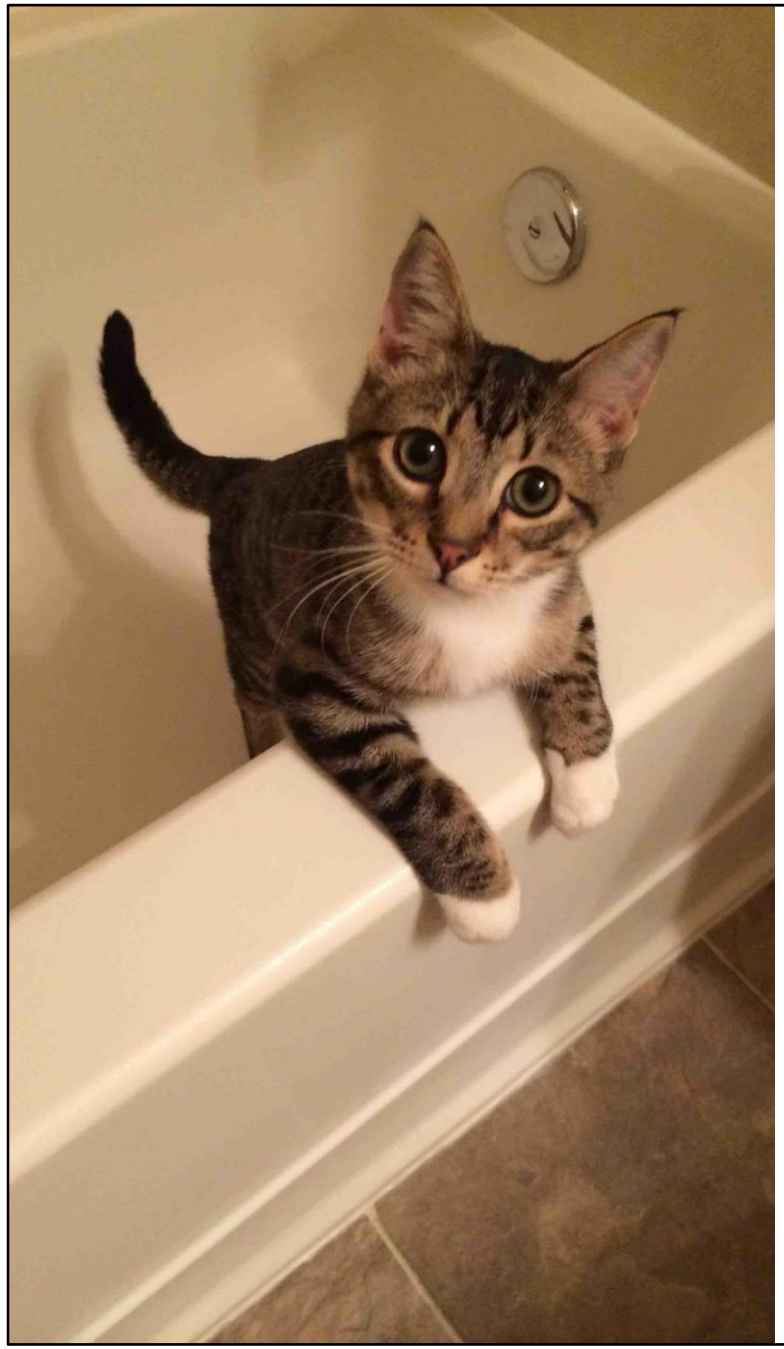
PARALLELISM AND THE GIL

What is this GIL thing that everybody loves to hate?

Laurie Opperman

Tonight's presentation

- Prerequisites (~8 minutes)
- All about the GIL (~7 minutes)
- Parallelism and concurrency in Python (~4 minutes)
- How the GIL interacts with parallelism (~5 minutes)



1. Prerequisites

- Memory management
  - Reference counting
- Intro to parallelism and concurrency
  - Difference between parallelism and concurrency
  - Data sharing
- Adding 1

Memory Management

- Programs have to ask for room to store their stuff (“allocate”)
- Programs should also say which memory their not using anymore (“deallocate”), or risk being killed by the OS or crashing when asking for more than is free
- Sounds tedious? There are techniques for automation, such as garbage-collection, so the virtual machine does memory management for you

Reference Counting

- Type of memory management
- Count the number of users who have access to a piece of data
- Automatically mark data as unreferenced (in Python, calls `__del__`), and deallocate
- Garbage-collection comes along and deallocates unreferenced data periodically, or when needed

Intro to Parallelism and Concurrency

- Sequential: follow a recipe one step at a time
- Concurrent: one person can switch between tasks while waiting
- Parallel: multiple people work on different parts of the recipe

1. Get ingredients
2. Make meringue
   a) Whisk eggs
   b) Whisk in sugar
   c) Bake
3. Whip cream
4. Cut fruit
5. Put fruit and cream on baked meringue

Parallelism vs Concurrency

- Concurrent: tasks don't have to be run one after the other
- Parallel: code is being executed at the same time
- You could say concurrency is necessary for parallelism (advanced technically not, eg SIMD)

1. Get ingredients
2. Make meringue
   a) Whisk eggs
   b) Whisk in sugar
   c) Bake
3. Whip cream
4. Cut fruit
5. Put fruit and cream on baked meringue

Data Sharing

- Threads: what's mine is yours. I can work on my own thing
- Processes: you can only have what I send over to you
- Locks/mutexes: I'm currently using this
- Code generally needs to be written to be "thread-safe"

Adding 1 – A Worked Example

```
a = 2
a = a + 1
a = a + 1
a == 4
```

- Sequential
- Each add is multiple steps
- Can only manipulate data in CPU memory ("registers")

```
1.  SET   2 -> R    # R = 2
2.  PUT   R -> a
3.  GET   a -> R    # R = 2
4.  SET   1 -> S    # S = 1
5.  ADD R,S -> T    # T = 3
6.  PUT   T -> a
7.  GET   a -> R    # R = 3
8.  SET   1 -> S    # S = 1
9.  ADD R,S -> T    # T = 4
10. PUT   T -> a
```

Adding 1 – A Worked Example

```
a = 2
a = a + 1   |   a = a + 1
a == 3
```

- Concurrent
- Oh oh, incorrect result

```
1.  SET   2 -> R    # R = 2
2.  PUT   R -> a
3.  GET   a -> R    # R = 2
4.  SET   1 -> S    # S = 1
5.  GET   a -> U    # U = 2
6.  ADD R,S -> T    # T = 3
7.  SET   1 -> V    # V = 1
8.  PUT   T -> a
9.  ADD U,V -> W    # W = 3
10. PUT   W -> a
```

2. The GIL (Global Interpreter Lock)

- How Python code is executed – a worked example
- Why GIL is necessary
- When GIL is released
- Feature of CPython

Python Code Execution – A Worked Example

```
def fn(a):
    return a.method()
b = MyObject()
fn(b)
```

- Modifying reference counts requires GIL to be held

1. Define `fn` and `b`
   1. Ensure GIL held
   2. Set reference count of `fn` and `b` to 1
2. Call `fn` with argument `b`
   1. Ensure GIL held
   2. Increase reference count of `b`
   3. Run `fn`
   4. Ensure GIL acquired
   5. Decrease reference count of `b`

Why the GIL is necessary

- Prevents reference counts from getting out of sync
  - There are also other parts of the internals of the interpreter which are not thread-safe
- Why not per-object locking?
  - Platform-native locks are slow, so per-object is slower than global (~2x)
  - Easier to develop CPython and exposed C-based objects
  - Backwards compatibility: many C-extensions are already written with the GIL

When the GIL is released

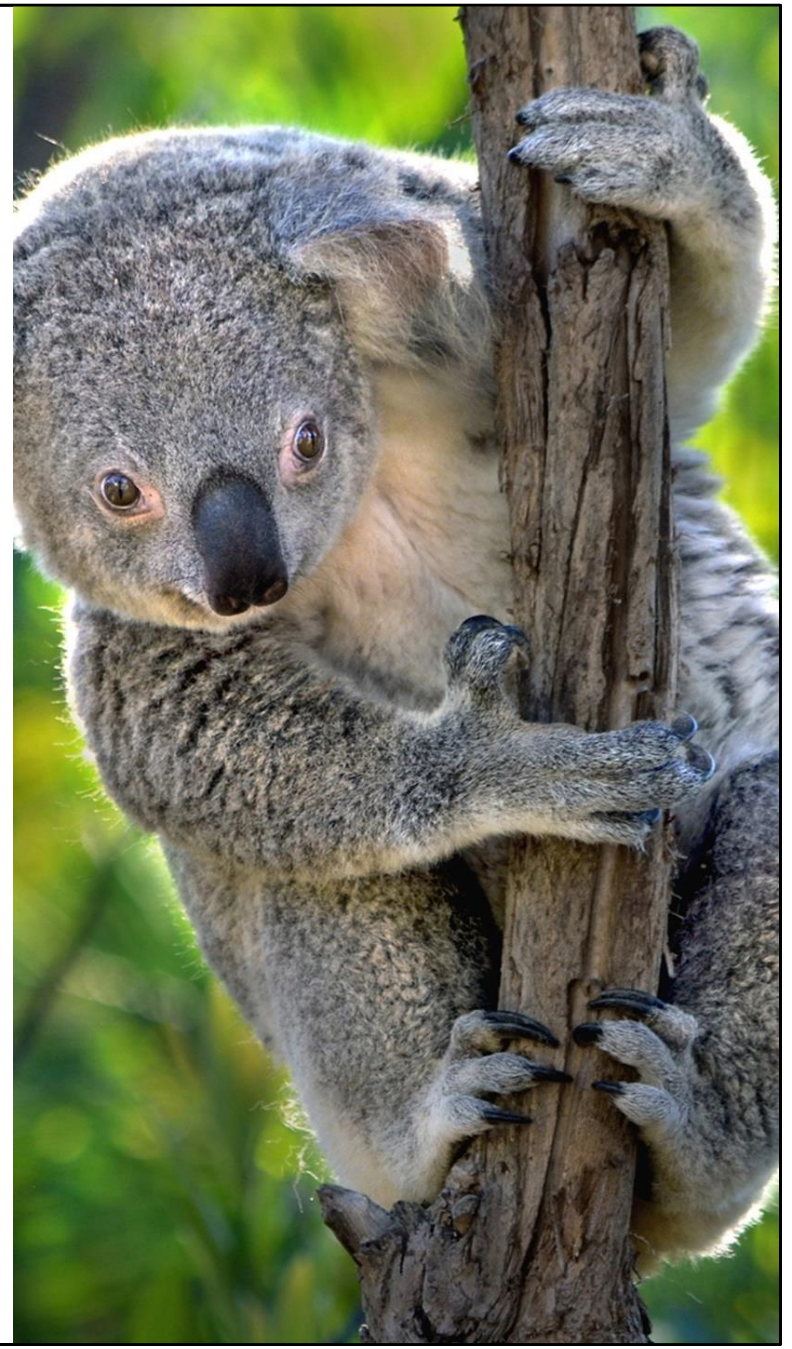
- I/O (input/output)
  - File read/write
  - Sockets/streams read/write and waiting for read/write
- C-extensions
  - Big array manipulation in NumPy
  - Compression/encryption
  - Python-controlled off-device computation (eg TensorFlow, Qiskit)
  - Any other library (standard or third-party) which says so
- Every now and then (context-switching)

Other Interpreters

- GIL is used in CPython (and PyPy, but differently)
  - Not part of the Python language specification
- In Cython, the GIL can be released by the user
- Jython and IronPython have no GIL!
  - The most popular Java virtual machine seems to have per-object locks, and uses path-based garbage collection
  - .NET (used by IronPython) is weird
- Other language's implementations have more sophisticated memory management than CPython

3. Parallelism and Concurrency in Python

- Threading
- Multiprocessing
- Coroutines (aka asynchronous functions)



Multi-Threading

- `threading` standard library
- Start a new thread: call a function in another thread
- Common objects are shared indiscriminately between threads
- Advanced: only have a new 'stack' for the thread

```
def send_heartbeats():
    while shared['continue']:
        send_heartbeat()
        time.sleep(5.0)


shared = {'continue': True}
heartbeat = threading.Thread(
    target=send_heartbeats)
heartbeat.start()
expensive_process_fn()
shared['continue'] = False
```

Multi-Processing

- `multiprocessing` standard library
  - Low-level process spawning in `os`
- Start a new process: call a function in another process
  - Smells like multi-threading!
- Has to copy the interpreter state, so RAM can become an issue
- Manual data sharing

```
def send_heartbeats(continue):
    while continue.value:
        send_heartbeat()
        time.sleep(5.0)


continue = mp.Value('B', 1)
heartbeat = mp.Process(
    target=send_heartbeats,
    args=(continue,))
heartbeat.start()
expensive_process_fn()
continue.value = 0
```

Coroutines

- Start a function, but don't wait for it to finish (we'll get, or wait for, the result later)
- Concurrency, not parallelism!
- Can be run in parallel using thread- or process-pool executors (in `concurrent.futures`)

```
async def fn():
    print('a', end='')
    await asyncio.sleep(1.0)
    print('b', end='')

async def main():
    await asyncio.gather(
        fn(), fn(), fn())

asyncio.run(main())
# prints: aaabbb
```

4. How the GIL Interacts with Parallelism

- When the GIL is really a problem in the real world
- When to use multi-threading, and when to use multi-processing

When the GIL Actually Causes Issues

- Lots of Python code being run
  - Pure-Python tree structures
  - List iteration, especially multi-dimensional lists
- Running Python code lots of times
  - Running a seemingly small analysis/control script for every web request
- Check your libraries: maybe they're pure-Python!

Multi-Threading vs Multi-Processing

- Interpreter doesn't care when other processes hold the GIL: there's a GIL for each process!
- That's about it

- Remember that GIL is released in C-extensions (NumPy) and I/O (file read/write or network communication), so multi-threading may work just as well

Conclusion

- GIL restricts Python code to being run in only one thread at a time
- GIL does not span across processes, so that can be used to your advantage
- GIL is released in computationally expensive C-extensions and blocking I/O
- GIL is a feature of only some implementations of Python (including the most common, CPython)

Further Reading

- Python documentation, specifically the C-API docs
- Performance considerations and actual GIL implementation: http://www.dabeaz.com/python/GIL.pdf
- Why no GIL in Jython or IronPython? StackOverflow