI" **algorithms performance** will benefit of free-threading

ML, DL and Data Science workloads obtain linear (based on number of cores) benefits

Impact of no-GIL in data science

...but... benefits will mainly impact the “glue code” surrounding data processing:

- *NumPy*, *Pandas*, *PyTorch*, *TensorFlow* libs **>>>** C, C++, Fortran

  😑

- for I/O they **already** release GIL

  😯

- for CPU/GPU-bound operations they start **working multi-threaded**

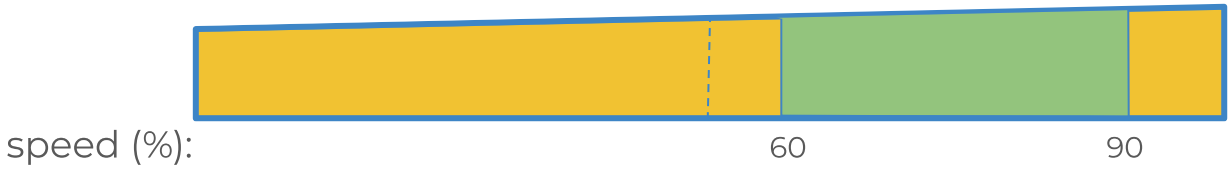
  😊

Impact of no-GIL in data science

e.g.

Pandas “fd.apply” is slow because it applies an operation “foobar” to all the rows, that is limited by GIL. The “foobar” func can be parallelized multi-threaded with no-GIL, applying it to chunks of data frame.

The improvements will be clearly noticeable, with a time reduction exceeding 60%



Impact of no-GIL in web frameworks

All the new features in Python 3.14 are also promising for web development and backend servers:

fastAPI

- throughput increased
- latency reduced (no more waits for slow requests)
- performance and quality enhanced

django

- throughput and performance increased
- from multi-process (workers) to multi-threads
- scaling up guaranteed
- memory consumption reduced
- I/O requests still slow due to DB bottleneck

Impact of no-GIL in web frameworks

Free-threaded applied to **fastAPI**:

throughput :

AsyncIO handles concurrency per single thread; if there is something CPU-consuming, the Event Loop stops and waits forever.

With no-GIL we would keep the Event Loop one-thread and dispatch all the CPU-bound tasks to other threads which would run in parallel **without blocking the Event Loop**.

scalability:

Without GIL it becomes easier to scale vertically (e.g. more CPU cores) without any need of using multiple workers or other complex external processes.

Impact of no-GIL in web frameworks

Free-threaded applied to **Django**:

memory optimization (game-changer improvement):

> The multiprocess approach with django and Gunicorn has a **huge memory** (RAM) **consumption** since we need to declare several workers (one for each CPU), *even multi-threaded*, where each worker loads the whole django app.
>
> The total amount of RAM is calculated multiplying the size of the app with the number of workers.

By removing the GIL, Python threads can finally execute code on different CPU cores, unlocking true parallelism **within a single process**. No-GIL transforms Django's workers from "processes which perform limited concurrent work" into "processes performing truly parallel work".

Conclusions

Road to Parallelism...

Python 3.14 is not just an update: it's a **re-architecture**.

Python 3.14 is not the end of the story: **it's the beginning**.

The GIL problem has been technically solved; now the work of adoption by the ecosystem begins.

For Data Science, it unlocks parallelism in your "glue code"; For the Web, it changes the rules of deployment.

The ecosystem (C extensions) is adapting, and this is the biggest challenge.

Conclusions

Road to Parallelism...

The future of Python is parallel. And it has just begun.

Slides and Resources:

https://github.com/Ardenine/road-to-parallelism/