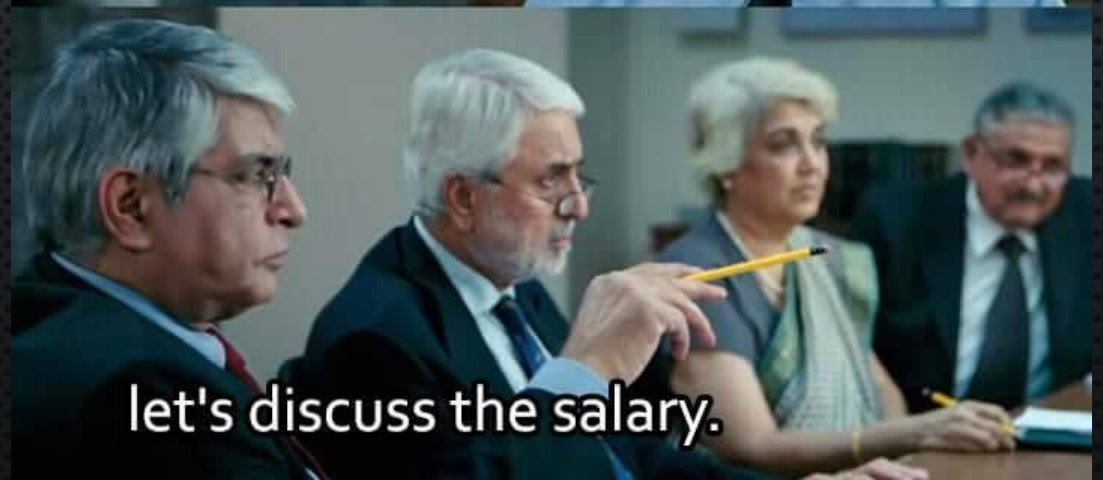
UNLOCK THE LOCK: MASTERING PYTHON MULTITHREADING DEADLOCKS BEFORE THEY MASTER YOU

About Me

- Principal Data Scientist @ Monarch Tractor Asia Pacific Pte Ltd
- Loves to learn through Communities (Slack, Discord)
- Teaches Bootcamps (Le Wagon)
- Writes
  - Medium: https://hanqi01.medium.com/
  - freeCodeCamp: https://www.freecodecamp.org/news/the-logic-philosophy-and-science-of-software-testing-handbook-for-developers/
- Plays Badminton, Piano, Drums, Chess, Sings
- Linkedin: https://www.linkedin.com/in/hanqi91 (QR code)
  - Please give your 1st time speaker some feedback!
- Code: https://github.com/gitgithan/pyconsg2025_multithreading
- Slides: Organizer will collect and share (also on github later)


IMPERIAL COLLEGE OF
Explain Deadlock, and we'll hire you.


Hire me, and I'll explain it to you.


let's discuss the salary.

Learning Objectives

- How to use VSCode Debugger
- Understand stepping behaviour
  - F10 (Step Over), F11 (Step In), Shift+F11 (Step Out), F5 (Continue)
- Understand Deadlock Spectrum
- See a Heisenbug
- Why use nested parallelism
- General Computer Science Concepts
  - Blocking
  - Decoupling with Queues
  - Synchronization with Locks
  - Deadlocks

Pre-requisites

- MultiThreading and Pre-emptive multitasking
- ThreadPoolExecutor
- Blocking vs Non-blocking code
- Locks

- Attention to Detail
- Curiosity
- Bias to action
- Comfortable switching between certainty and uncertainty

- Your worst case: You're completely lost, but gain transferrable VSCode debugging skills
- My worst case: Code deadlocks when it shouldn't, wing it for the rest of the talk

Talk Map

- PEP3148
- Deadlock Spectrum
- Script overview and Demo
- Deadlock Mechanics
- Debugging Techniques
- Deviation in Control Flow
- Influencing Script Termination
- Unswallow RunTimeError: The HeisenBug
- Overview of Solutions
- Nested Concurrency Patterns
- What else can you learn
- Bonus: 2 types of shutdown and Context Manager
- FAQ
- Resources

PEP3148

peps.python.org/pep-3148/#threadpoolexecutor

Page Source (GitHub)

ThreadPoolExecutor

The `ThreadPoolExecutor` class is an `Executor` subclass that uses a pool of threads to execute calls asynchronously.

Deadlock can occur when the callable associated with a `Future` waits on the results of another `Future`. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

Why this example?

- 5 lines, deceptively simple
- Uses ThreadPoolExecutor instead of lower level threading.Thread

Why would you ever do this?

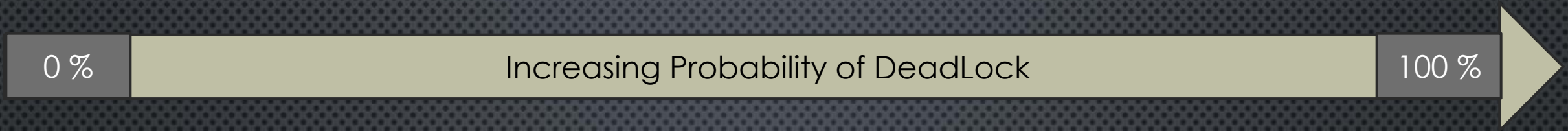
- No reason, usually
- Awareness of debugger behaviours
- Improved thinking to write better code, debug future problems

Pitfalls

- Possibly different behaviour when run in Jupyter Notebook

Whole talk using Python 3.10.15

Deadlock Spectrum



No Deadlock

Sometimes Deadlock

Always Deadlock

Discourage Deadlock

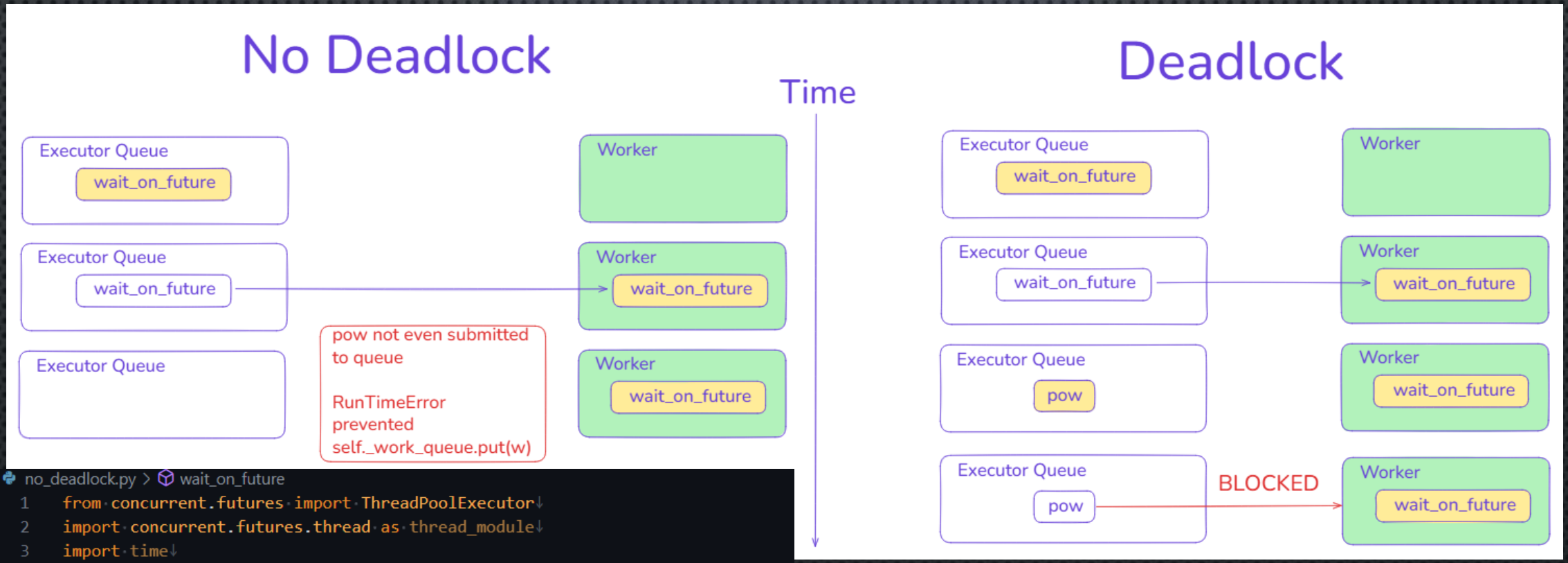
Force Deadlock

Script Overview and Demo

Module name	Behaviour
no_deadlock.py	No deadlock
sometimes_deadlock.py	More often deadlock than no deadlock
always_deadlock.py	Deadlock*
discourage_deadlock.py	More often no deadlock than deadlock
force_deadlock.py	Deadlock
unswallow_runtimeerror.py	No deadlock**

* During debugging, ensure enough time.sleep(seconds)
** RunTimeError visible on terminal

Deadlock Mechanics



no_deadlock.py > wait_on_future

```python
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures.thread as thread_module
import time


def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())


executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

Debugging Techniques

- Watch window
  - Single Expression
  - Auto updates (Saves time from typing repeatedly in Debug Console)
  - Defined or not depends on current scope
- Call Stack
  - Helps track what has been executed, and where to put earlier breakpoints
  - Shows when and how many threads/processes started
  - Shows which thread is breakpoint paused at (PAUSED ON BREAKPOINT), and which thread is paused because other threads are paused (PAUSED)
- Debug Console
  - REPL (as many expression as you want)
- Variables
  - Investigate complex objects by clicking instead of typing in Debug Console
  - Choose important items to move to Watch window

Deviation in Control Flow

No Deadlock (no_deadlock.py)

- **executor.submit(wait_on_future)** in main scope
- **_python_exit** in thread.py
  - Sets global variable **_shutdown** = **True**
  - Note that global is only global to specific module (thread._shutdown)
- **executor.submit(pow, 5, 2)** in wait_on_future
  - Reads _shutdown as True, **raise RuntimeError**

Deadlock (always_deadlock.py)

- **executor.submit(wait_on_future)** in main scope
- **time.sleep** in main scope
  - Global variable **_shutdown** is still **False**
- **executor.submit(pow, 5, 2)** in wait_on_future
  - Successfully submits **self._work_queue.put(w)**

Influencing Script Termination

sometimes_deadlock.py

- Undeterministic behaviour

discourage_deadlock.py

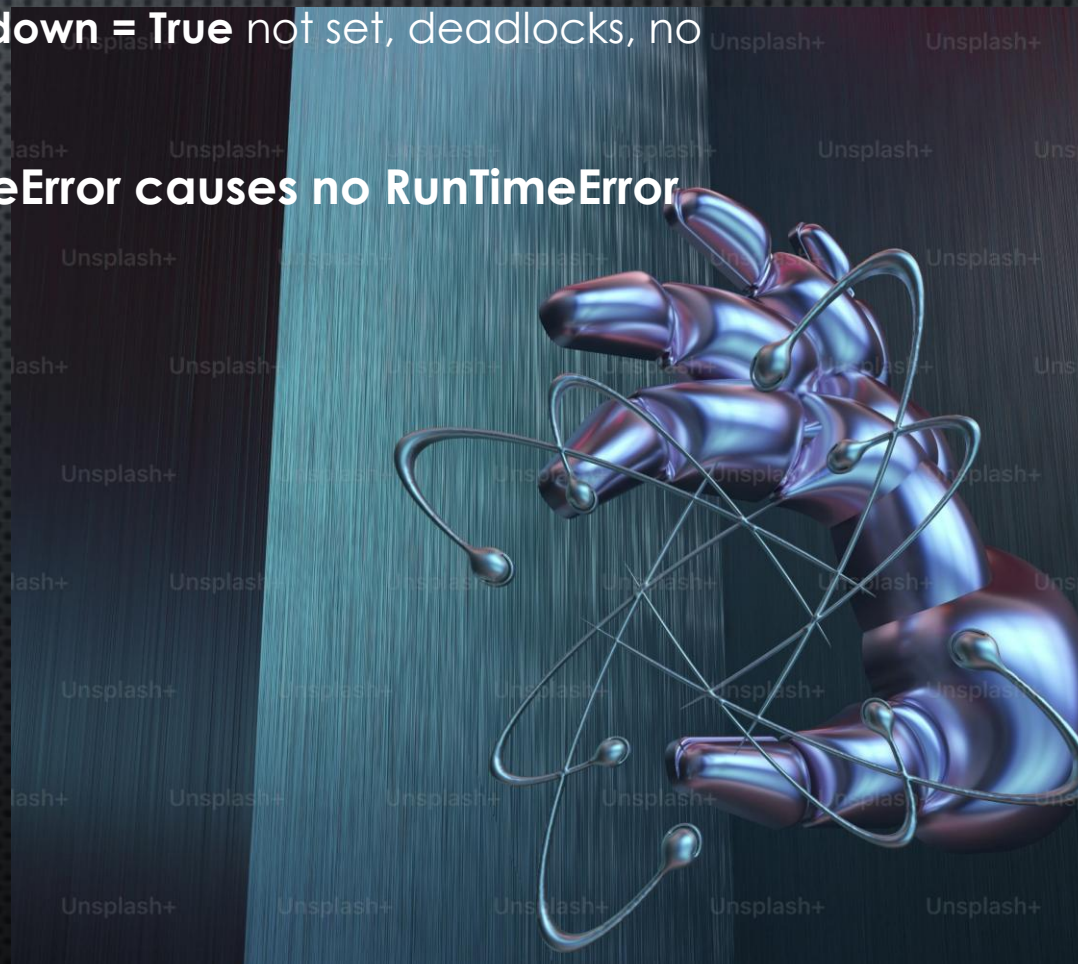
- **executor.shutdown(wait=False)** increases chance of main script exiting first

force_deadlock.py

- **f = executor.submit(wait_on_future)**
- **f.result()**
  - Blocking using condition variables, stronger because no time limit like time.sleep
    - threading.Condition() which use RLock() by default

Unswallow RunTimeError: The HeisenBug

- Need no deadlock scenario to see RunTimeError (_shutdown = True)
- Need **f.result()** or **f.exception** to see exceptions propagated through futures to the submitting scope
- **f.result()** blocks main from exit, **_python_exit** not run, **shutdown = True** not set, deadlocks, no RunTimeError
- **Heisenbug: Adding f.result() to observe the RunTimeError causes no RunTimeError**



Overview of Solutions

solution1_max_workers.py > ...

```python
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures.thread as thread_module
import time


def wait_on_future():
    print("Running wait_on_future")
    f = executor.submit(pow, 5, 2)

    print(f.result())


executor = ThreadPoolExecutor(max_workers=2)  # with 2 workers, this will not deadlock
f = executor.submit(wait_on_future)
f.result()
```

solution2_processpool.py > wait_on_future

```python
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import concurrent.futures.thread as thread_module
import time


def wait_on_future():
    print("Running wait_on_future")
    executor = ProcessPoolExecutor(max_workers=1)
    f = executor.submit(pow, 5, 2)

    print(f.result())


executor = ThreadPoolExecutor(max_workers=1)
f = executor.submit(wait_on_future)
f.result()
```

solution3_externalqueue.py > wait_on_future

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


def wait_on_future(result_queue):
    f = executor.submit(pow, 5, 2)
    result_queue.put(f)  # Put the Future object in the queue


result_queue = queue.Queue()
with ThreadPoolExecutor(max_workers=1) as executor:
    executor.submit(wait_on_future, result_queue)
    # Retrieve result in the main thread
    future = result_queue.get()
    print(future.result())
```

solution4_timeout.py > ...

```python
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures.thread as thread_module
import time


def wait_on_future():
    print("Running wait_on_future")
    f = executor.submit(pow, 5, 2)

    print(f.result())


executor = ThreadPoolExecutor(max_workers=1)
f = executor.submit(wait_on_future)
f.result(timeout=1)
```

Overview of Solutions

Solution Filename	Key idea	Pros	Cons
solution1_max_workers.py	**Increase Worker Count:** Ensure enough threads are available for nested tasks. ThreadPoolExecutor(max_workers=2)	- Simple and effective - Allows concurrent execution of inner and outer tasks within the same pool.	- Increases resource consumption (more threads) - May not solve deadlocks if deeper nesting of tasks exceeds max_workers.
solution2_processpool.py	**Separate Process Pool:** Offload the inner task to a ProcessPoolExecutor to avoid ThreadPoolExecutor deadlock ProcessPoolExecutor(max_workers=1)	- Bypasses ThreadPoolExecutor deadlock by using separate processes - Suitable for CPU-bound tasks (no GIL contention).	- Higher overhead due to process creation - More complex inter-process communication for data - Less suitable for I/O-bound tasks - Not ideal if shared memory access is required.
solution3_externalqueue.py	**External Queue for Futures:** Put the Future object into a queue.Queue and retrieve it from an independent thread (e.g., main thread) for result() result_queue.put(f)	- Breaks the direct blocking dependency on f.result() within the worker thread - Provides more control over result retrieval.	- Adds complexity due to the external queue - Requires explicit management of putting Future objects into and getting them from the queue - The main thread (or another thread) now bears the responsibility of awaiting the result.
solution4_timeout.py	**Timeout:** Prevent indefinite blocking by raising a TimeoutError if the result is not available within a specified time print(f.result(timeout=1))	- Prevents indefinite blocking; raises TimeoutError instead - Useful for diagnosing deadlocks or unresponsive tasks.	- Does not solve the underlying deadlock; merely fails gracefully (or noisily) - The inner task still won't execute - Choosing an appropriate timeout value can be tricky and may lead to false positives or negatives - More of an error-handling mechanism than a true concurrency solution.

Nested Concurrency Patterns

Pattern	Outer Executor	Inner Executor	Use Case	Why Nested?
Web Scraping	ThreadPoolExecutor	ThreadPoolExecutor	Scrape multiple sites, then their subpages	Don't know subpage URLs upfront; need to group results per site
Batch Processing	ThreadPoolExecutor	ProcessPoolExecutor	Fetch data batches, then compute on each batch	Don't know compute jobs until parsing; need aggregation per batch
Compute + Upload	ProcessPoolExecutor	ThreadPoolExecutor	Heavy computation, then upload resulting files	Don't know file count upfront; need grouped upload handling
ML Ensemble	ProcessPoolExecutor	ProcessPoolExecutor	Generate bootstrap samples, then train models	Dynamic sample generation; need model aggregation

Reason for not flattening	Description	Critical Need
Dynamic Task Discovery	Tasks are only known after previous tasks complete	Can't pre-plan all work
Result Grouping	Need to aggregate/reduce results within logical groups	Maintains data relationships
Resource Optimization	Different task types need different executor types	Matches workload to resources
Error Handling	Group-level retry/failure policies	Logical error boundaries

What else can you learn

1. **Signaling — The Sentinel Pattern**
   - The use of None to tell threads to exit is a classic sentinel.
   - Lesson: Simple values can be powerful signals in concurrent systems.
   - Related patterns: poison pill, end-of-stream markers.
2. **Queues — Decoupled, Asynchronous System Design**
   - Workers and producers communicate via a queue.
   - Lesson: Queues enable asynchronous, decoupled, and safe task submission between threads.
   - Foundation of many event-driven and pipeline architectures.
3. **Resource Control — Bounded Concurrency**
   - max_workers limits the number of concurrent threads.
   - Lesson: Concurrency isn't about "as much as possible", but as much as needed.
   - Teaches backpressure and resource safety.
4. **Lifecycle Management — Ownership and Cleanup**
   - Explicit shutdown() vs. implicit _python_exit().
   - Lesson: Resource lifecycles (like threads) require clear ownership and cleanup responsibility.
   - Think: RAII, context managers, try/finally.
5. **Task Abstraction — Futures and the submit() Interface**
   - Tasks submitted return Future objects, hiding execution details.
   - Lesson: Future-based APIs allow for composability, polling, and callback chaining.
   - Common in async frameworks, distributed systems, cloud functions.
6. **Thread Safety via Synchronization — Locks and Atomicity**
   - Internals use threading.Lock, WeakKeyDictionary, etc.
   - Lesson: Even with high-level abstractions, low-level race condition prevention is still required.
   - Reinforces importance of safe data sharing.

What else can you learn

7. **Graceful Degradation — Ignore Failures at Shutdown**
   - _python_exit() doesn't retry, check state, or raise — it assumes shutdown is best-effort.
   - Lesson: Graceful failure and best-effort cleanup are often more practical than perfection, especially at system exit.
8. **Daemon vs. Non-Daemon Threads**
   - By default, threads are non-daemon, so they keep the process alive unless shut down.
   - Lesson: Thread lifecycle directly affects process lifetime — daemon threads vs. non-daemon can make or break shutdown behavior.
9. **Finalizers and atexit Hooks**
   - Uses weakref.finalize and atexit.register to manage global cleanup.
   - Lesson: Be careful with global state and cleanup — it must not assume full interpreter health during teardown.
10. **Blocking vs. Non-blocking APIs**
    - submit() is non-blocking; shutdown(wait=True) is blocking.
    - Lesson: Understand and choose between immediate return and wait-for-completion based on context and backpressure needs.
11. **Thread Leaks and Zombie Threads**
    - Failing to shutdown() or join() threads can lead to leaks or stalled interpreter exit.
    - Lesson: Proper thread lifecycle hygiene is essential for long-running or daemonized applications.

BONUS: 2 types of shutdown

- **_python_exit** setting **_shutdown**
- Heisenbug solution setting **executor.shutdown(wait=False)**
- Both handled by **executor.submit**
- Difference in scope (**python interpreter level** vs **executor level**)

```
118  class ThreadPoolExecutor(_base.Executor):
161      def submit(self, fn, /, *args, **kwargs):
162          with self._shutdown_lock, _global_shutdown_lock:
163              if self._broken:
164                  raise BrokenThreadPool(self._broken)
165
166              if self._shutdown:
167                  raise RuntimeError('cannot schedule new futures after shutdown')
168              if _shutdown:
169                  raise RuntimeError('cannot schedule new futures after '
170                                     'interpreter shutdown')
```

BONUS: Context Manager

- concurrent.futures are usually used with context managers to automatically close the executor
- Similarly to the Heisenbug solution, both shutdown using **executor.shutdown**, albeit with differences in wait = True/False

context_manager.py > ...

```python
from concurrent.futures import ThreadPoolExecutor
import concurrent.futures.thread as thread_module
import time


def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())


with ThreadPoolExecutor(max_workers=1) as executor:
    executor.submit(wait_on_future)
```

usr > lib > python3.10 > concurrent > futures > _base.py > Executor > __exit__

```
571  class Executor(object):
628
629      def shutdown(self, wait=True, *, cancel_futures=False):
630          """Clean-up the resources associated with the Executor.
631
632          It is safe to call this method several times. Otherwise, no other
633          methods can be called after this one.
634
635          Args:
636              wait: If True then shutdown will not return until all running
637                  futures have finished executing and the resources used by the
638                  executor have been reclaimed.
639              cancel_futures: If True then shutdown will cancel all pending
640                  futures. Futures that are completed or running will not be
641                  cancelled.
642          """
643          pass
644
645      def __enter__(self):
646          return self
647
648      def __exit__(self, exc_type, exc_val, exc_tb): self = <concurrent.futures.t
649          self.shutdown(wait=True)
650          return False
```

Frequently Asked Questions

1. **How to know where to put the breakpoints?**
   - Put breakpoint early in execution flow, Step Over (F10) line by line on first few runs until familiar
   - Later put breakpoints at critical sections you want to revisit and use Continue (F5) to make larger jumps

2. **How to know whether there is deadlock?**
   - Simply observing that your script finishes execution doesn't guarantee the absence of deadlocks; you might just be "lucky" during that specific run.
   - Conversely, experiencing a deadlock once doesn't mean it will *always* deadlock.
   - **Halting Problem**: no tool can definitively guarantee the detection of all possible deadlocks in all possible programs *before* they actually occur or get stuck.
   - If a program is still running or already stuck, we can investigate though

Frequently Asked Questions

3. **Are there tools that can help detect deadlocks?**
   - **faulthandler** (Python): This built-in Python module can dump tracebacks of all active threads when a fatal error occurs or after a specified timeout. This can reveal which threads are blocked and what they are waiting on if a deadlock causes the program to hang.
   - OS-level tools (e.g., **strace** on Linux, **BPF Compiler Collection** tools): These system-level utilities can trace system calls, including lock acquisitions and releases. While powerful, interpreting their output for deadlock analysis can require a deeper understanding of operating system internals. They show you the resources threads are contending for at a given moment.
   - **pydld** (Python Deadlock Detector): This library actively monitors lock operations in a running Python program. It builds a real-time dependency graph of lock acquisitions and can detect cycles in this graph, indicating an actual or imminent deadlock. It effectively tells you "a deadlock has occurred" or "threads are currently stuck."

Resources

1. PEP3148: https://peps.python.org/pep-3148/#threadpoolexecutor
2. Introduction to PEP: https://peps.python.org/pep-0001/
3. How to use debugger: https://www.youtube.com/watch?v=7qZBwhSIfOo
4. VSCode python debugging: https://code.visualstudio.com/docs/python/debugging
5. Deterministic Logic vs Undeterministic Science: https://www.freecodecamp.org/news/the-logic-philosophy-and-science-of-software-testing-handbook-for-developers/
6. Synchronization with threading.Barrier: https://www.linkedin.com/feed/update/urn:li:activity:7332378628250288128
7. Avoiding multiprocessing race conditions: https://www.linkedin.com/feed/update/urn:li:activity:7324384085206863872
8. Multiprocessing and Pipes: https://www.linkedin.com/feed/update/urn:li:activity:7323283169821061120
9. LLMs to visualize relationship between synchronization primitives (Event, Condition, Lock, Semaphore, Barrier, Rlock, Timer): https://www.linkedin.com/posts/hanqi91_is-claude-ready-for-education-yes-i-saw-activity-7313614310440587267--J33
10. Problems in Concurrent algorithm design: https://en.wikipedia.org/wiki/Dining_philosophers_problem
11. Unintuitive blocking behaviour in nested context managers: https://stackoverflow.com/questions/79538469/is-with-concurrent-futures-executor-blocking/79538630
12. State of No GIL python: https://pyfound.blogspot.com/2025/06/python-language-summit-2025-state-of-free-threaded-python.html
13. BPF Compiler Collection (BCC): https://github.com/iovisor/bcc

About Me

- Principal Data Scientist @ Monarch Tractor Asia Pacific Pte Ltd
- Loves to learn through Communities (Slack, Discord)
- Teaches Bootcamps (Le Wagon)
- Writes
  - Medium: https://hanqi01.medium.com/
  - freeCodeCamp: https://www.freecodecamp.org/news/the-logic-philosophy-and-science-of-software-testing-handbook-for-developers/
- Plays Badminton, Piano, Drums, Chess, Sings
- Linkedin: https://www.linkedin.com/in/hanqi91 (QR code)
  - Please give your 1st time speaker some feedback!
- Code: https://github.com/gitgithan/pyconsg2025_multithreading
- Slides: Organizer will collect and share (also on github later)