Beyond the GIL: Python's Journey to Free Threading

RND Light Talk, July 2025

Yonatan Bitton
@bityob

Timeline

- 1991: Python 0.9.0

Timeline

- 1991: Python 0.9.0
- 1992: Python: threads and the GIL

Timeline

- 1991: Python 0.9.0
- 1992: Python: threads and the GIL
- 1996: "free threading" patch (Greg Stein)

Timeline

- 1991: Python 0.9.0
- 1992: Python: threads and the GIL
- 1996: "free threading" patch (Greg Stein)
- 2007: python-safethread (Adam Olsen)

Timeline

- 1991: Python 0.9.0
- 1992: Python: threads and the GIL
- 1996: "free threading" patch (Greg Stein)
- 2007: python-safethread (Adam Olsen)
- 2016: Gilectomy (Larry Hastings)

Timeline

- 2021: nogil (Sam Gross)

Timeline

- 2021: nogil (Sam Gross)
- 2023: PEP 703 published (Jan 2023)

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
Type:	Standards Track
Created:	09-Jan-2023
Python-Version:	3.13
Post-History:	09-Jan-2023, 04-May-2023
Resolution:	24-Oct-2023

Source: https://peps.python.org/pep-0703

Timeline

Timeline

- 2023: Meta supports and commits three engineer-years through 2025

A fast, free threading Python

Ideas

Carl Meyer carljm CPython core developer Jul 2023

> Guido van Rossum:
>
> it would be great if Meta or another tech company could spare some engineers with established CPython internals experience to help the core dev team with this work.

We've had a chance to discuss this internally with the right people. Our team believes in the value that nogil will provide, and we are committed to working collaboratively to improve Python for everyone.

If PEP 703 is accepted, Meta can commit to support in the form of three engineer-years (from engineers experienced working in CPython internals) between the acceptance of PEP 703 and the end of 2025, to collaborate with the core dev team on landing the PEP 703 implementation smoothly in CPython and on ongoing improvements to the compatibility and performance of nogil CPython.

Source: https://discuss.python.org/t/a-fast-free-threading-python/27903/99

Timeline

- 2023: PEP 703 **accepted** with clear proviso (Oct 2023)

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).

Source: https://peps.python.org/pep-0703

Timeline

- 2024: Python 3.13 with **experimental** support

Timeline

- 2024: Python 3.13 with **experimental** support
- 2025: PEP 779 - Criteria for supported status for free-threaded Python

Timeline

- 2024: Python 3.13 with **experimental** support
- 2025: PEP 779 - Criteria for supported status for free-threaded Python
- 2025: Python 3.14.0b3 released in Jun 2025

Timeline

- 2024: Python 3.13 with **experimental** support
- 2025: PEP 779 - Criteria for supported status for free-threaded Python
- 2025: Python 3.14.0b3 released in Jun 2025
    - Not **experimental**, not yet the default build

Timeline

- 2024: Python 3.13 with **experimental** support
- 2025: PEP 779 - Criteria for supported status for free-threaded Python
- 2025: Python 3.14.0b3 released in Jun 2025
    - Not **experimental**, not yet the default build
- 2030~: Python default build?

> ***The free-threaded mode is experimental*** *and work is ongoing to improve it: expect some bugs and a substantial single-threaded performance hit.*

Source: https://docs.python.org/3/howto/free-threading-python.html

So… what is this all about?

So… what is this all about?

- Python is fast to write — but not always fast to run

So… what is this all about?

- Python is fast to write — but not always fast to run
- Especially limited when using **threads** on multi-core machines

So… what is this all about?

- Python is fast to write — but not always fast to run
- Especially limited when using **threads** on multi-core machines
- The **GIL** has been a long-standing pain point

So… what is this all about?

- Python is fast to write — but not always fast to run
- Especially limited when using **threads** on multi-core machines
- The **GIL** has been a long-standing pain point
- Now, after years of effort, **free-threaded Python** is becoming real

So… what is this all about?

- Python is fast to write — but not always fast to run
- Especially limited when using **threads** on multi-core machines
- The **GIL** has been a long-standing pain point
- Now, after years of effort, **free-threaded Python** is becoming real
- Let's talk about what it means — and why it's exciting

The Global Interpreter Lock (GIL)

The Global Interpreter Lock (GIL)

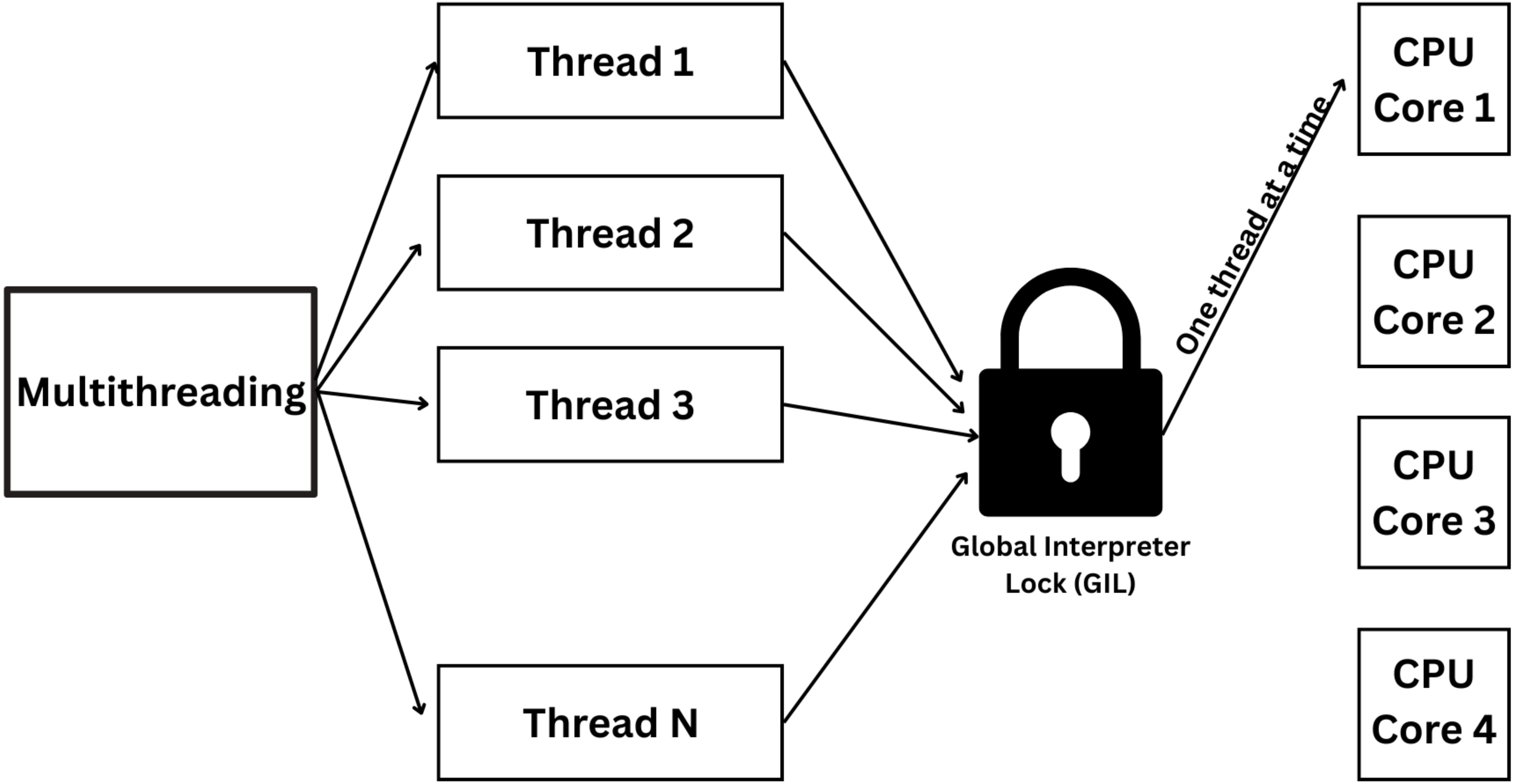
- Global lock in CPython

The Global Interpreter Lock (GIL)

- Global lock in CPython
- Blocks true multi-core parallelism

The Global Interpreter Lock (GIL)

- Global lock in CPython
- Blocks true multi-core parallelism
- Key reason behind Python’s single-thread limitations



Source: https://www.codecademy.com/article/understanding-the-global-interpreter-lock-gil-in-python

```
$ uv python install 3.14t
Installed Python 3.14.0a6 in 15ms
 + cpython-3.14.0a6+freethreaded-linux-x86_64-gnu
```

```python
import time
from concurrent.futures import ThreadPoolExecutor

N = 50_000_000
THREADS = 10

def work(n):
    for _ in range(n):
        pass

start = time.time()

with ThreadPoolExecutor(max_workers=THREADS) as pool:
    list(pool.map(work, [N] * THREADS))

```

```python
from concurrent.futures import ThreadPoolExecutor

N = 50_000_000
THREADS = 10

def work(n):
    for _ in range(n):
        pass

start = time.time()

with ThreadPoolExecutor(max_workers=THREADS) as pool:
    list(pool.map(work, [N] * THREADS))

print(f"Done in {time.time() - start:.2f} sec")
```

```
$ uv run --python=3.14t python demo.py
Done in 2.21 sec
$ uv run --python=3.14 python demo.py
Done in 7.90 sec
```

So... why the GIL?

- Simpler memory management

So... why the GIL?

- Simpler memory management
- Ensures thread safety in CPython

So... why the GIL?

- Simpler memory management
- Ensures thread safety in CPython
- Made sense in single-core era (Moore's law etc.)

GIL and Thread Safety

- Prevents data corruption

GIL and Thread Safety

- Prevents data corruption
- Protects internal Python objects

GIL and Thread Safety

- Prevents data corruption
- Protects internal Python objects
- Not all race conditions eliminated

The GIL in the Multi-Core Era

- Modern CPUs → multi-core

The GIL in the Multi-Core Era

- Modern CPUs → multi-core
- GIL bottlenecks CPU-bound apps

The GIL in the Multi-Core Era

- Modern CPUs → multi-core
- GIL bottlenecks CPU-bound apps
- Wastes hardware potential

Bypassing the GIL

- I/O-bound tasks → GIL released

Bypassing the GIL

- I/O-bound tasks → GIL released
- C extensions release GIL

Bypassing the GIL

- I/O-bound tasks → GIL released
- C extensions release GIL
- Asyncio → concurrency in one thread

The Goal — Free-Threaded Python

- **True** multi-core parallelism

The Goal — Free-Threaded Python

- **True** multi-core parallelism
- **Preserve** single-thread speed

The Goal — Free-Threaded Python

- **True** multi-core parallelism
- **Preserve** single-thread speed
- Unlock Python’s full performance

Challenges

Challenges

- Reference Counting

Challenges

- Reference Counting
- GC

Challenges

- Reference Counting
- GC
- Locking and atomic APIs

Challenges

- Reference Counting
- GC
- Locking and atomic APIs
- Containers

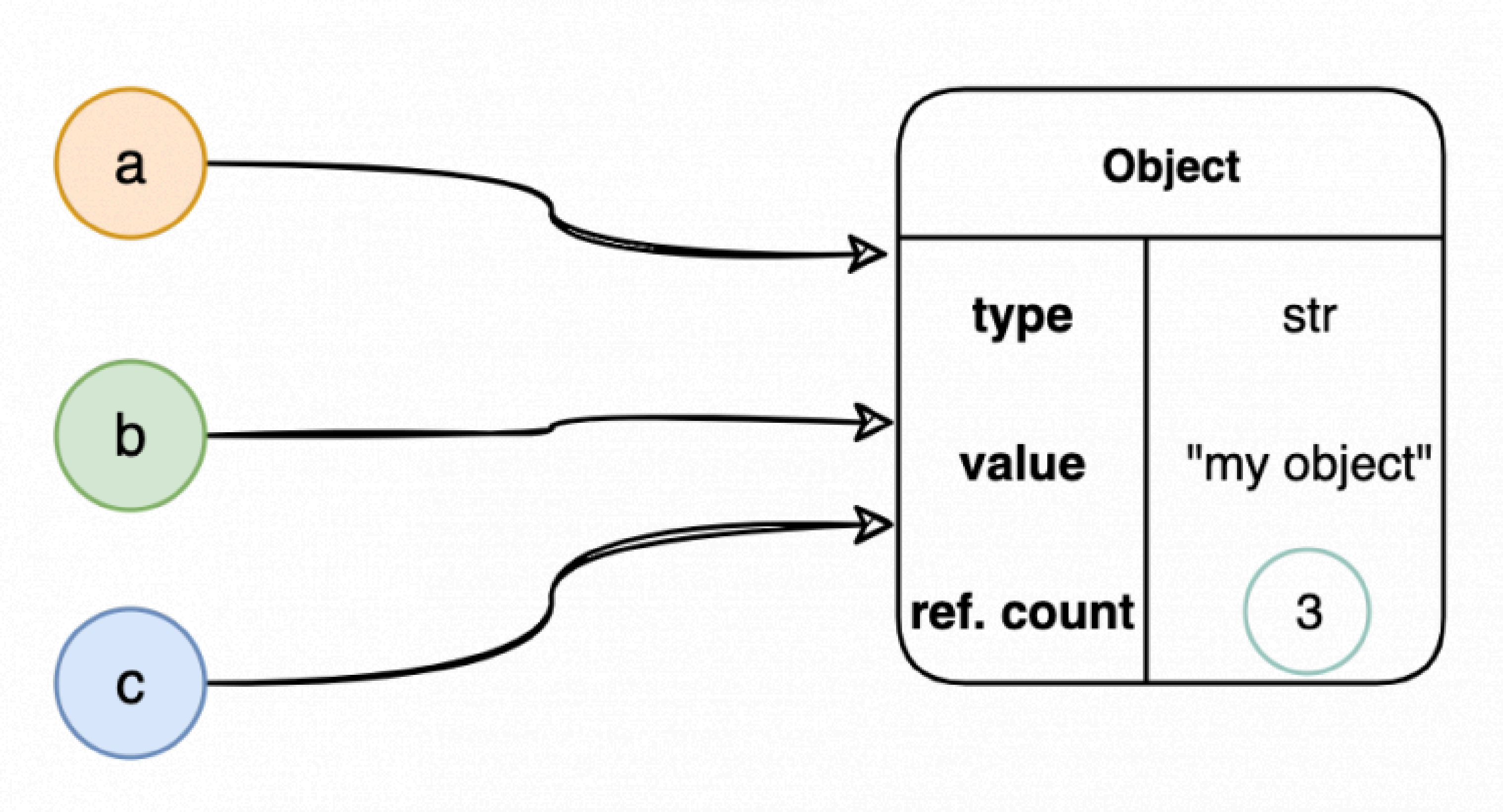
Reference Counting

Reference Counting

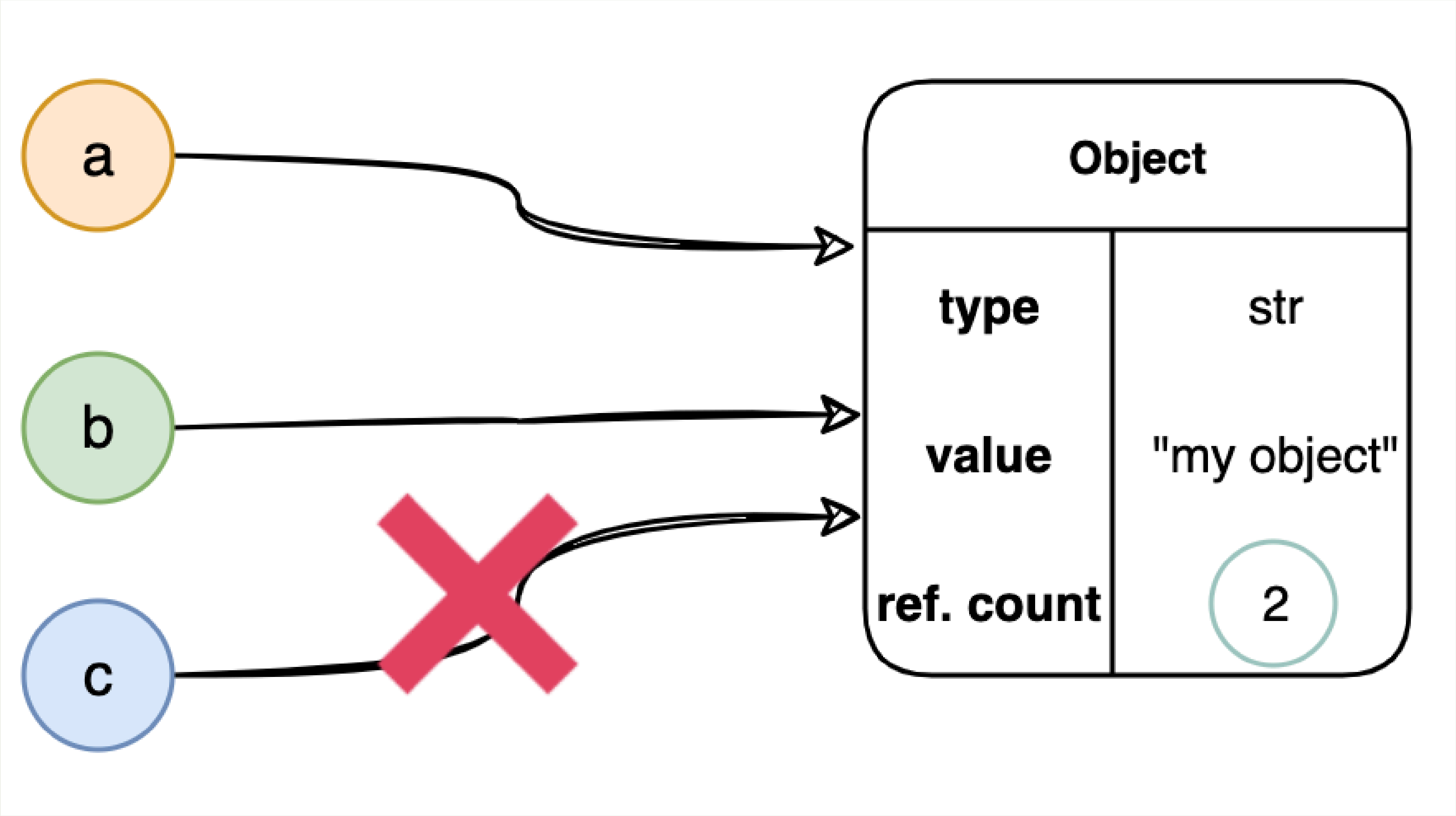
- Reference count goes up with new references, down when deleted or reassigned

Reference Counting

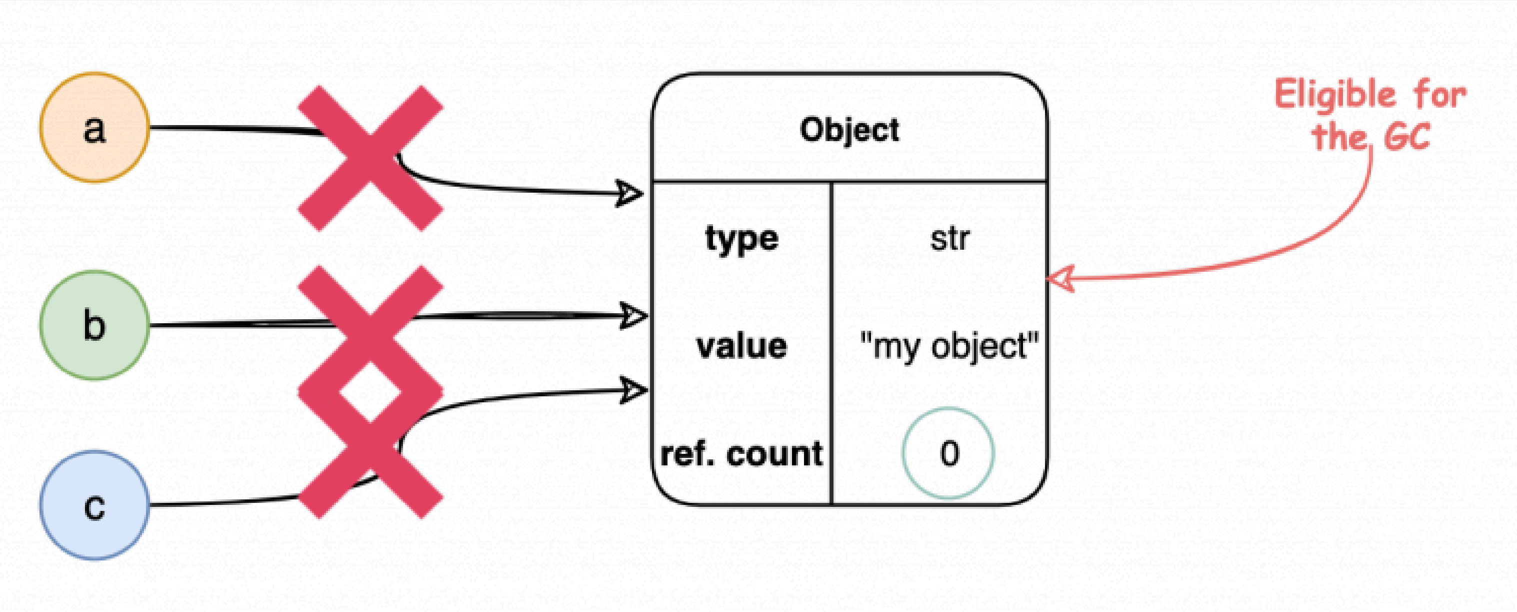
- Reference count goes up with new references, down when deleted or reassigned
- When it hits zero — the object is immediately freed



Source: https://jenaiz.com/2022/02/understanding-how-the-python-garbage-collector-works/



Source: https://jenaiz.com/2022/02/understanding-how-the-python-garbage-collector-works/



Source: https://jenaiz.com/2022/02/understanding-how-the-python-garbage-collector-works/

Why Reference Counting Is a Problem

- Python tracks memory via reference counts

Why Reference Counting Is a Problem

- Python tracks memory via reference counts
- GIL protects those updates today

Why Reference Counting Is a Problem

- Python tracks memory via reference counts
- GIL protects those updates today
- Without GIL → race conditions

Biased Reference Counting

Biased Reference Counting

- Most objects are accessed by just one thread

Biased Reference Counting

- Most objects are accessed by just one thread
- Optimized for single-threaded access

Biased Reference Counting

- Most objects are accessed by just one thread
- Optimized for single-threaded access
- Fast for owning thread

Biased Reference Counting

- Most objects are accessed by just one thread
- Optimized for single-threaded access
- Fast for owning thread
- Atomic ops only when needed

```c
struct _object {
  _PyObject_HEAD_EXTRA
  uintptr_t ob_tid;          // owning thread id (4-8 bytes)
  uint16_t __padding;        // reserved for future use (2 bytes)
  PyMutex ob_mutex;          // per-object mutex (1 byte)
  uint8_t ob_gc_bits;        // GC fields (1 byte)
  uint32_t ob_ref_local;     // local reference count (4 bytes)
  Py_ssize_t ob_ref_shared; // shared reference count and state bits (4-8 bytes)
  PyTypeObject *ob_type;
};
```

```c
struct _object {
  _PyObject_HEAD_EXTRA
  uintptr_t ob_tid;         // owning thread id (4-8 bytes)
  uint16_t __padding;       // reserved for future use (2 bytes)
  PyMutex ob_mutex;         // per-object mutex (1 byte)
  uint8_t ob_gc_bits;       // GC fields (1 byte)
  uint32_t ob_ref_local;    // local reference count (4 bytes)
  Py_ssize_t ob_ref_shared; // shared reference count and state bits (4-8 bytes)
  PyTypeObject *ob_type;
};
```

```c
struct _object {
  _PyObject_HEAD_EXTRA
  uintptr_t ob_tid;         // owning thread id (4-8 bytes)
  uint16_t __padding;       // reserved for future use (2 bytes)
  PyMutex ob_mutex;         // per-object mutex (1 byte)
  uint8_t ob_gc_bits;       // GC fields (1 byte)
  uint32_t ob_ref_local;    // local reference count (4 bytes)
  Py_ssize_t ob_ref_shared; // shared reference count and state bits (4-8 bytes)
  PyTypeObject *ob_type;
};
```

Immortal Objects

Immortal Objects

- Common objects are never deallocated

Immortal Objects

- Common objects are never deallocated
  - `True`, `False`, `None`, small integers, etc.

Immortal Objects

- Common objects are never deallocated
  - `True`, `False`, `None`, small integers, etc.
- Skip reference count updates

Immortal Objects

- Common objects are never deallocated
  - `True`, `False`, `None`, small integers, etc.
- Skip reference count updates
- Reduce thread contention

Deferred Reference Counting

Deferred Reference Counting

- Skip reference count changes for hot objects

Deferred Reference Counting

- Skip reference count changes for hot objects
    - Reference counting on every access is inefficient

Deferred Reference Counting

- Skip reference count changes for hot objects
    - Reference counting on every access is inefficient
- GC handles cleanup later

Deferred Reference Counting

- Skip reference count changes for hot objects
    - Reference counting on every access is inefficient
- GC handles cleanup later
- Used for code, functions, modules

Garbage Collection (GC)

Garbage Collection (GC)

- Detects and collects **cyclic references** that reference counting misses

Garbage Collection (GC)

- Detects and collects **cyclic references** that reference counting misses
- Uses **generational GC**: objects are grouped by age; older ones are scanned less often

GC without GIL

GC without GIL

- Can’t rely on the GIL to **synchronize memory access** — GC must now be **explicitly thread-safe**

GC without GIL

- Can’t rely on the GIL to **synchronize memory access** — GC must now be **explicitly thread-safe**
- “Stop-the-world” GC needed

GC without GIL

- Can’t rely on the GIL to **synchronize memory access** — GC must now be **explicitly thread-safe**
- “Stop-the-world” GC needed
    - All threads pause during collection to avoid race conditions

GC without GIL

GC without GIL

- Single-generation GC

GC without GIL

- Single-generation GC
  - Using a **single generation** avoids frequent pauses

GC without GIL

- Single-generation GC
  - Using a **single generation** avoids frequent pauses
  - **Generational GC is less critical** in Python — young objects are usually cleaned up early via **reference counting**

GC without GIL

- Single-generation GC
  - Using a **single generation** avoids frequent pauses
  - **Generational GC is less critical** in Python — young objects are usually cleaned up early via **reference counting**
- Integration with deferred and biased reference counting

Locking and atomic APIs

Locking and atomic APIs

- Global locks like the **GIL** add overhead but **no deadlocks** if only one lock exists

Locking and atomic APIs

- Global locks like the **GIL** add overhead but **no deadlocks** if only one lock exists
- Using **multiple locks** can cause **deadlocks** (e.g., double locks)

Locking and atomic APIs

- Global locks like the **GIL** add overhead but **no deadlocks** if only one lock exists
- Using **multiple locks** can cause **deadlocks** (e.g., double locks)
- Python uses **critical sections locked on individual objects**

Locking and atomic APIs

- Global locks like the **GIL** add overhead but **no deadlocks** if only one lock exists
- Using **multiple locks** can cause **deadlocks** (e.g., double locks)
- Python uses **critical sections locked on individual objects**
  - Lock acquired only when accessing that object

Locking and atomic APIs

- Global locks like the **GIL** add overhead but **no deadlocks** if only one lock exists
- Using **multiple locks** can cause **deadlocks** (e.g., double locks)
- Python uses **critical sections locked on individual objects**
    - Lock acquired only when accessing that object
    - Released if thread needs another lock or pauses (e.g., for blocking I/O)

```c
struct _object {
  _PyObject_HEAD_EXTRA
  uintptr_t ob_tid;         // owning thread id (4-8 bytes)
  uint16_t __padding;       // reserved for future use (2 bytes)
  PyMutex ob_mutex;         // per-object mutex (1 byte)
  uint8_t ob_gc_bits;       // GC fields (1 byte)
  uint32_t ob_ref_local;    // local reference count (4 bytes)
  Py_ssize_t ob_ref_shared; // shared reference count and state bits (4-8 bytes)
  PyTypeObject *ob_type;
};
```

```c
Py_BEGIN_CRITICAL_SECTION(a->ob_mutex);
item = a->ob_item[i];
Py_INCREF(item);
Py_END_CRITICAL_SECTION(a->ob_mutex);
```

Locking and atomic APIs

Locking and atomic APIs

- **Read-write locks** are expensive and complex

Locking and atomic APIs

- **Read-write locks** are expensive and complex
  - Instead, Python prefers **try-fast approaches** with reference count checks

Locking and atomic APIs

- **Read-write locks** are expensive and complex
  - Instead, Python prefers **try-fast approaches** with reference count checks
  - Similar to **Read-Copy-Update (RCU)** pattern in Linux for efficient simple field access (e.g., `list[0]`)

Containers Thread-Safety

Containers Thread-Safety

- `list`, `set`, `dict`...

Containers Thread-Safety

- `list`, `set`, `dict`...
- Per-object locks replace GIL safety

Containers Thread-Safety

- `list`, `set`, `dict`...
- Per-object locks replace GIL safety
- Read operations may skip locks

Borrowed References

Borrowed References

- A **borrowed reference** is a pointer to a Python object you get **without** increasing its reference count

Borrowed References

- A **borrowed reference** is a pointer to a Python object you get **without** increasing its reference count
- To keep an object alive beyond immediate use, code must **convert borrowed to owned** by calling `Py_INCREF`

Borrowed References

Borrowed References

- Borrowed references were safe under the GIL
  → No other thread could mutate the object in between

Borrowed References

- Borrowed references were safe under the GIL
  → No other thread could mutate the object in between
- Now unsafe:
  → Another thread might free the object before `Py_INCREF`
  → Leads to **use-after-free** bugs

```c
// 'item' is a borrowed reference returned by PyList_GetItem
PyObject *item = PyList_GetItem(list, idx);

// Another thread might free 'item' here before we increment its refcount

// Trying to increment the refcount to keep 'item' alive
// ❌ Unsafe: 'item' might be already freed, leading to use-after-free
Py_INCREF(item);
```

```c
// 'item' is a borrowed reference returned by PyList_GetItem
PyObject *item = PyList_GetItem(list, idx);

// Another thread might free 'item' here before we increment its refcount

// Trying to increment the refcount to keep 'item' alive
// ❌ Unsafe: 'item' might be already freed, leading to use-after-free
Py_INCREF(item);
```

```c
// 'item' is a borrowed reference returned by PyList_GetItem
PyObject *item = PyList_GetItem(list, idx);

// Another thread might free 'item' here before we increment its refcount

// Trying to increment the refcount to keep 'item' alive
// ❌ Unsafe: 'item' might be already freed, leading to use-after-free
Py_INCREF(item);
```

Atomic “Fetch” APIs (Slower)

New APIs return **owned references** safely in one step

```
// ✅ Safe – Py_INCREF is done internally
PyObject *item = PyList_FetchItem(list, idx);`
```

Optimistic Fast Access

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability
- How it works:

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability
- How it works:
  - Atomically read slots and items

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability
- How it works:
    - Atomically read slots and items
    - Conditionally increment reference count (`Py_INCREF`) if safe

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability
- How it works:
  - Atomically read slots and items
  - Conditionally increment reference count (`Py_INCREF`) if safe
  - Verify no changes during access

Optimistic Fast Access

- Common list/dict accesses **avoid locks** for better speed and scalability
- How it works:
  - Atomically read slots and items
  - Conditionally increment reference count (`Py_INCREF`) if safe
  - Verify no changes during access
- If any check fails, **fallback** acquires a lock to ensure safety

Python’s Future Without the GIL

Python's Future Without the GIL

- Python can finally use many cores at once

Python's Future Without the GIL

- Python can finally use many cores at once
- Big speedups for multi-threaded programs

Python's Future Without the GIL

- Python can finally use many cores at once
- Big speedups for multi-threaded programs
- New memory designs keep things safe

🐍 Python's Future Without the GIL

- Python can finally use many cores at once
- Big speedups for multi-threaded programs
- New memory designs keep things safe
- Still the same Python we know

Python’s Future Without the GIL

- Python can finally use many cores at once
- Big speedups for multi-threaded programs
- New memory designs keep things safe
- Still the same Python we know
- **But:** Work still in progress, not all code will see speedups yet

Any Questions?

Yonatan Bitton @bityob

linktr.ee/bityob

Sources

- PEP 703 – Making the Global Interpreter Lock Optional in CPython (peps.python.org)
- PEP 779 – Criteria for supported status for free-threaded Python (peps.python.org)
- Python Free-Threading Guide (py-free-threading.github.io)

Learn More

- Multithreaded Python without the GIL - presented by Sam Gross (youtube.com)
- High-Performance Python: Faster Type Checking and Free Threaded Execution (youtube.com)
- Python Docs How To - Python experimental support for free (docs.python.org)