asyncio: We Did It Wrong

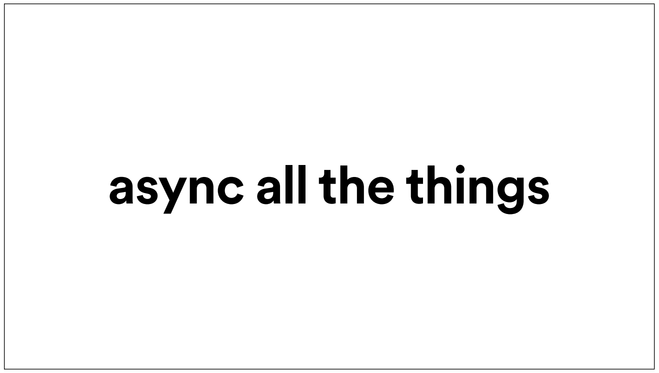
Lynn Root | SRE | @roguelynn

$ whoami_

In actuality, what an SRE does at spotify - as it varies widely at different companies - is a combination of backend development where my team and I run a few services that other engineers use daily, plus a little devops and system administration.

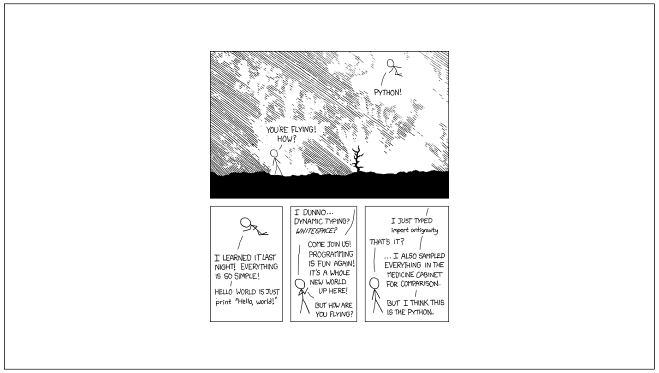
I'm also our FOSS evangelist: I help a lot of teams release their projects and tools under the spotify GitHub organization.

Lastly, I help lead PyLadies - a global mentorship group for women, and friends, to help increase diversity in the Python community.

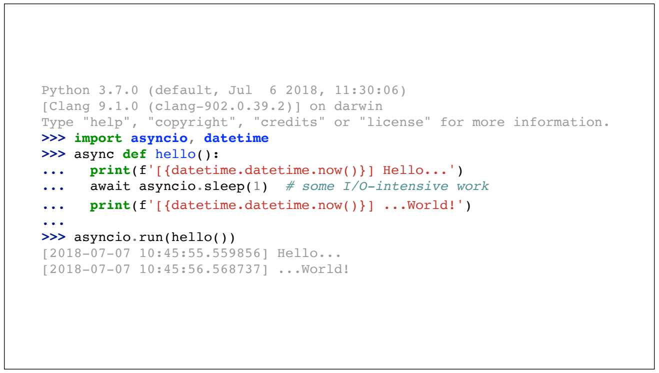


async all the things

asyncio - The concurrent Python programmer’s dream, the answer to everyone's asynchronous prayers.


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

The `asyncio` module has various layers of abstraction allowing developers as much control as they need and are comfortable with.



```
Python 3.7.0 (default, Jul  6 2018, 11:30:06)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio, datetime
>>> async def hello():
...   print(f'[{datetime.datetime.now()}] Hello...')
...   await asyncio.sleep(1)  # some I/O-intensive work
...   print(f'[{datetime.datetime.now()}] ...World!')
...
>>> asyncio.run(hello())
[2018-07-07 10:45:55.559856] Hello...
[2018-07-07 10:45:56.568737] ...World!
```

Simple "Hello, World"-like examples show how it can be so simple, look at that!

But it's easy to get lulled into a false sense of security. This ain't helpful.

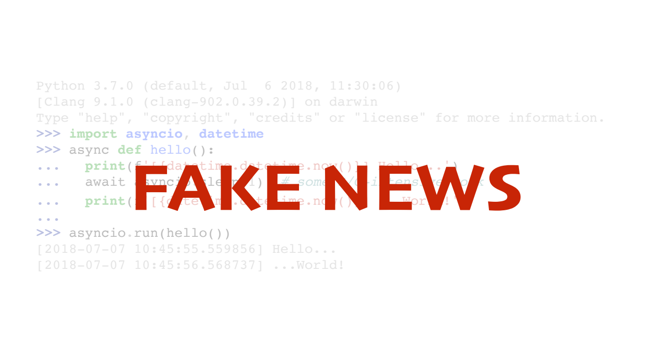
```
Python 3.7.0 (default, Jul  6 2018, 11:30:06)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio, datetime
>>> async def hello():
...   print(f'[{datetime.datetime.now()}] Hello...')
...   await asyncio.sleep(1)  # some I/O-intensive work
...   print(f'[{datetime.datetime.now()}] ...World!')
...
>>> asyncio.run(hello())
[2018-07-07 10:45:55.559856] Hello...
[2018-07-07 10:45:56.568737] ...World!
```

FAKE NEWS

We're led to believe that we're able to do a lot with the structured `async`/`await` API layer. Some tutorials, while great for the developer getting their toes wet, try to illustrate real world examples, but are actually just beefed-up "hello, world"s.

Some even misuse parts of `asyncio`'s interface, allowing one to easily fall into the depths of callback hell.

Some get you easily up and running with `asyncio`, but then you may not realize it's not correct or exactly what you want, or only gets you part of the way there. While some tutorials and walk throughs do a lot to improve upon the basic "hello, world" use case, it is still just a web crawler. I'm not sure about others, but I'm not building web crawlers at Spotify.



```
Python 3.7.0 (default, Jul  6 2018, 11:30:06)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio, datetime
>>> async def hello():
...     print(f'[{datetime.datetime.now()}] Hello...')
...     await asyncio.sleep(1)  # some I/O-intensive work
...     print(f'[{datetime.datetime.now()}] ...World!')
...
>>> asyncio.run(hello())
[2018-07-07 10:45:55.559856] Hello...
[2018-07-07 10:45:56.568737] ...World!
```

FAKE NEWS

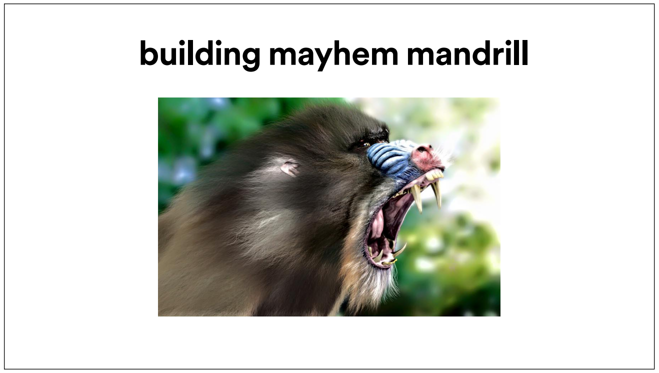
Sure, we needed to make a lot of HTTP requests that should be non-blocking. But these services also had to react to events from a pubsub, measure the progress of actions initiated from those events, handle any incomplete actions or other external errors, deal with pubsub message lease management, measure service level indicators, and send metrics. And needed to use non-`asyncio`-friendly dependencies. This quickly got difficult.

```
Python 3.7.0 (default, Jul  6 2018, 11:30:06)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio, datetime
>>> async def hello():
...     print(f'[{datetime.datetime.now()}] Hello...')
...     await asyncio.sleep(1)  # some I/O-intensive work
...     print(f'[{datetime.datetime.now()}] ...World!')
...
>>> asyncio.run(hello())
[2018-07-07 10:45:55.559856] Hello...
[2018-07-07 10:45:56.568737] ...World!
```

FAKE NEWS

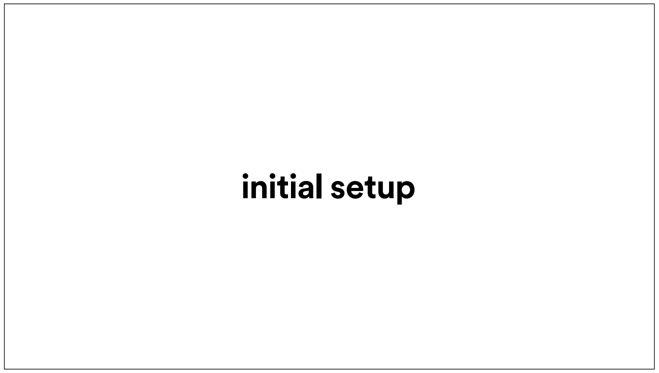
Sure, we needed to make a lot of HTTP requests that should be non-blocking. But these services also had to react to events from a pubsub, measure the progress of actions initiated from those events, handle any incomplete actions or other external errors, deal with pubsub message lease management, measure service level indicators, and send metrics. And needed to use non-`asyncio`-friendly dependencies. This quickly got difficult.

Allow me to provide you a real-world example that actually comes from the real world. Recently at Spotify, we built a service that does periodic hard restarts our entire fleet of instances.



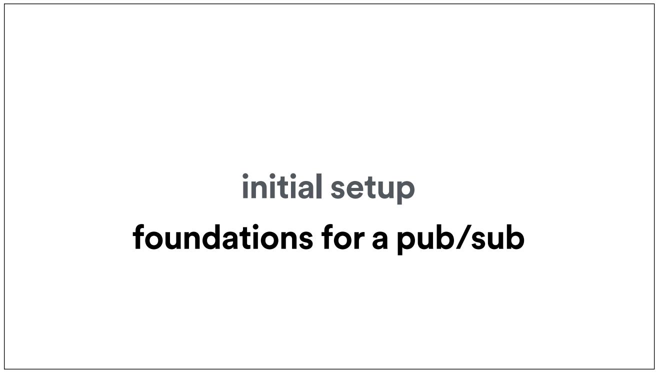
building mayhem mandrill

And we’re going to do that here. Let’s build a service called Mayhem Mandrill which will listen for a pub/sub message and restart a host based off of that message. As we build this service, I’ll point out the traps that I may or may not have fallen into. This will essentially become the type of resource that past Lynn would have wanted a year or two ago.



initial setup

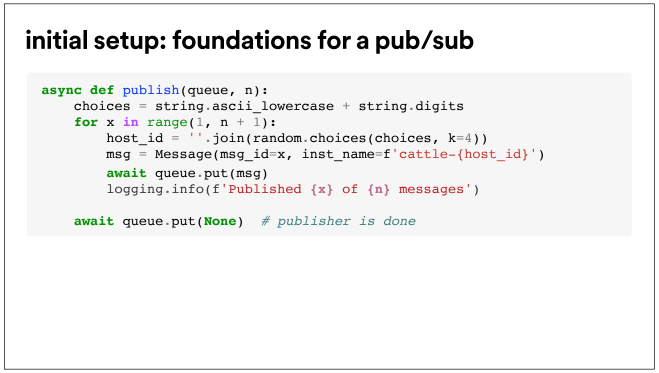
We’ll start with some foundational code…



initial setup

foundations for a pub/sub

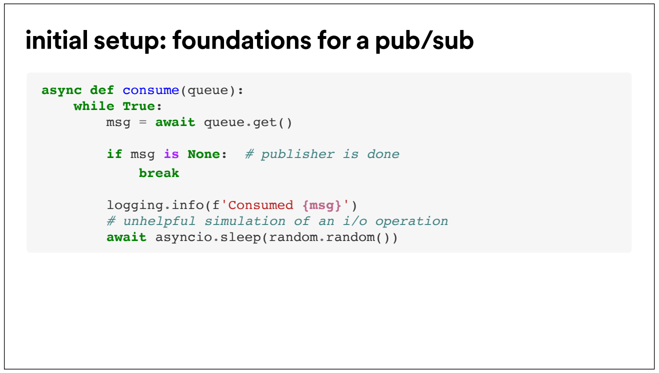
This is pretty much inspired by a tutorial from asyncio’s documentation.



initial setup: foundations for a pub/sub

```
async def publish(queue, n):
    choices = string.ascii_lowercase + string.digits
    for x in range(1, n + 1):
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=x, inst_name=f'cattle-{host_id}')
        await queue.put(msg)
        logging.info(f'Published {x} of {n} messages')

    await queue.put(None)  # publisher is done
```

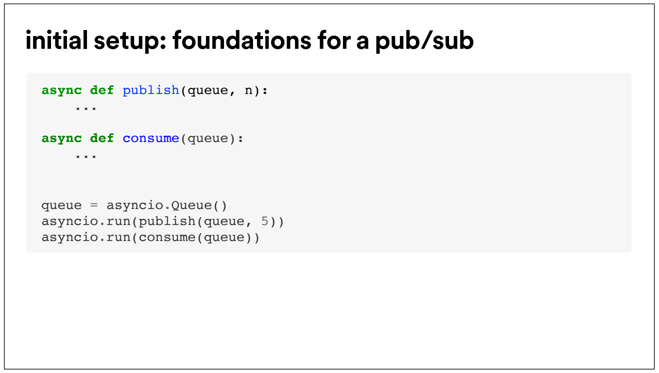
<describe code>



initial setup: foundations for a pub/sub

```
async def consume(queue):
    while True:
        msg = await queue.get()

        if msg is None:  # publisher is done
            break

        logging.info(f'Consumed {msg}')
        # unhelpful simulation of an i/o operation
        await asyncio.sleep(random.random())
```

<describe code>

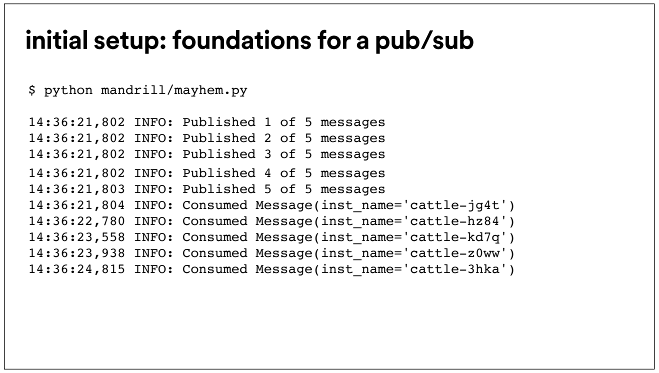


initial setup: foundations for a pub/sub

```
async def publish(queue, n):
    ...

async def consume(queue):
    ...


queue = asyncio.Queue()
asyncio.run(publish(queue, 5))
asyncio.run(consume(queue))
```

Using Python 3.7's latest syntactic sugar! <TODO:expand>

When we run this, we see the following:



initial setup: foundations for a pub/sub

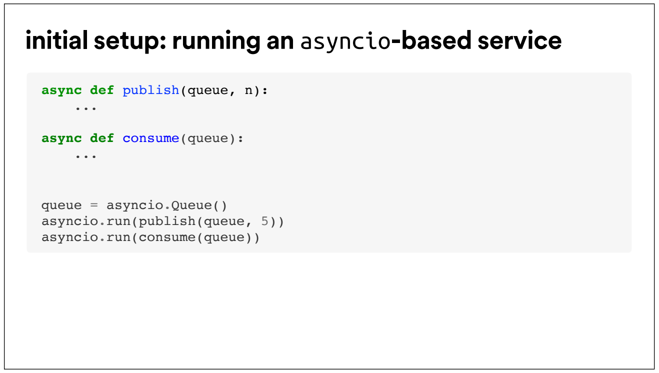
```
$ python mandrill/mayhem.py

14:36:21,802 INFO: Published 1 of 5 messages
14:36:21,802 INFO: Published 2 of 5 messages
14:36:21,802 INFO: Published 3 of 5 messages
14:36:21,802 INFO: Published 4 of 5 messages
14:36:21,803 INFO: Published 5 of 5 messages
14:36:21,804 INFO: Consumed Message(inst_name='cattle-jg4t')
14:36:22,780 INFO: Consumed Message(inst_name='cattle-hz84')
14:36:23,558 INFO: Consumed Message(inst_name='cattle-kd7q')
14:36:23,938 INFO: Consumed Message(inst_name='cattle-z0ww')
14:36:24,815 INFO: Consumed Message(inst_name='cattle-3hka')
```

<describe>
So let's work off of this. We'll first start with boilerplate-like code to start and stop this pub/sub simulator.

initial setup

running an `asyncio`-based service

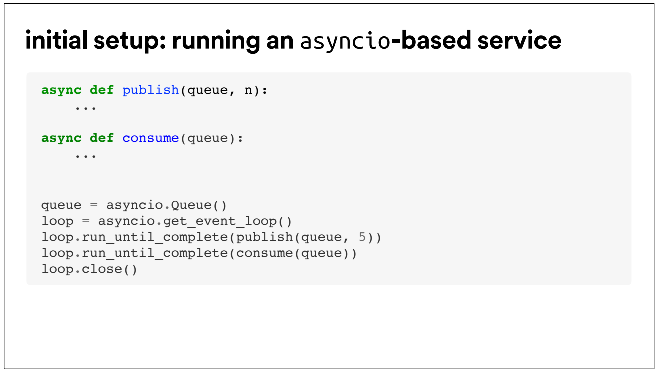
<NEW>So far, we don't have a running service…



initial setup: running an `asyncio`-based service

```
async def publish(queue, n):
    ...

async def consume(queue):
    ...


queue = asyncio.Queue()
asyncio.run(publish(queue, 5))
asyncio.run(consume(queue))
```

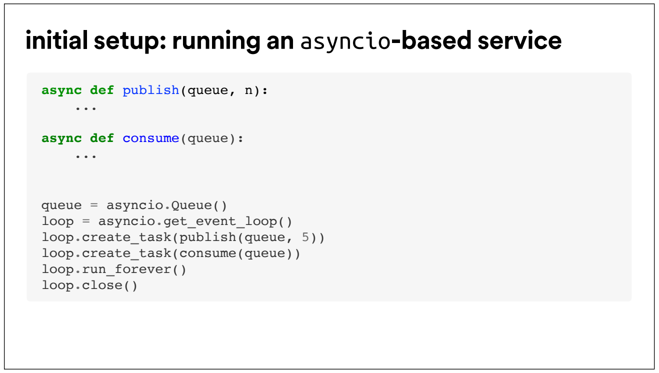
<NEW>…it’s merely just a pipeline or a batch job right now. Note that `asyncio.run` is new as of 3.7,



initial setup: running an `asyncio`-based service

```
async def publish(queue, n):
    ...

async def consume(queue):
    ...


queue = asyncio.Queue()
loop = asyncio.get_event_loop()
loop.run_until_complete(publish(queue, 5))
loop.run_until_complete(consume(queue))
loop.close()
```

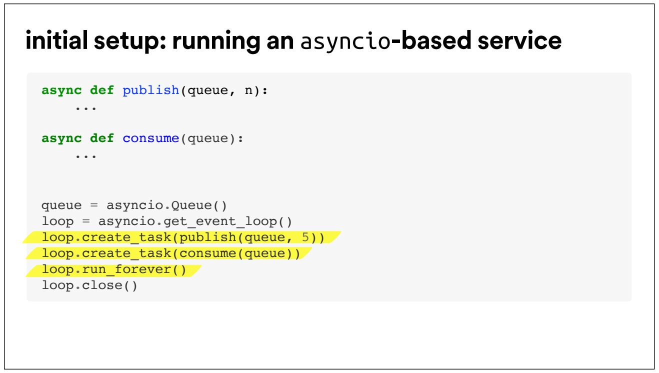
<NEW>before 3.7, we had to setup and teardown the loop ourselves like this. Note that it is a good habit to clean up and close the event loop since we created it.



initial setup: running an `asyncio`-based service

```
async def publish(queue, n):
    ...

async def consume(queue):
    ...


queue = asyncio.Queue()
loop = asyncio.get_event_loop()
loop.create_task(publish(queue, 5))
loop.create_task(consume(queue))
loop.run_forever()
loop.close()
```

<NEW>but this is a service, so we don’t want it to just run once, but continually consume from a publisher. And unfortunately, there isn’t a decent way to start a long-running service that is not an HTTP server in python 3.7. So we’ll stick with 3.6 approach,

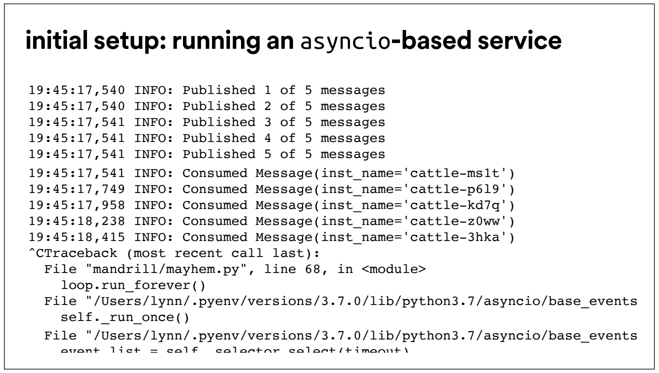


initial setup: running an `asyncio`-based service

```
async def publish(queue, n):
    ...

async def consume(queue):
    ...


queue = asyncio.Queue()
loop = asyncio.get_event_loop()
loop.create_task(publish(queue, 5))
loop.create_task(consume(queue))
loop.run_forever()
loop.close()
```

<NEW>With that, we’ll create tasks out of the two coroutines, which will schedule them on the loop. And then start the loop, telling it to run forever.

So then running with this updated code…

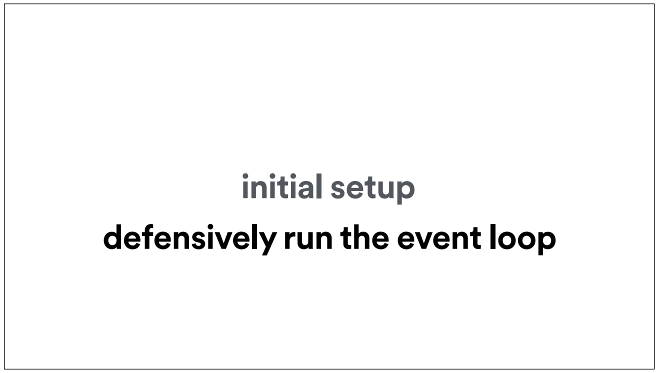


initial setup: running an `asyncio`-based service

```
19:45:17,540 INFO: Published 1 of 5 messages
19:45:17,540 INFO: Published 2 of 5 messages
19:45:17,541 INFO: Published 3 of 5 messages
19:45:17,541 INFO: Published 4 of 5 messages
19:45:17,541 INFO: Published 5 of 5 messages
19:45:17,541 INFO: Consumed Message(inst_name='cattle-ms1t')
19:45:17,749 INFO: Consumed Message(inst_name='cattle-p6l9')
19:45:17,958 INFO: Consumed Message(inst_name='cattle-kd7q')
19:45:18,238 INFO: Consumed Message(inst_name='cattle-z0ww')
19:45:18,415 INFO: Consumed Message(inst_name='cattle-3hka')
^CTraceback (most recent call last):
  File "mandrill/mayhem.py", line 68, in <module>
    loop.run_forever()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    self._run_once()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    event_list = self._selector.select(timeout)
```

...we see that all messages are published and then consumed, and then we hang, because there is no more work to be done (we only published 5 messages, after all). To stop the process, we have to interrupt it (via ^C or sending a signal like kill -9 <pid>).

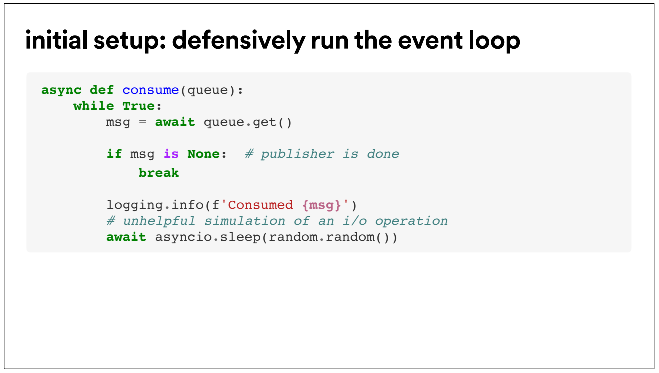
So yeah, That's nice and ugly... You may notice that we'll also never get to the loop.close() line either. Nor are we handling any exceptions that may raise from awaiting the publish and consume coroutines.



initial setup

defensively run the event loop

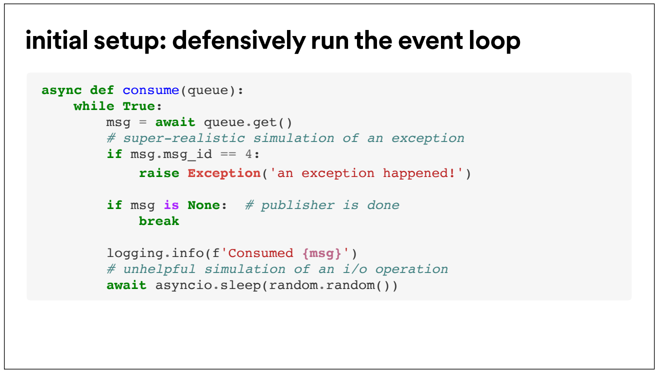
We'll first address the catching of exceptions that arise from coroutines. Let's fake an error in the consume coroutine:



initial setup: defensively run the event loop

```python
async def consume(queue):
    while True:
        msg = await queue.get()

        if msg is None:  # publisher is done
            break

        logging.info(f'Consumed {msg}')
        # unhelpful simulation of an i/o operation
        await asyncio.sleep(random.random())
```

So this is where we started

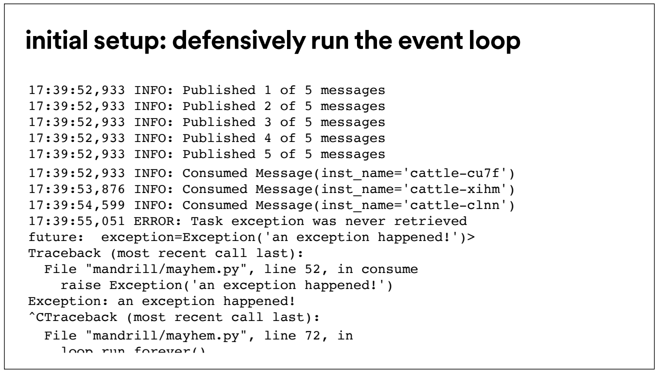


initial setup: defensively run the event loop

```python
async def consume(queue):
    while True:
        msg = await queue.get()
        # super-realistic simulation of an exception
        if msg.msg_id == 4:
            raise Exception('an exception happened!')

        if msg is None:  # publisher is done
            break

        logging.info(f'Consumed {msg}')
        # unhelpful simulation of an i/o operation
        await asyncio.sleep(random.random())
```

Now I’m just going to add a fake error, so if the message ID is equal to 4, we get some silly generic exception.

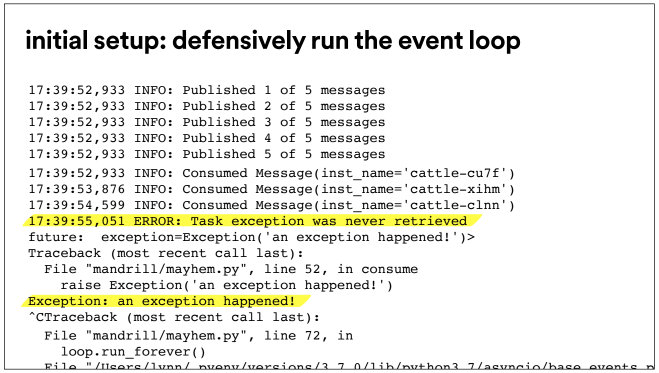
Now if we run it as is



initial setup: defensively run the event loop

```
17:39:52,933 INFO: Published 1 of 5 messages
17:39:52,933 INFO: Published 2 of 5 messages
17:39:52,933 INFO: Published 3 of 5 messages
17:39:52,933 INFO: Published 4 of 5 messages
17:39:52,933 INFO: Published 5 of 5 messages
17:39:52,933 INFO: Consumed Message(inst_name='cattle-cu7f')
17:39:53,876 INFO: Consumed Message(inst_name='cattle-xihm')
17:39:54,599 INFO: Consumed Message(inst_name='cattle-clnn')
17:39:55,051 ERROR: Task exception was never retrieved
future:  exception=Exception('an exception happened!')>
Traceback (most recent call last):
  File "mandrill/mayhem.py", line 52, in consume
    raise Exception('an exception happened!')
Exception: an exception happened!
^CTraceback (most recent call last):
  File "mandrill/mayhem.py", line 72, in
    loop.run_forever()
```

We get an error saying "exception was never retrieved."…



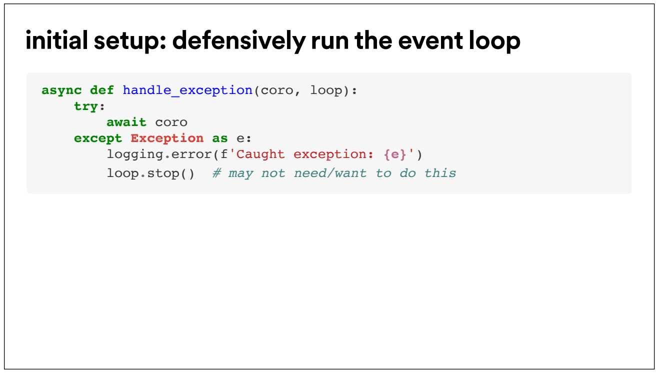
initial setup: defensively run the event loop

```
17:39:52,933 INFO: Published 1 of 5 messages
17:39:52,933 INFO: Published 2 of 5 messages
17:39:52,933 INFO: Published 3 of 5 messages
17:39:52,933 INFO: Published 4 of 5 messages
17:39:52,933 INFO: Published 5 of 5 messages
17:39:52,933 INFO: Consumed Message(inst_name='cattle-cu7f')
17:39:53,876 INFO: Consumed Message(inst_name='cattle-xihm')
17:39:54,599 INFO: Consumed Message(inst_name='cattle-clnn')
17:39:55,051 ERROR: Task exception was never retrieved
future:  exception=Exception('an exception happened!')>
Traceback (most recent call last):
  File "mandrill/mayhem.py", line 52, in consume
    raise Exception('an exception happened!')
Exception: an exception happened!
^CTraceback (most recent call last):
  File "mandrill/mayhem.py", line 72, in
    loop.run_forever()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events.p
```

…from our faked exception. This is admittedly a part of the asyncio API that's not that friendly. If this was synchronous code, we'd simply see the error that we raised, and the script itself would fail. But it get's swallowed up into this unretrieved task.

So to deal with this, as advised in the asyncio documentation [1] , we'll need to have a wrapper coroutine to consume the exception and stop the loop.

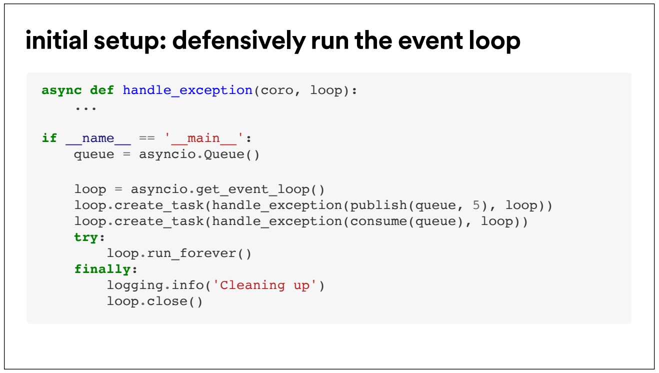
[1] https://docs.python.org/3/library/asyncio-dev.html#detect-exceptions-never-consumed



initial setup: defensively run the event loop

```
async def handle_exception(coro, loop):
    try:
        await coro
    except Exception as e:
        logging.error(f'Caught exception: {e}')
        loop.stop()  # may not need/want to do this
```

So we make a little top-level wrapper to run and handle exceptions from coroutines. And we're just deciding to force our service to crash hard by stopping the event loop if we do happen upon an exception.

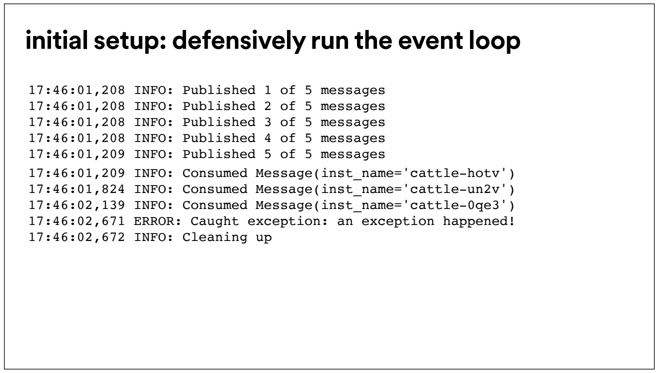


initial setup: defensively run the event loop

```
async def handle_exception(coro, loop):
    ...

if __name__ == '__main__':
    queue = asyncio.Queue()

    loop = asyncio.get_event_loop()
    loop.create_task(handle_exception(publish(queue, 5), loop))
    loop.create_task(handle_exception(consume(queue), loop))
    try:
        loop.run_forever()
    finally:
        logging.info('Cleaning up')
        loop.close()
```

Updating our main section, we wrap our publish and consume coroutine functions with the handle_exception.

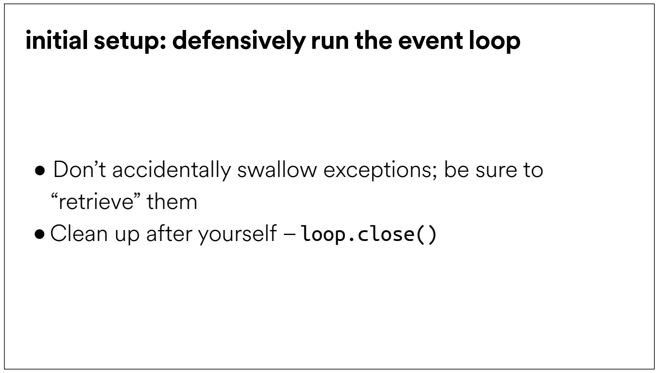
So now when we run our script, we get something a little cleaner:



initial setup: defensively run the event loop

```
17:46:01,208 INFO: Published 1 of 5 messages
17:46:01,208 INFO: Published 2 of 5 messages
17:46:01,208 INFO: Published 3 of 5 messages
17:46:01,208 INFO: Published 4 of 5 messages
17:46:01,209 INFO: Published 5 of 5 messages
17:46:01,209 INFO: Consumed Message(inst_name='cattle-hotv')
17:46:01,824 INFO: Consumed Message(inst_name='cattle-un2v')
17:46:02,139 INFO: Consumed Message(inst_name='cattle-0qe3')
17:46:02,671 ERROR: Caught exception: an exception happened!
17:46:02,672 INFO: Cleaning up
```

Ah that's a bit more clear.



initial setup: defensively run the event loop

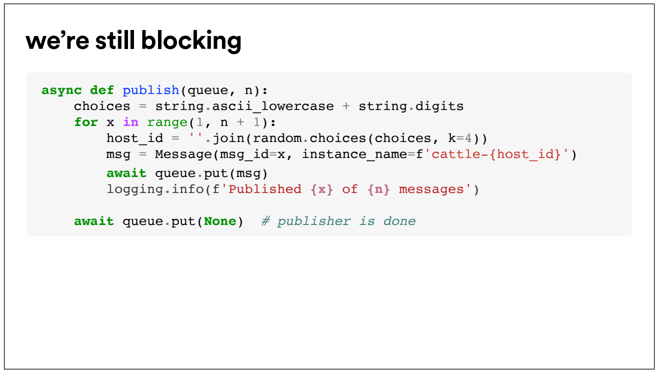
- Don’t accidentally swallow exceptions; be sure to “retrieve” them
- Clean up after yourself – `loop.close()`

So far, for setting up an asyncio service, you want to be sure you surface the exceptions from your coroutines, and to clean up what you've created. We'll expand on that clean up bit later on. This is clean enough for now.

we’re still blocking

I've seen quite a tutorials that make use of async and await in a way that, while does not block the event loop, is still iterating through tasks serially, effectively not actually adding any concurrency.

Taking a look at where our script is now:



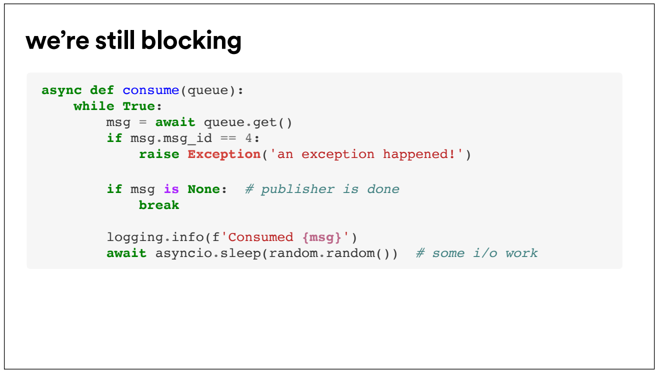
we’re still blocking

```python
async def publish(queue, n):
    choices = string.ascii_lowercase + string.digits
    for x in range(1, n + 1):
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=x, instance_name=f'cattle-{host_id}')
        await queue.put(msg)
        logging.info(f'Published {x} of {n} messages')

    await queue.put(None)  # publisher is done
```

As this was adapted from the asyncio tutorial [1], we are still serially processing each item we produce, and then consume. The event loop itself isn't blocked; if we had other tasks/coroutines going on, they of course wouldn't be blocked.

This might seem obvious to some, but it definitely isn't to all. We are blocking ourselves; first we produce all the messages, one by one.

[1] http://asyncio.readthedocs.io/en/latest/producer_consumer.html

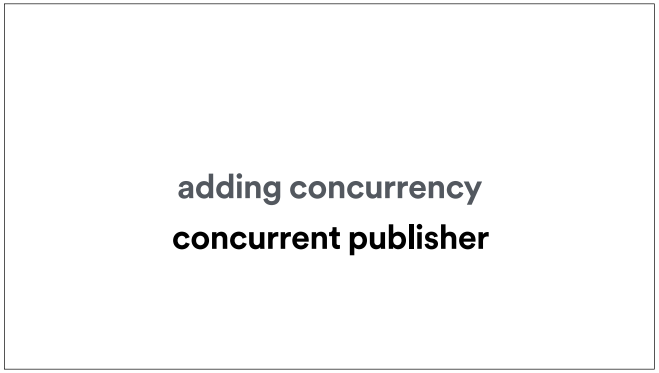


we're still blocking

```
async def consume(queue):
    while True:
        msg = await queue.get()
        if msg.msg_id == 4:
            raise Exception('an exception happened!')

        if msg is None:  # publisher is done
            break

        logging.info(f'Consumed {msg}')
        await asyncio.sleep(random.random())  # some i/o work
```

Then we consume them, one by one. The loops we have (for x in range(1, n+1) in publish(), and while True in consume()) block ourselves from moving onto the next message while we await to do something.

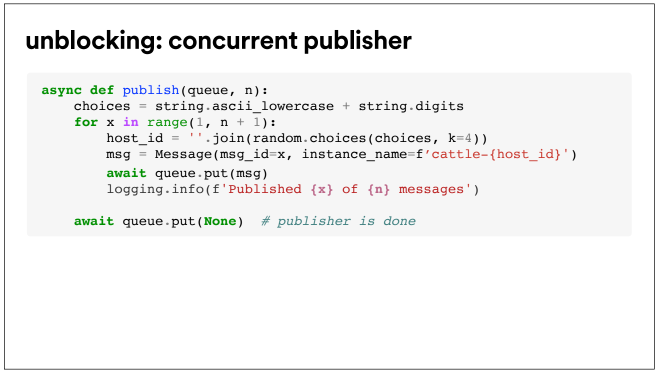
While this is technically a working example of a pub/sub-like queue with asyncio, it's not what we want. It's no different than synchronous code. Whether we are building an event-driven service (like this walk through), or a pipeline/batch job, we're not taking advantage of the concurrency that asyncio can provide.



adding concurrency

concurrent publisher

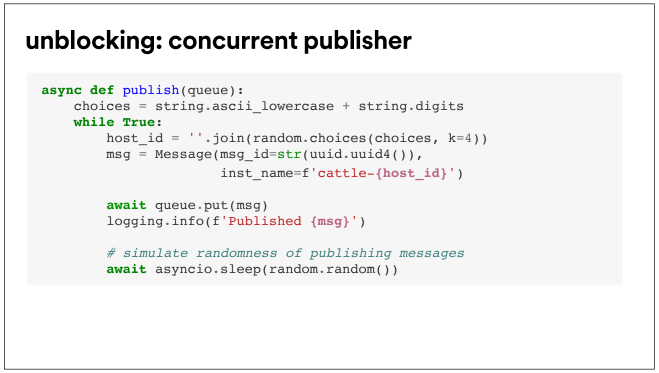
Let's start with the publisher part



unblocking: concurrent publisher

```python
async def publish(queue, n):
    choices = string.ascii_lowercase + string.digits
    for x in range(1, n + 1):
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=x, instance_name=f'cattle-{host_id}')
        await queue.put(msg)
        logging.info(f'Published {x} of {n} messages')

    await queue.put(None)  # publisher is done
```

Here is where we’re starting.

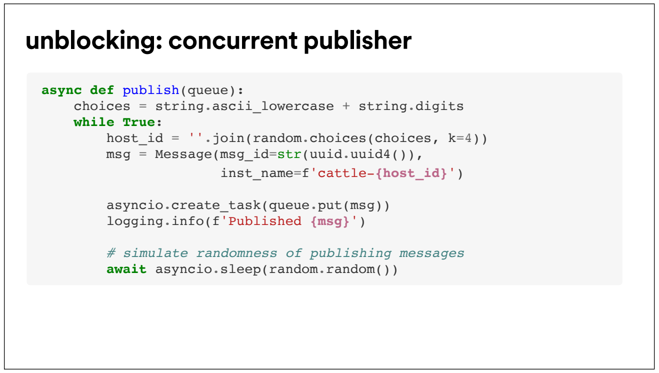


unblocking: concurrent publisher

```python
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        await queue.put(msg)
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

And we essentially throw in a `while True` loop, removing the need to provide a set number of messages to publish. I’ve also just added the creation of a unique ID for each message produced, since it’s no longer as simple as 1 through 5.

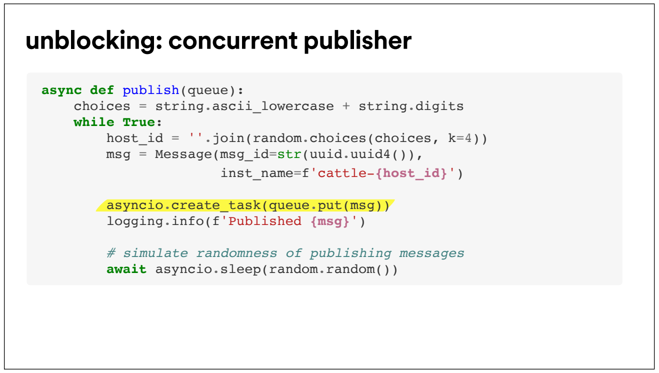
One other thing…



unblocking: concurrent publisher

```
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

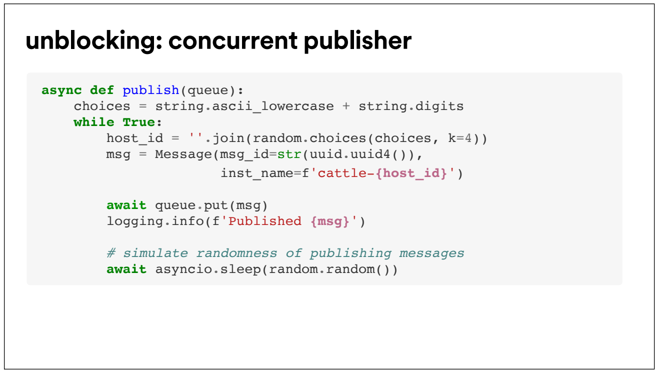
I'm also going to remove the `await` for the `queue.put` and…



unblocking: concurrent publisher

```
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

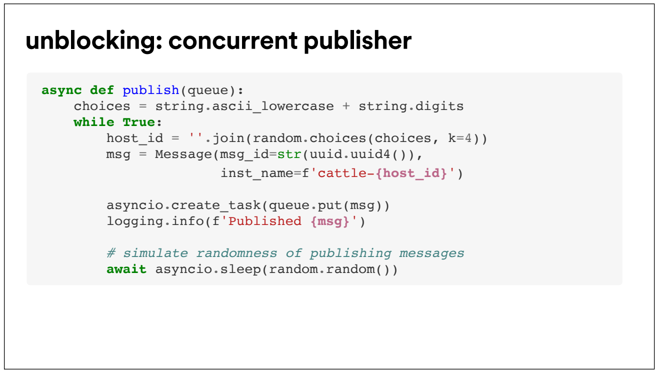
… replace it with create_task. The asyncio.create_task will actually schedule the coroutine on the loop without blocking the rest of the for-loop. The create_task method does return a task, but we can also use it as a "fire and forget" mechanism.



unblocking: concurrent publisher

```python
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        await queue.put(msg)
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

If we left the `await` in here, everything after it will be blocked. This isn’t an issue in our current setup; it could be if we limit the size of the queue, then that await would be waiting on space to free up in the queue.

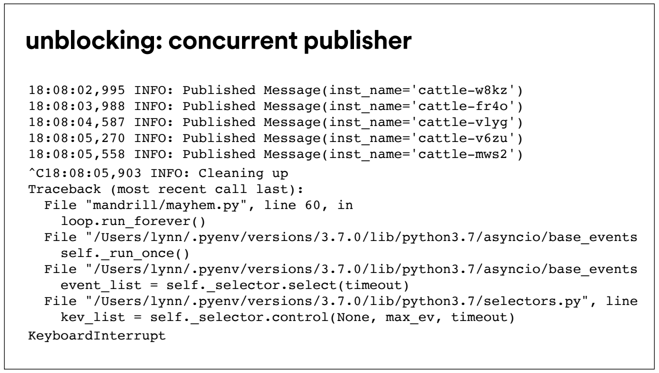


unblocking: concurrent publisher

```python
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

So using `create_task` tells the loop to put the message on the queue as soon as it gets a chance, and allows us to continue on publishing messages.

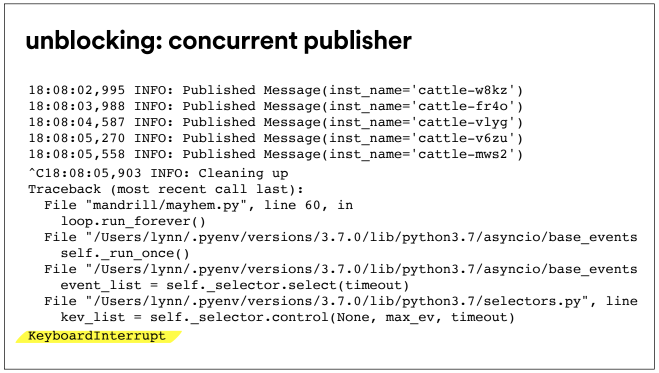
Now in running this:



unblocking: concurrent publisher

```
18:08:02,995 INFO: Published Message(inst_name='cattle-w8kz')
18:08:03,988 INFO: Published Message(inst_name='cattle-fr4o')
18:08:04,587 INFO: Published Message(inst_name='cattle-vlyg')
18:08:05,270 INFO: Published Message(inst_name='cattle-v6zu')
18:08:05,558 INFO: Published Message(inst_name='cattle-mws2')
^C18:08:05,903 INFO: Cleaning up
Traceback (most recent call last):
  File "mandrill/mayhem.py", line 60, in
    loop.run_forever()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    self._run_once()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    event_list = self._selector.select(timeout)
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/selectors.py", line
    kev_list = self._selector.control(None, max_ev, timeout)
KeyboardInterrupt
```

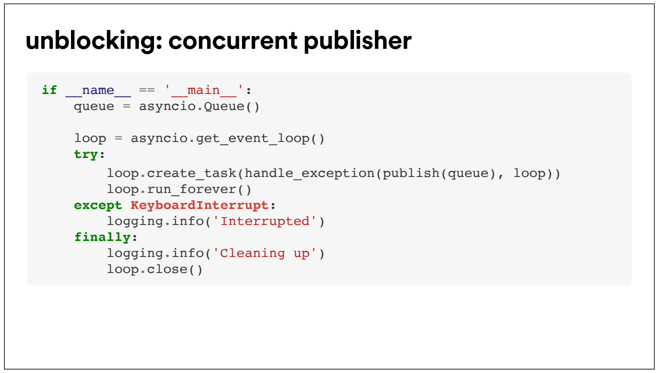
We're happily creating and publishing messages,…



unblocking: concurrent publisher

```
18:08:02,995 INFO: Published Message(inst_name='cattle-w8kz')
18:08:03,988 INFO: Published Message(inst_name='cattle-fr4o')
18:08:04,587 INFO: Published Message(inst_name='cattle-vlyg')
18:08:05,270 INFO: Published Message(inst_name='cattle-v6zu')
18:08:05,558 INFO: Published Message(inst_name='cattle-mws2')
^C18:08:05,903 INFO: Cleaning up
Traceback (most recent call last):
  File "mandrill/mayhem.py", line 60, in
    loop.run_forever()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    self._run_once()
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/base_events
    event_list = self._selector.select(timeout)
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/selectors.py", line
    kev_list = self._selector.control(None, max_ev, timeout)
KeyboardInterrupt
```

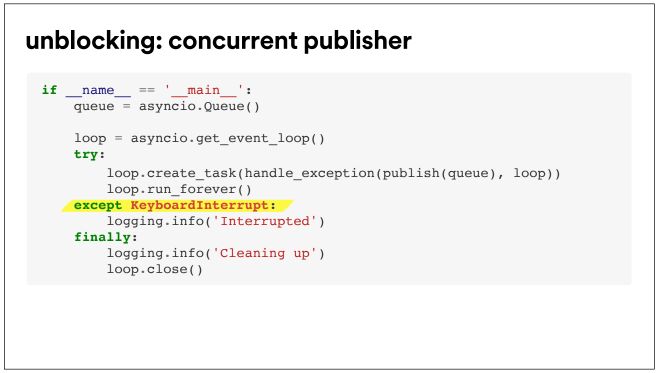
but you'll notice that KeyboardInterrupt – trigged by the ^C – is not actually caught.



unblocking: concurrent publisher

```
if __name__ == '__main__':
    queue = asyncio.Queue()

    loop = asyncio.get_event_loop()
    try:
        loop.create_task(handle_exception(publish(queue), loop))
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

Let's quickly clean up that traceback…

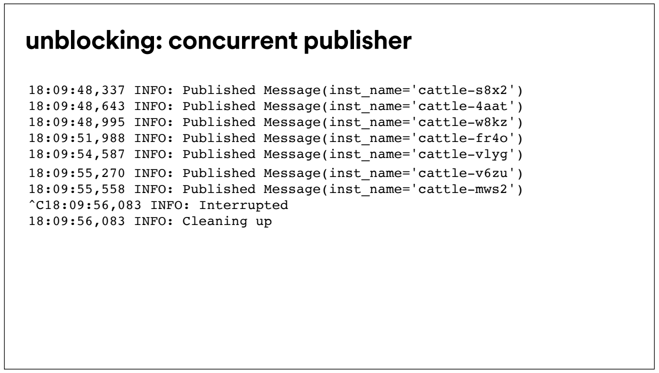


unblocking: concurrent publisher

```
if __name__ == '__main__':
    queue = asyncio.Queue()

    loop = asyncio.get_event_loop()
    try:
        loop.create_task(handle_exception(publish(queue), loop))
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

…from the KeyboardInterrupt; it's a quick bandaid, as I'll explain further on.

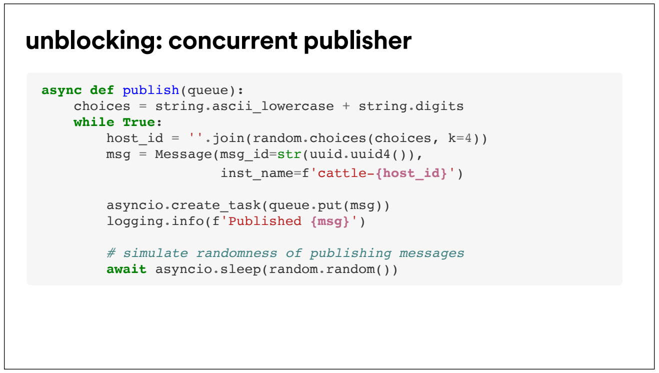
So now we see:



unblocking: concurrent publisher

```
18:09:48,337 INFO: Published Message(inst_name='cattle-s8x2')
18:09:48,643 INFO: Published Message(inst_name='cattle-4aat')
18:09:48,995 INFO: Published Message(inst_name='cattle-w8kz')
18:09:51,988 INFO: Published Message(inst_name='cattle-fr4o')
18:09:54,587 INFO: Published Message(inst_name='cattle-vlyg')
18:09:55,270 INFO: Published Message(inst_name='cattle-v6zu')
18:09:55,558 INFO: Published Message(inst_name='cattle-mws2')
^C18:09:56,083 INFO: Interrupted
18:09:56,083 INFO: Cleaning up
```

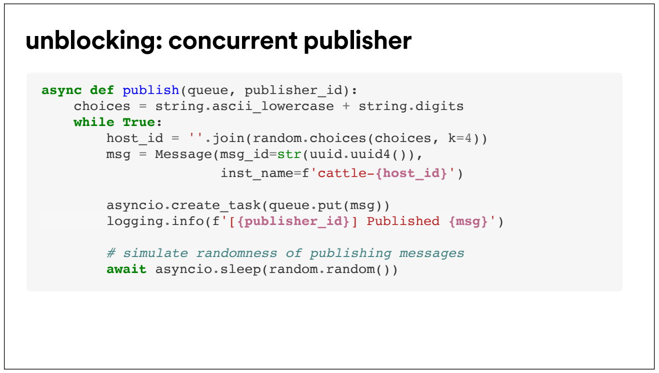
Fantastic! Much cleaner.



unblocking: concurrent publisher

```python
async def publish(queue):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

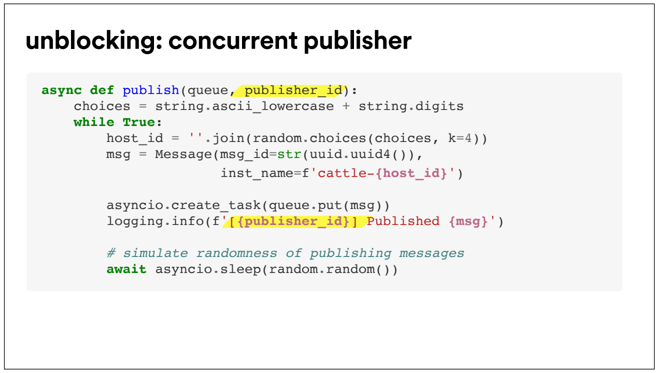
So, it's probably hard to see how this is concurrent right now.



unblocking: concurrent publisher

```python
async def publish(queue, publisher_id):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'[{publisher_id}] Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

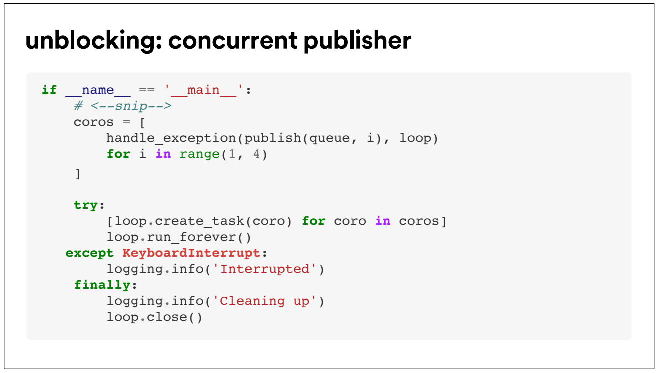
Let's have multiple publishers.



unblocking: concurrent publisher

```python
async def publish(queue, publisher_id):
    choices = string.ascii_lowercase + string.digits
    while True:
        host_id = ''.join(random.choices(choices, k=4))
        msg = Message(msg_id=str(uuid.uuid4()),
                      inst_name=f'cattle-{host_id}')

        asyncio.create_task(queue.put(msg))
        logging.info(f'[{publisher_id}] Published {msg}')

        # simulate randomness of publishing messages
        await asyncio.sleep(random.random())
```

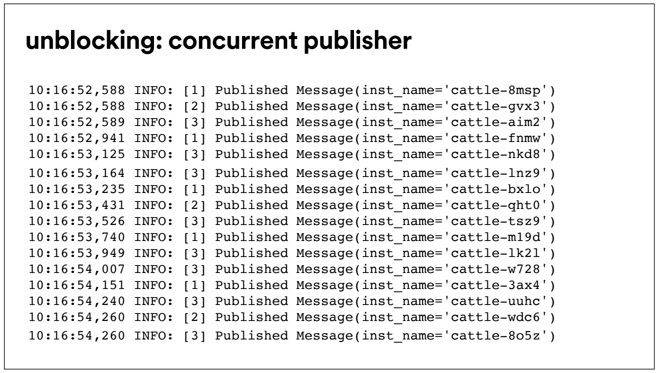
I’ll just add another argument for a publisher ID to be logged on every publish of a message



unblocking: concurrent publisher

```python
if __name__ == '__main__':
    # <--snip-->
    coros = [
        handle_exception(publish(queue, i), loop)
        for i in range(1, 4)
    ]

    try:
        [loop.create_task(coro) for coro in coros]
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

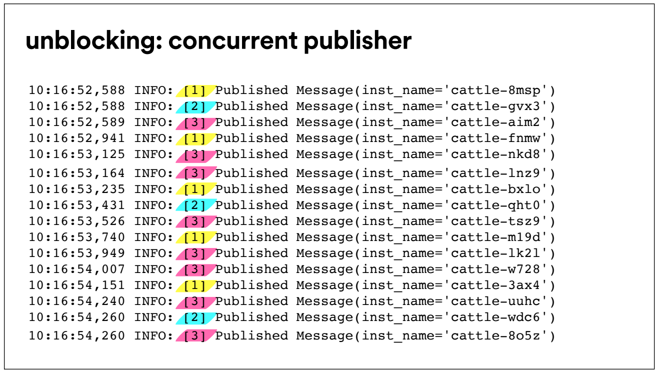
We'll create three publishers real quick, and when running:



unblocking: concurrent publisher

```
10:16:52,588 INFO: [1] Published Message(inst_name='cattle-8msp')
10:16:52,588 INFO: [2] Published Message(inst_name='cattle-gvx3')
10:16:52,589 INFO: [3] Published Message(inst_name='cattle-aim2')
10:16:52,941 INFO: [1] Published Message(inst_name='cattle-fnmw')
10:16:53,125 INFO: [3] Published Message(inst_name='cattle-nkd8')
10:16:53,164 INFO: [3] Published Message(inst_name='cattle-lnz9')
10:16:53,235 INFO: [1] Published Message(inst_name='cattle-bxlo')
10:16:53,431 INFO: [2] Published Message(inst_name='cattle-qht0')
10:16:53,526 INFO: [3] Published Message(inst_name='cattle-tsz9')
10:16:53,740 INFO: [1] Published Message(inst_name='cattle-m19d')
10:16:53,949 INFO: [3] Published Message(inst_name='cattle-lk2l')
10:16:54,007 INFO: [3] Published Message(inst_name='cattle-w728')
10:16:54,151 INFO: [1] Published Message(inst_name='cattle-3ax4')
10:16:54,240 INFO: [3] Published Message(inst_name='cattle-uuhc')
10:16:54,260 INFO: [2] Published Message(inst_name='cattle-wdc6')
10:16:54,260 INFO: [3] Published Message(inst_name='cattle-8o5z')
```

Huzzah! …

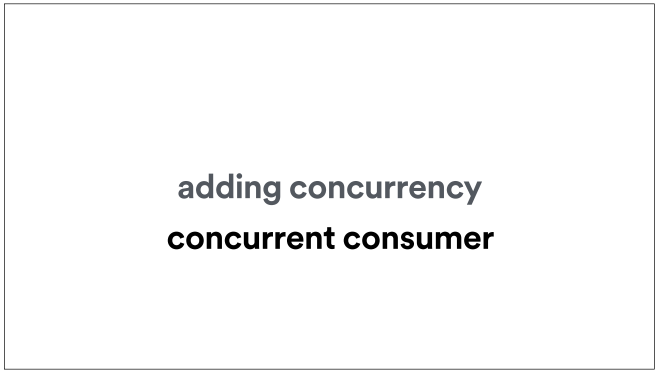


unblocking: concurrent publisher

```
10:16:52,588 INFO: [1] Published Message(inst_name='cattle-8msp')
10:16:52,588 INFO: [2] Published Message(inst_name='cattle-gvx3')
10:16:52,589 INFO: [3] Published Message(inst_name='cattle-aim2')
10:16:52,941 INFO: [1] Published Message(inst_name='cattle-fnmw')
10:16:53,125 INFO: [3] Published Message(inst_name='cattle-nkd8')
10:16:53,164 INFO: [3] Published Message(inst_name='cattle-lnz9')
10:16:53,235 INFO: [1] Published Message(inst_name='cattle-bxlo')
10:16:53,431 INFO: [2] Published Message(inst_name='cattle-qht0')
10:16:53,526 INFO: [3] Published Message(inst_name='cattle-tsz9')
10:16:53,740 INFO: [1] Published Message(inst_name='cattle-m19d')
10:16:53,949 INFO: [3] Published Message(inst_name='cattle-lk2l')
10:16:54,007 INFO: [3] Published Message(inst_name='cattle-w728')
10:16:54,151 INFO: [1] Published Message(inst_name='cattle-3ax4')
10:16:54,240 INFO: [3] Published Message(inst_name='cattle-uuhc')
10:16:54,260 INFO: [2] Published Message(inst_name='cattle-wdc6')
10:16:54,260 INFO: [3] Published Message(inst_name='cattle-8o5z')
```

We can see some concurrency among the publishers.

For the rest of the walk through, I'll remove the multiple publishers; this was just to easily convey that it's now concurrent, not just non-blocking.

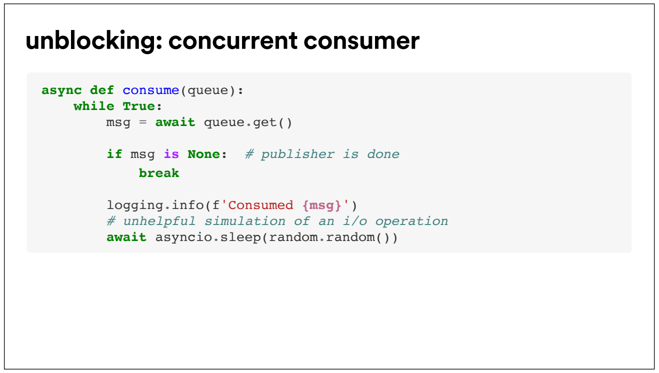


adding concurrency

concurrent consumer

Now time to add concurrency to the consumer bit. For this, the goal is to constantly consume messages from the queue and create non-blocking work based off of a newly-consumed message; in this case, to restart an instance.

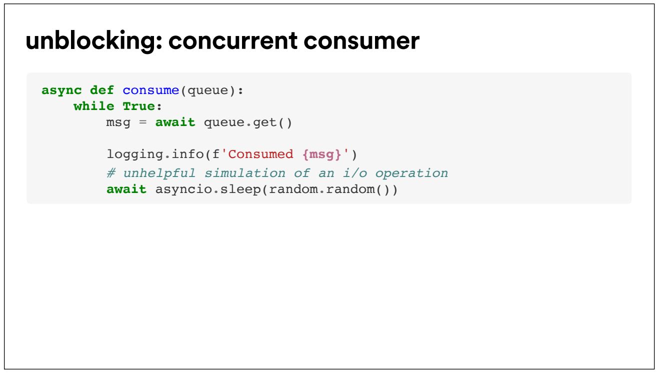
The tricky part is the consumer needs to be written in a way that the consumption of a new message from the queue is separate from when the consumption happens. In other words, we have to simulate being "event-driven" by regularly pulling for a message in the queue since there's no way to trigger work based off of a new message available in the queue (a.k.a. push-based). Remember that the producer coroutine function is merely meant to simulate an external pub/sub like Google Cloud Pub/Sub (not promoting, just most familiar).



unblocking: concurrent consumer

```
async def consume(queue):
    while True:
        msg = await queue.get()

        if msg is None:  # publisher is done
            break

        logging.info(f'Consumed {msg}')
        # unhelpful simulation of an i/o operation
        await asyncio.sleep(random.random())
```

Looking again at where we're starting from. We'll just remove the `if msg is None` part,

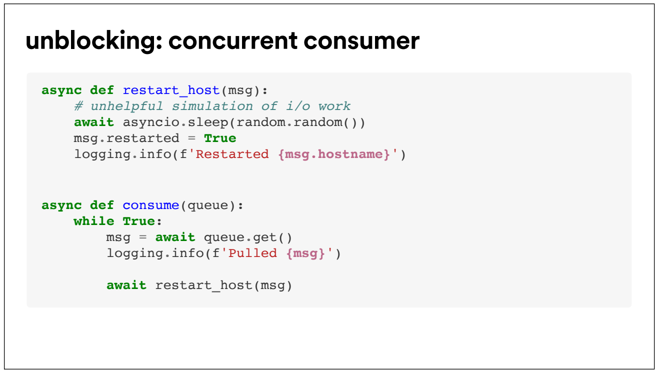


unblocking: concurrent consumer

```
async def consume(queue):
    while True:
        msg = await queue.get()

        logging.info(f'Consumed {msg}')
        # unhelpful simulation of an i/o operation
        await asyncio.sleep(random.random())
```

…since the publisher won't ever be done.

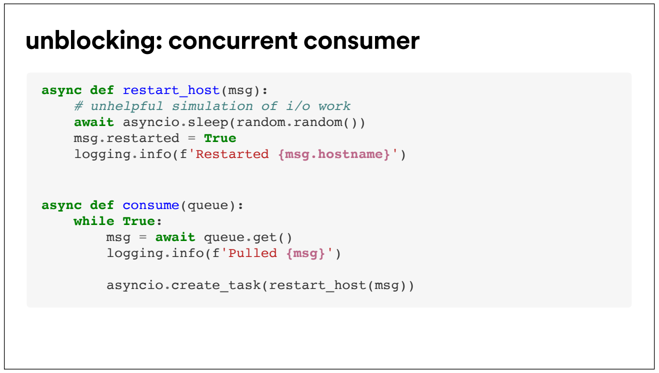
Now instead of an `asyncio.sleep` within the consumer, we'll



unblocking: concurrent consumer

```
async def restart_host(msg):
    # unhelpful simulation of i/o work
    await asyncio.sleep(random.random())
    msg.restarted = True
    logging.info(f'Restarted {msg.hostname}')


async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        await restart_host(msg)
```

Write a coroutine function that mocks the restart work that needs to be done on any consumed message. Just separate out the faked io work that’s done based on a message versus actually consuming the message.



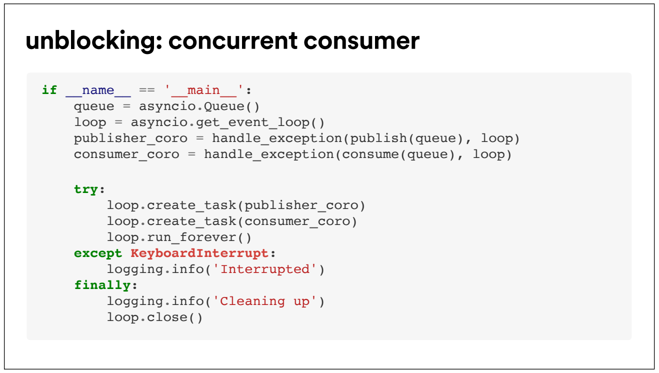
unblocking: concurrent consumer

```
async def restart_host(msg):
    # unhelpful simulation of i/o work
    await asyncio.sleep(random.random())
    msg.restarted = True
    logging.info(f'Restarted {msg.hostname}')


async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(restart_host(msg))
```

We’ll also create a task out of restart_host. For similar reasons with the publisher, we don’t want to unnecessarily block the consumption of messages.

We will leave the `await` for the `queue.get`. It makes sense to block on this because we can’t do much if there are no messages to consume.

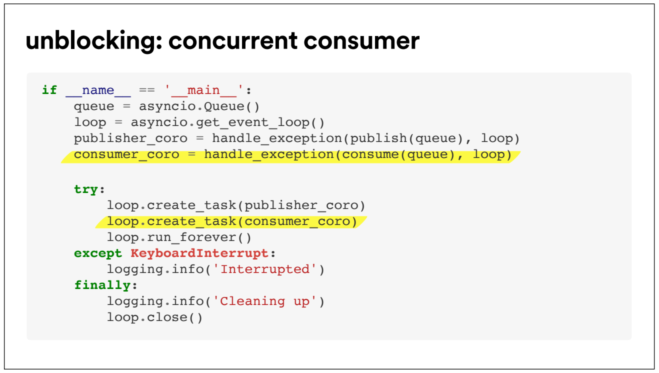
I should note that when I say that this await blocks, it doesn’t block the event loop. Other tasks on the loop will continue. It just blocks the logic that comes after it.



unblocking: concurrent consumer

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    loop = asyncio.get_event_loop()
    publisher_coro = handle_exception(publish(queue), loop)
    consumer_coro = handle_exception(consume(queue), loop)

    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

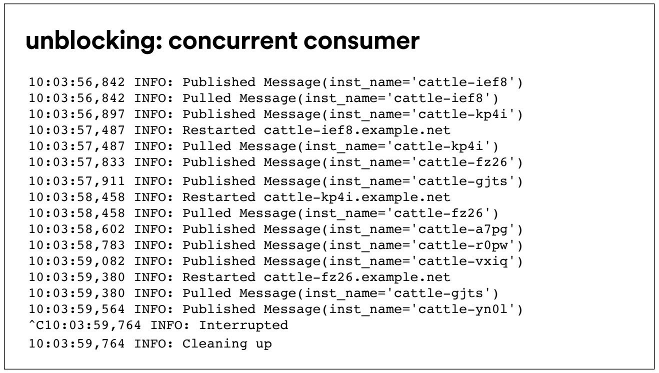
Ok so then re-adding…



unblocking: concurrent consumer

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    loop = asyncio.get_event_loop()
    publisher_coro = handle_exception(publish(queue), loop)
    consumer_coro = handle_exception(consume(queue), loop)

    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

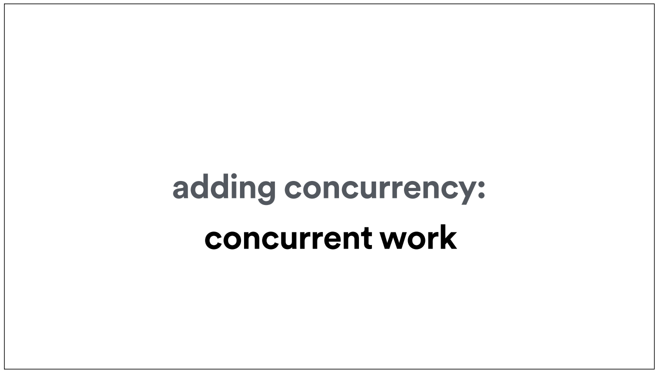
… it to our main section



unblocking: concurrent consumer

```
10:03:56,842 INFO: Published Message(inst_name='cattle-ief8')
10:03:56,842 INFO: Pulled Message(inst_name='cattle-ief8')
10:03:56,897 INFO: Published Message(inst_name='cattle-kp4i')
10:03:57,487 INFO: Restarted cattle-ief8.example.net
10:03:57,487 INFO: Pulled Message(inst_name='cattle-kp4i')
10:03:57,833 INFO: Published Message(inst_name='cattle-fz26')
10:03:57,911 INFO: Published Message(inst_name='cattle-gjts')
10:03:58,458 INFO: Restarted cattle-kp4i.example.net
10:03:58,458 INFO: Pulled Message(inst_name='cattle-fz26')
10:03:58,602 INFO: Published Message(inst_name='cattle-a7pg')
10:03:58,783 INFO: Published Message(inst_name='cattle-r0pw')
10:03:59,082 INFO: Published Message(inst_name='cattle-vxiq')
10:03:59,380 INFO: Restarted cattle-fz26.example.net
10:03:59,380 INFO: Pulled Message(inst_name='cattle-gjts')
10:03:59,564 INFO: Published Message(inst_name='cattle-yn0l')
^C10:03:59,764 INFO: Interrupted
10:03:59,764 INFO: Cleaning up
```

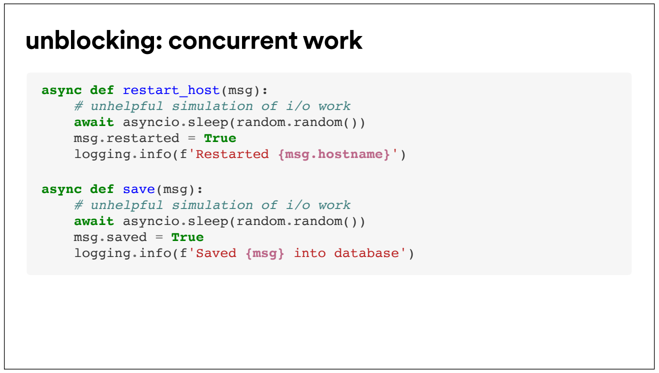
Nice. We're now pulling for messages whenever they're available.



adding concurrency:

concurrent work

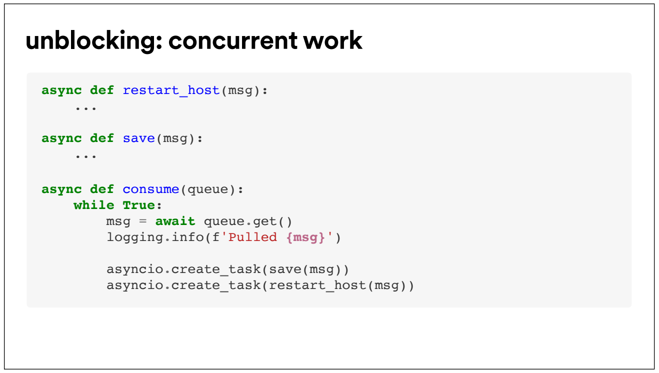
We may want to do more than one thing per message.



unblocking: concurrent work

```
async def restart_host(msg):
    # unhelpful simulation of i/o work
    await asyncio.sleep(random.random())
    msg.restarted = True
    logging.info(f'Restarted {msg.hostname}')

async def save(msg):
    # unhelpful simulation of i/o work
    await asyncio.sleep(random.random())
    msg.saved = True
    logging.info(f'Saved {msg} into database')
```

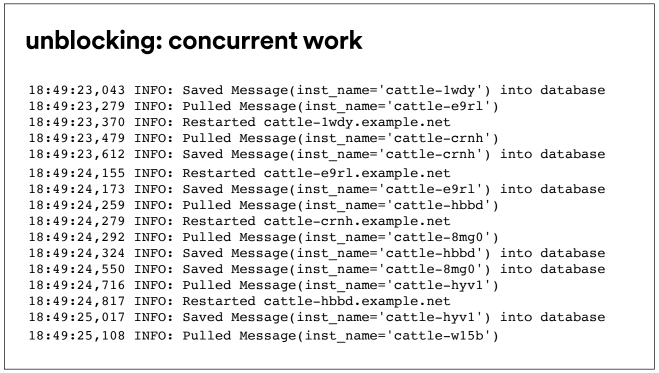
For example, we'd like to store the message in a database for potentially replaying later as well as initiate a restart of the given host:



unblocking: concurrent work

```
async def restart_host(msg):
    ...

async def save(msg):
    ...

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(save(msg))
        asyncio.create_task(restart_host(msg))
```

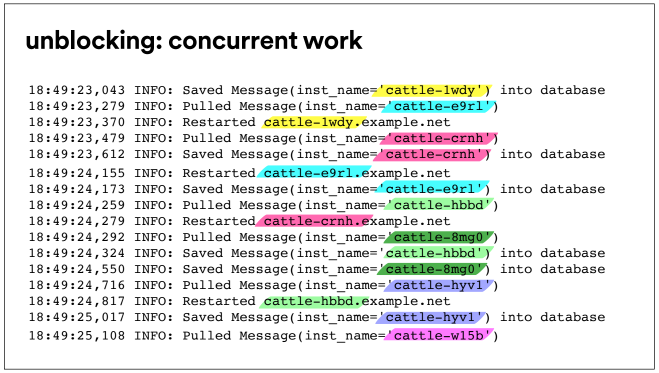
We'll make use asyncio.create_task again for the save coroutine to be scheduled on the loop, basically chucking it over to the loop for it to execute when it next can.



unblocking: concurrent work

```
18:49:23,043 INFO: Saved Message(inst_name='cattle-1wdy') into database
18:49:23,279 INFO: Pulled Message(inst_name='cattle-e9rl')
18:49:23,370 INFO: Restarted cattle-1wdy.example.net
18:49:23,479 INFO: Pulled Message(inst_name='cattle-crnh')
18:49:23,612 INFO: Saved Message(inst_name='cattle-crnh') into database
18:49:24,155 INFO: Restarted cattle-e9rl.example.net
18:49:24,173 INFO: Saved Message(inst_name='cattle-e9rl') into database
18:49:24,259 INFO: Pulled Message(inst_name='cattle-hbbd')
18:49:24,279 INFO: Restarted cattle-crnh.example.net
18:49:24,292 INFO: Pulled Message(inst_name='cattle-8mg0')
18:49:24,324 INFO: Saved Message(inst_name='cattle-hbbd') into database
18:49:24,550 INFO: Saved Message(inst_name='cattle-8mg0') into database
18:49:24,716 INFO: Pulled Message(inst_name='cattle-hyv1')
18:49:24,817 INFO: Restarted cattle-hbbd.example.net
18:49:25,017 INFO: Saved Message(inst_name='cattle-hyv1') into database
18:49:25,108 INFO: Pulled Message(inst_name='cattle-w15b')
```

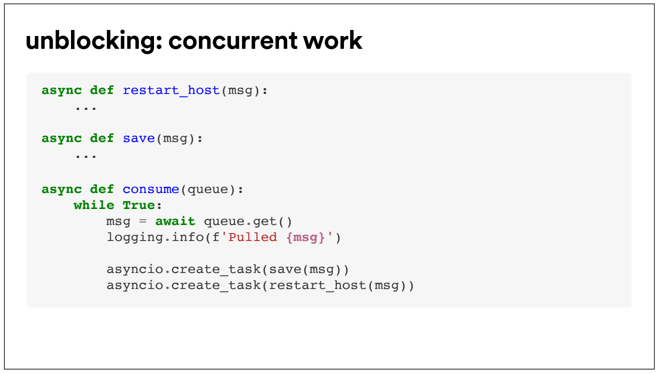
Running it, we see



unblocking: concurrent work

```
18:49:23,043 INFO: Saved Message(inst_name='cattle-1wdy') into database
18:49:23,279 INFO: Pulled Message(inst_name='cattle-e9rl')
18:49:23,370 INFO: Restarted cattle-1wdy.example.net
18:49:23,479 INFO: Pulled Message(inst_name='cattle-crnh')
18:49:23,612 INFO: Saved Message(inst_name='cattle-crnh') into database
18:49:24,155 INFO: Restarted cattle-e9rl.example.net
18:49:24,173 INFO: Saved Message(inst_name='cattle-e9rl') into database
18:49:24,259 INFO: Pulled Message(inst_name='cattle-hbbd')
18:49:24,279 INFO: Restarted cattle-crnh.example.net
18:49:24,292 INFO: Pulled Message(inst_name='cattle-8mg0')
18:49:24,324 INFO: Saved Message(inst_name='cattle-hbbd') into database
18:49:24,550 INFO: Saved Message(inst_name='cattle-8mg0') into database
18:49:24,716 INFO: Pulled Message(inst_name='cattle-hyv1')
18:49:24,817 INFO: Restarted cattle-hbbd.example.net
18:49:25,017 INFO: Saved Message(inst_name='cattle-hyv1') into database
18:49:25,108 INFO: Pulled Message(inst_name='cattle-w15b')
```

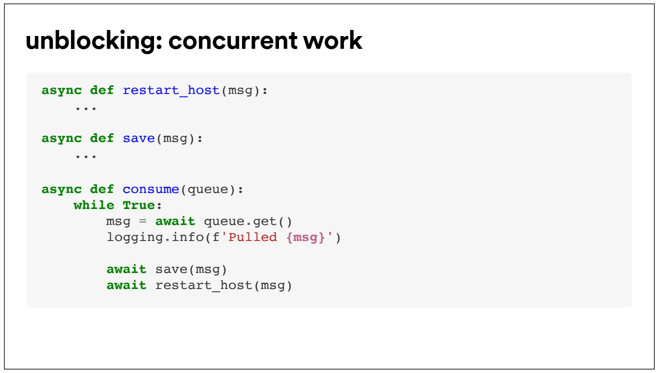
That for every message, we may either restart or save it first, showing some asynchronous work going on; we also don’t block the consumption of messages with the work we do generated by each message.



unblocking: concurrent work

```python
async def restart_host(msg):
    ...

async def save(msg):
    ...

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(save(msg))
        asyncio.create_task(restart_host(msg))
```

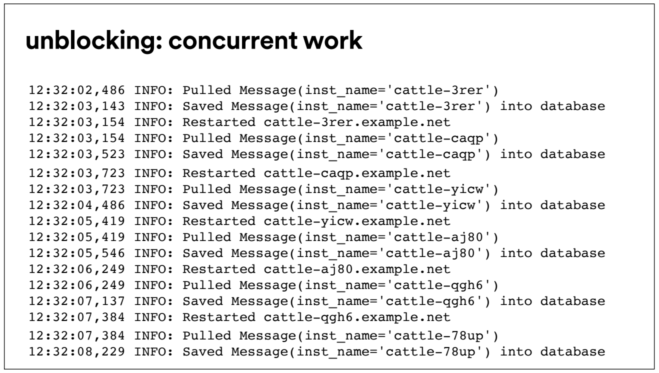
To illustrate what I mean a little better, let’s switch



unblocking: concurrent work

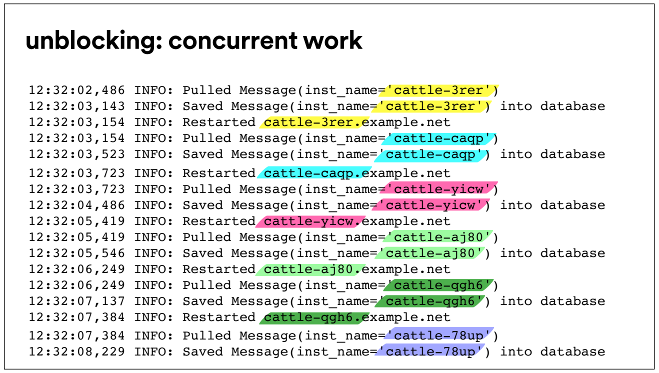
```python
async def restart_host(msg):
    ...

async def save(msg):
    ...

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        await save(msg)
        await restart_host(msg)
```

create_task for save and restart, to awaiting them.



unblocking: concurrent work

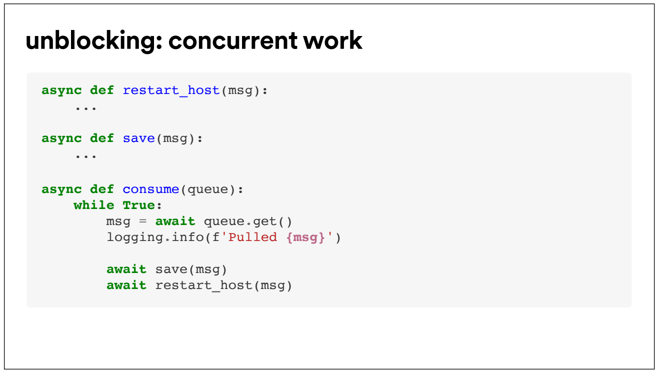
```
12:32:02,486 INFO: Pulled Message(inst_name='cattle-3rer')
12:32:03,143 INFO: Saved Message(inst_name='cattle-3rer') into database
12:32:03,154 INFO: Restarted cattle-3rer.example.net
12:32:03,154 INFO: Pulled Message(inst_name='cattle-caqp')
12:32:03,523 INFO: Saved Message(inst_name='cattle-caqp') into database
12:32:03,723 INFO: Restarted cattle-caqp.example.net
12:32:03,723 INFO: Pulled Message(inst_name='cattle-yicw')
12:32:04,486 INFO: Saved Message(inst_name='cattle-yicw') into database
12:32:05,419 INFO: Restarted cattle-yicw.example.net
12:32:05,419 INFO: Pulled Message(inst_name='cattle-aj80')
12:32:05,546 INFO: Saved Message(inst_name='cattle-aj80') into database
12:32:06,249 INFO: Restarted cattle-aj80.example.net
12:32:06,249 INFO: Pulled Message(inst_name='cattle-qgh6')
12:32:07,137 INFO: Saved Message(inst_name='cattle-qgh6') into database
12:32:07,384 INFO: Restarted cattle-qgh6.example.net
12:32:07,384 INFO: Pulled Message(inst_name='cattle-78up')
12:32:08,229 INFO: Saved Message(inst_name='cattle-78up') into database
```



unblocking: concurrent work

```
12:32:02,486 INFO: Pulled Message(inst_name='cattle-3rer')
12:32:03,143 INFO: Saved Message(inst_name='cattle-3rer') into database
12:32:03,154 INFO: Restarted cattle-3rer.example.net
12:32:03,154 INFO: Pulled Message(inst_name='cattle-caqp')
12:32:03,523 INFO: Saved Message(inst_name='cattle-caqp') into database
12:32:03,723 INFO: Restarted cattle-caqp.example.net
12:32:03,723 INFO: Pulled Message(inst_name='cattle-yicw')
12:32:04,486 INFO: Saved Message(inst_name='cattle-yicw') into database
12:32:05,419 INFO: Restarted cattle-yicw.example.net
12:32:05,419 INFO: Pulled Message(inst_name='cattle-aj80')
12:32:05,546 INFO: Saved Message(inst_name='cattle-aj80') into database
12:32:06,249 INFO: Restarted cattle-aj80.example.net
12:32:06,249 INFO: Pulled Message(inst_name='cattle-qgh6')
12:32:07,137 INFO: Saved Message(inst_name='cattle-qgh6') into database
12:32:07,384 INFO: Restarted cattle-qgh6.example.net
12:32:07,384 INFO: Pulled Message(inst_name='cattle-78up')
12:32:08,229 INFO: Saved Message(inst_name='cattle-78up') into database
```

We can see that although it doesn't block the event loop, await save(msg) blocks await restart_host(msg), which blocks the consumption of future messages. There’s a clear order here.

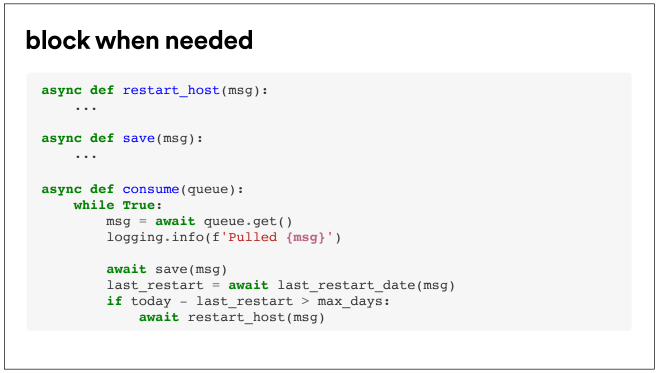


unblocking: concurrent work

```python
async def restart_host(msg):
    ...

async def save(msg):
    ...

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        await save(msg)
        await restart_host(msg)
```

But maybe you want to your work to happen serially. You may not _want_ to have concurrency for some asynchronous tasks

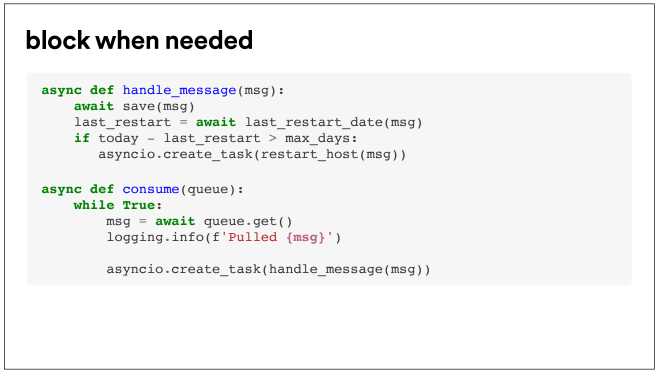
These two tasks don't necessarily need to depend on one another – completely side-stepping the potential concern/complexity of "should we restart a host if we fail to add the message to the database".



block when needed

```python
async def restart_host(msg):
    ...

async def save(msg):
    ...

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        await save(msg)
        last_restart = await last_restart_date(msg)
        if today - last_restart > max_days:
            await restart_host(msg)
```

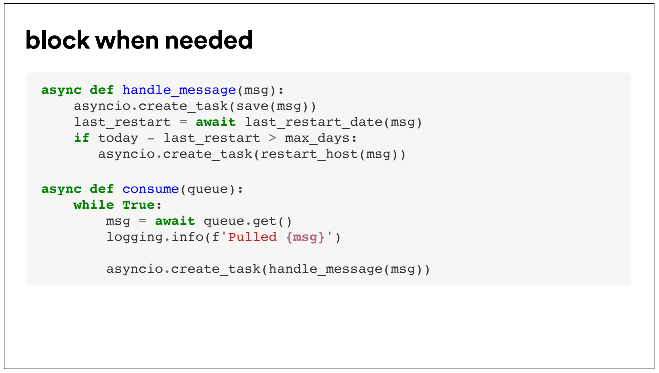
Maybe you restart hosts that have an uptime of more than 7 days. Or maybe you should check the balance of an account before you debit it. Needing code to be serial, to have steps or dependencies, it doesn't mean that you can't be asynchronous. The await last_restart_date will yield to the loop, but it doesn't mean that restart_host will be the next thing that the loop executes. It just allows other things to happen outside of this coroutine.



block when needed

```python
async def handle_message(msg):
    await save(msg)
    last_restart = await last_restart_date(msg)
    if today - last_restart > max_days:
        asyncio.create_task(restart_host(msg))

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(handle_message(msg))
```

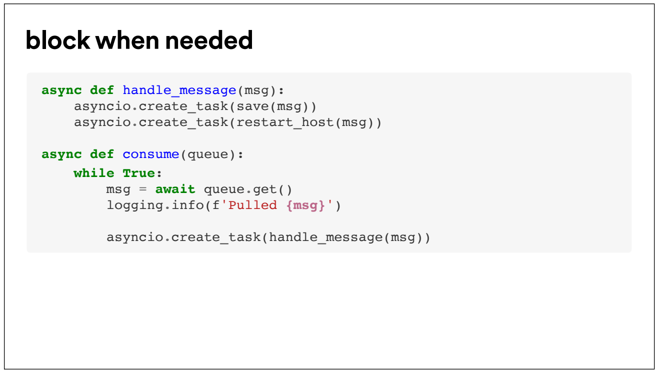
I'll just put all this message-related logic in a separate coroutine function, so we don't have to block the consumption of messages….



block when needed

```python
async def handle_message(msg):
    asyncio.create_task(save(msg))
    last_restart = await last_restart_date(msg)
    if today - last_restart > max_days:
        asyncio.create_task(restart_host(msg))

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(handle_message(msg))
```

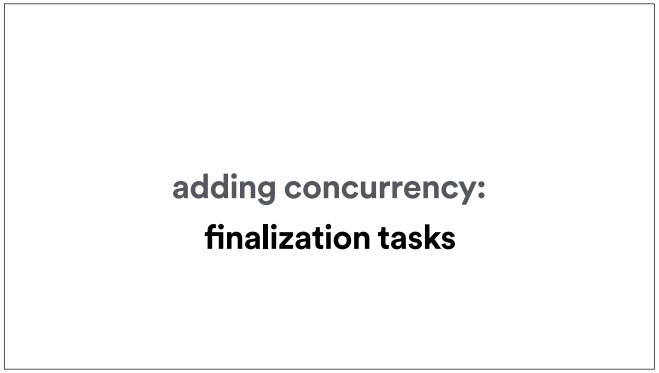
Saving a message shouldn't block a restart of a host if needed, so we'll return that to being a task.



block when needed

```python
async def handle_message(msg):
    asyncio.create_task(save(msg))
    asyncio.create_task(restart_host(msg))

async def consume(queue):
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(handle_message(msg))
```

For simplicity, pretend I moved this checking of last restart date logic out of the handle message and into the restart host function.

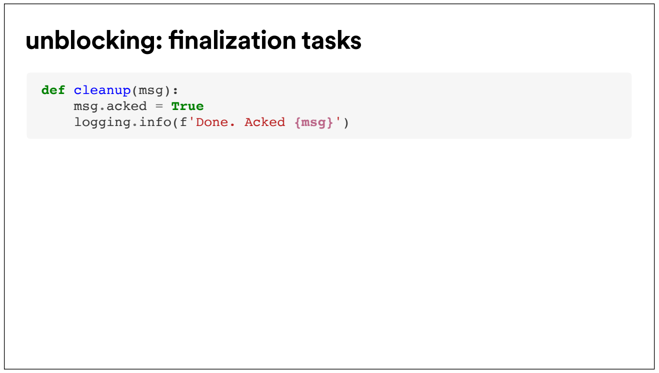


adding concurrency:

finalization tasks

We've pulled a message from the queue, and fanned out work based off of that message.

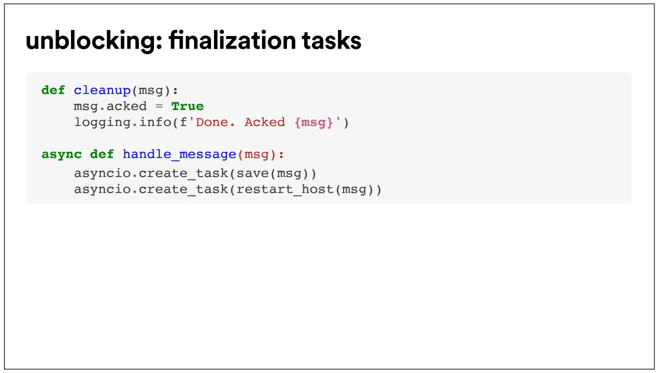
Now we need to perform any finalizing work on that message;



unblocking: finalization tasks

```python
def cleanup(msg):
    msg.acked = True
    logging.info(f'Done. Acked {msg}')
```

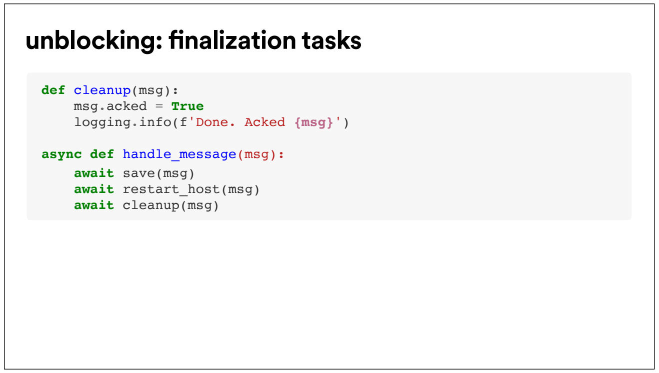
for example, acknowledging the message so it isn't re-delivered.



unblocking: finalization tasks

```python
def cleanup(msg):
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def handle_message(msg):
    asyncio.create_task(save(msg))
    asyncio.create_task(restart_host(msg))
```

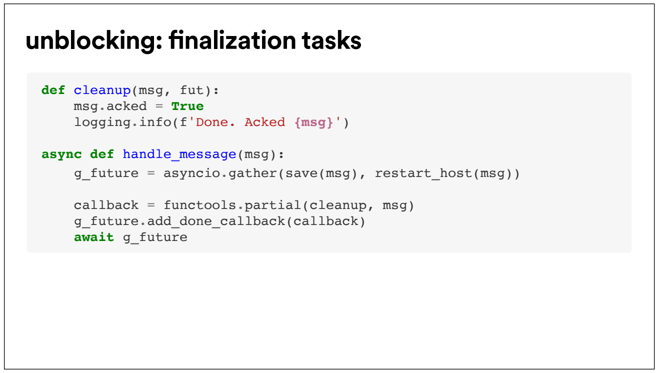
We currently have two separate tasks, `save` and `restart_host`, and we want to make sure both are done before the message is cleaned up.



unblocking: finalization tasks

```python
def cleanup(msg):
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def handle_message(msg):
    await save(msg)
    await restart_host(msg)
    await cleanup(msg)
```

We could go back to the sequential `await`s since that's a very direct way to manipulate the ordering.



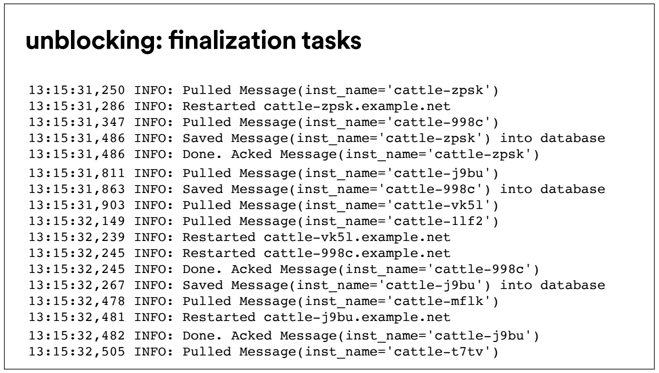
unblocking: finalization tasks

```python
def cleanup(msg, fut):
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def handle_message(msg):
    g_future = asyncio.gather(save(msg), restart_host(msg))

    callback = functools.partial(cleanup, msg)
    g_future.add_done_callback(callback)
    await g_future
```

But we can also use callbacks on a completed task. What we therefore want is to somehow have a task that wraps around the two tasks of `save` and `restart_host`, since we have to wait for both to finish before cleaning up can happen.

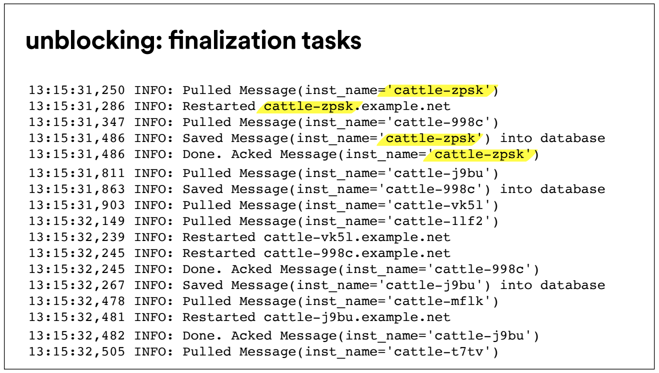
We can make use of asyncio.gather, which returns a future-like object, to which we can attach the callback of "cleanup" via `add_done_callback`.

We can now just await that future in order to kick off the `save` and `restart_host` coroutines. And - obviously - the callback of "cleanup" will be called once those two are done.



unblocking: finalization tasks

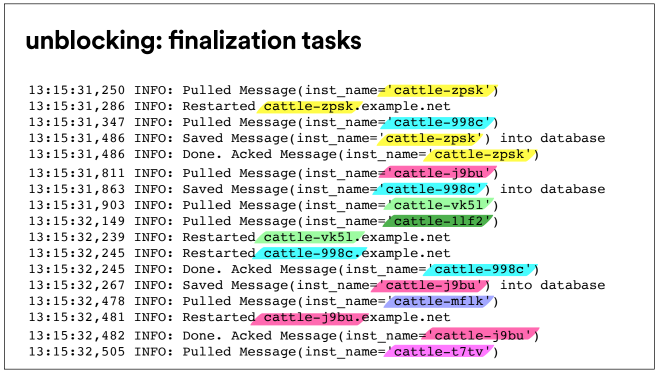
```
13:15:31,250 INFO: Pulled Message(inst_name='cattle-zpsk')
13:15:31,286 INFO: Restarted cattle-zpsk.example.net
13:15:31,347 INFO: Pulled Message(inst_name='cattle-998c')
13:15:31,486 INFO: Saved Message(inst_name='cattle-zpsk') into database
13:15:31,486 INFO: Done. Acked Message(inst_name='cattle-zpsk')
13:15:31,811 INFO: Pulled Message(inst_name='cattle-j9bu')
13:15:31,863 INFO: Saved Message(inst_name='cattle-998c') into database
13:15:31,903 INFO: Pulled Message(inst_name='cattle-vk5l')
13:15:32,149 INFO: Pulled Message(inst_name='cattle-1lf2')
13:15:32,239 INFO: Restarted cattle-vk5l.example.net
13:15:32,245 INFO: Restarted cattle-998c.example.net
13:15:32,245 INFO: Done. Acked Message(inst_name='cattle-998c')
13:15:32,267 INFO: Saved Message(inst_name='cattle-j9bu') into database
13:15:32,478 INFO: Pulled Message(inst_name='cattle-mflk')
13:15:32,481 INFO: Restarted cattle-j9bu.example.net
13:15:32,482 INFO: Done. Acked Message(inst_name='cattle-j9bu')
13:15:32,505 INFO: Pulled Message(inst_name='cattle-t7tv')
```



unblocking: finalization tasks

```
13:15:31,250 INFO: Pulled Message(inst_name='cattle-zpsk')
13:15:31,286 INFO: Restarted cattle-zpsk.example.net
13:15:31,347 INFO: Pulled Message(inst_name='cattle-998c')
13:15:31,486 INFO: Saved Message(inst_name='cattle-zpsk') into database
13:15:31,486 INFO: Done. Acked Message(inst_name='cattle-zpsk')
13:15:31,811 INFO: Pulled Message(inst_name='cattle-j9bu')
13:15:31,863 INFO: Saved Message(inst_name='cattle-998c') into database
13:15:31,903 INFO: Pulled Message(inst_name='cattle-vk5l')
13:15:32,149 INFO: Pulled Message(inst_name='cattle-1lf2')
13:15:32,239 INFO: Restarted cattle-vk5l.example.net
13:15:32,245 INFO: Restarted cattle-998c.example.net
13:15:32,245 INFO: Done. Acked Message(inst_name='cattle-998c')
13:15:32,267 INFO: Saved Message(inst_name='cattle-j9bu') into database
13:15:32,478 INFO: Pulled Message(inst_name='cattle-mflk')
13:15:32,481 INFO: Restarted cattle-j9bu.example.net
13:15:32,482 INFO: Done. Acked Message(inst_name='cattle-j9bu')
13:15:32,505 INFO: Pulled Message(inst_name='cattle-t7tv')
```

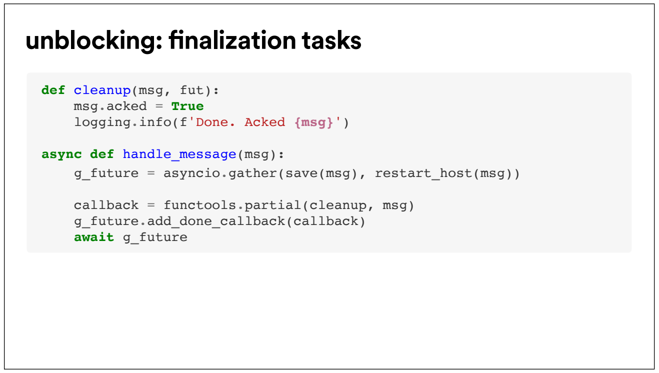
So once's both save coroutine and restart coroutine are complete, cleanup will be called that signifies a message is completely done:



unblocking: finalization tasks

```
13:15:31,250 INFO: Pulled Message(inst_name='cattle-zpsk')
13:15:31,286 INFO: Restarted cattle-zpsk.example.net
13:15:31,347 INFO: Pulled Message(inst_name='cattle-998c')
13:15:31,486 INFO: Saved Message(inst_name='cattle-zpsk') into database
13:15:31,486 INFO: Done. Acked Message(inst_name='cattle-zpsk')
13:15:31,811 INFO: Pulled Message(inst_name='cattle-j9bu')
13:15:31,863 INFO: Saved Message(inst_name='cattle-998c') into database
13:15:31,903 INFO: Pulled Message(inst_name='cattle-vk5l')
13:15:32,149 INFO: Pulled Message(inst_name='cattle-1lf2')
13:15:32,239 INFO: Restarted cattle-vk5l.example.net
13:15:32,245 INFO: Restarted cattle-998c.example.net
13:15:32,245 INFO: Done. Acked Message(inst_name='cattle-998c')
13:15:32,267 INFO: Saved Message(inst_name='cattle-j9bu') into database
13:15:32,478 INFO: Pulled Message(inst_name='cattle-mflk')
13:15:32,481 INFO: Restarted cattle-j9bu.example.net
13:15:32,482 INFO: Done. Acked Message(inst_name='cattle-j9bu')
13:15:32,505 INFO: Pulled Message(inst_name='cattle-t7tv')
```

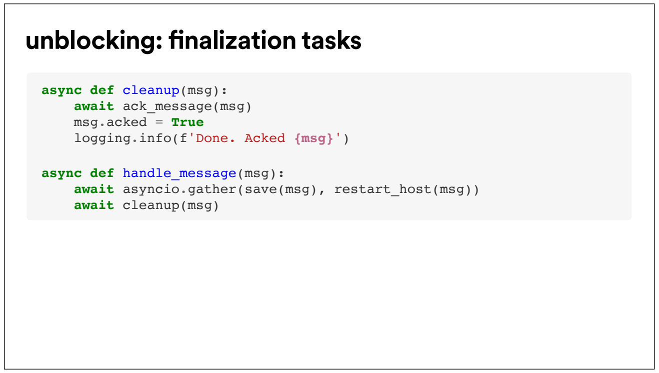
And we’ve still maintained appropriate concurrency.



unblocking: finalization tasks

```python
def cleanup(msg, fut):
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def handle_message(msg):
    g_future = asyncio.gather(save(msg), restart_host(msg))

    callback = functools.partial(cleanup, msg)
    g_future.add_done_callback(callback)
    await g_future
```

I personally have an allergy to callbacks.

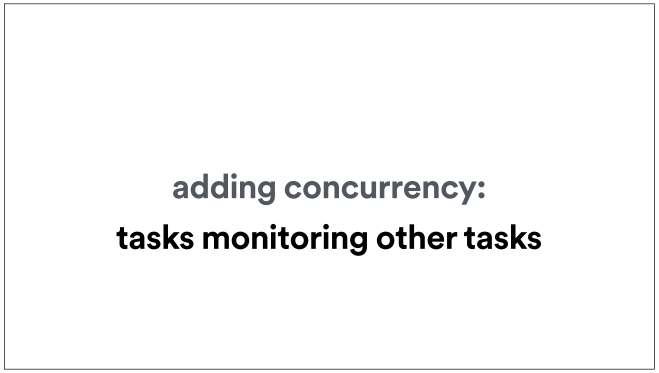


unblocking: finalization tasks

```
async def cleanup(msg):
    await ack_message(msg)
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def handle_message(msg):
    await asyncio.gather(save(msg), restart_host(msg))
    await cleanup(msg)
```

As well, perhaps we need cleanup to be non-blocking.

Another approach could be just to await it since the order of operations does matter.



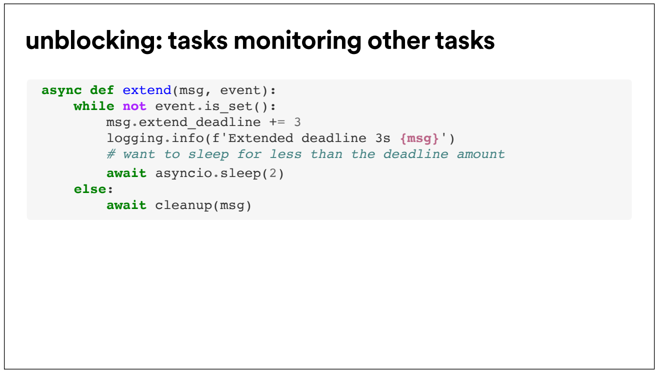
adding concurrency:

tasks monitoring other tasks

Now, much like Google's Pub/Sub, let's say that the publisher will redeliver a message after 10 seconds if it has not been acknowledged. We are able to extend that "timeout" period or acknowledgement deadline for a message.

In order to do that, we now have to have a coroutine that, in essence, monitors all the other worker tasks.

While our tasks are working to restart hosts and save the pulled message, we'll have another coroutine will extend the message acknowledgement deadline; then once they're done, it should stop extending and cleanup the message.

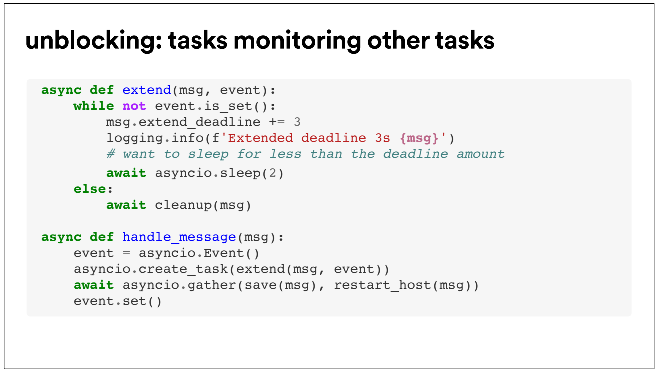


unblocking: tasks monitoring other tasks

```python
async def extend(msg, event):
    while not event.is_set():
        msg.extend_deadline += 3
        logging.info(f'Extended deadline 3s {msg}')
        # want to sleep for less than the deadline amount
        await asyncio.sleep(2)
    else:
        await cleanup(msg)
```

One approach is to make use of asyncio.Event[1] primatives. An event instance essentially just provides is with a boolean flag. Our `extend` coroutine will essentially loop while that flag is not yet set, continually extending the message's deadline until we're done.

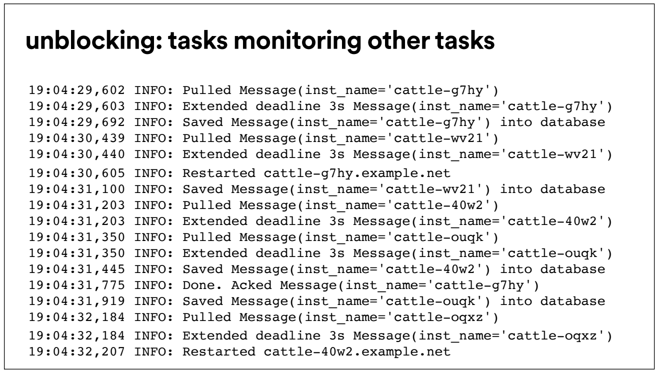
[1] https://docs.python.org/3/library/asyncio-sync.html#event



unblocking: tasks monitoring other tasks

```python
async def extend(msg, event):
    while not event.is_set():
        msg.extend_deadline += 3
        logging.info(f'Extended deadline 3s {msg}')
        # want to sleep for less than the deadline amount
        await asyncio.sleep(2)
    else:
        await cleanup(msg)

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    await asyncio.gather(save(msg), restart_host(msg))
    event.set()
```

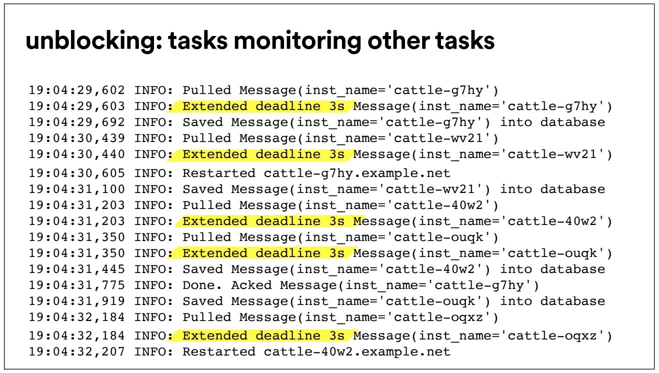
And then we update our `handle_message` function: we'll create an event instance to give to the `extend` coroutine, and create a task out of it, basically "fire and forget" the task. We still await the `asyncio.gather`, and then once we're done, we `set` the event to cue the `extend` function to start the cleanup process.



unblocking: tasks monitoring other tasks

```
19:04:29,602 INFO: Pulled Message(inst_name='cattle-g7hy')
19:04:29,603 INFO: Extended deadline 3s Message(inst_name='cattle-g7hy')
19:04:29,692 INFO: Saved Message(inst_name='cattle-g7hy') into database
19:04:30,439 INFO: Pulled Message(inst_name='cattle-wv21')
19:04:30,440 INFO: Extended deadline 3s Message(inst_name='cattle-wv21')
19:04:30,605 INFO: Restarted cattle-g7hy.example.net
19:04:31,100 INFO: Saved Message(inst_name='cattle-wv21') into database
19:04:31,203 INFO: Pulled Message(inst_name='cattle-40w2')
19:04:31,203 INFO: Extended deadline 3s Message(inst_name='cattle-40w2')
19:04:31,350 INFO: Pulled Message(inst_name='cattle-ouqk')
19:04:31,350 INFO: Extended deadline 3s Message(inst_name='cattle-ouqk')
19:04:31,445 INFO: Saved Message(inst_name='cattle-40w2') into database
19:04:31,775 INFO: Done. Acked Message(inst_name='cattle-g7hy')
19:04:31,919 INFO: Saved Message(inst_name='cattle-ouqk') into database
19:04:32,184 INFO: Pulled Message(inst_name='cattle-oqxz')
19:04:32,184 INFO: Extended deadline 3s Message(inst_name='cattle-oqxz')
19:04:32,207 INFO: Restarted cattle-40w2.example.net
```

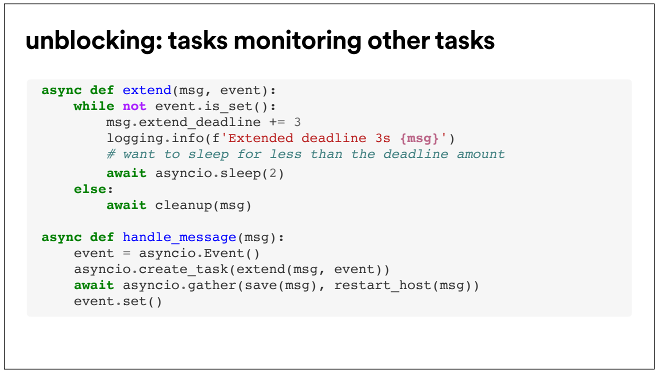
Running this,



unblocking: tasks monitoring other tasks

```
19:04:29,602 INFO: Pulled Message(inst_name='cattle-g7hy')
19:04:29,603 INFO: Extended deadline 3s Message(inst_name='cattle-g7hy')
19:04:29,692 INFO: Saved Message(inst_name='cattle-g7hy') into database
19:04:30,439 INFO: Pulled Message(inst_name='cattle-wv21')
19:04:30,440 INFO: Extended deadline 3s Message(inst_name='cattle-wv21')
19:04:30,605 INFO: Restarted cattle-g7hy.example.net
19:04:31,100 INFO: Saved Message(inst_name='cattle-wv21') into database
19:04:31,203 INFO: Pulled Message(inst_name='cattle-40w2')
19:04:31,203 INFO: Extended deadline 3s Message(inst_name='cattle-40w2')
19:04:31,350 INFO: Pulled Message(inst_name='cattle-ouqk')
19:04:31,350 INFO: Extended deadline 3s Message(inst_name='cattle-ouqk')
19:04:31,445 INFO: Saved Message(inst_name='cattle-40w2') into database
19:04:31,775 INFO: Done. Acked Message(inst_name='cattle-g7hy')
19:04:31,919 INFO: Saved Message(inst_name='cattle-ouqk') into database
19:04:32,184 INFO: Pulled Message(inst_name='cattle-oqxz')
19:04:32,184 INFO: Extended deadline 3s Message(inst_name='cattle-oqxz')
19:04:32,207 INFO: Restarted cattle-40w2.example.net
```

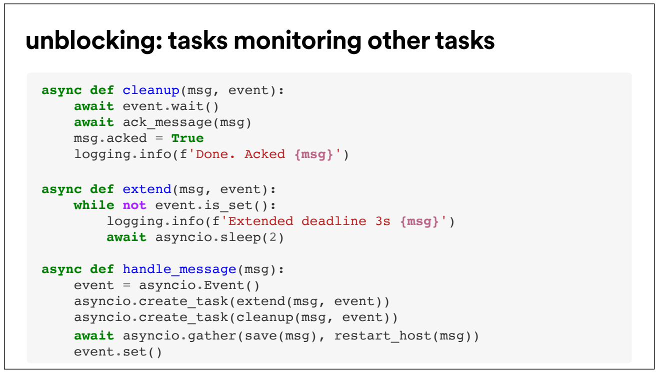
we can see we're extending while work continues, and cleaning up once done



unblocking: tasks monitoring other tasks

```python
async def extend(msg, event):
    while not event.is_set():
        msg.extend_deadline += 3
        logging.info(f'Extended deadline 3s {msg}')
        # want to sleep for less than the deadline amount
        await asyncio.sleep(2)
    else:
        await cleanup(msg)

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    await asyncio.gather(save(msg), restart_host(msg))
    event.set()
```

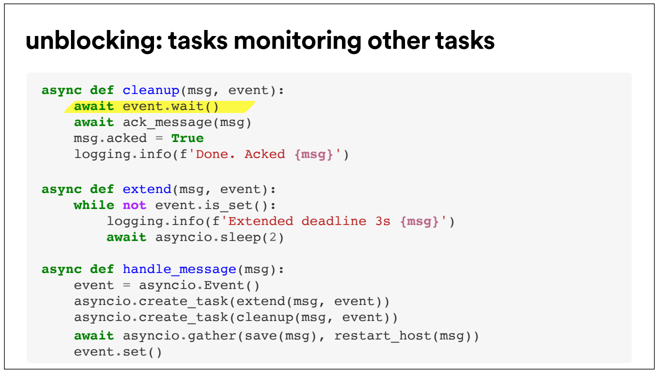
If you love events, you could adapt this a little bit…



unblocking: tasks monitoring other tasks

```python
async def cleanup(msg, event):
    await event.wait()
    await ack_message(msg)
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def extend(msg, event):
    while not event.is_set():
        logging.info(f'Extended deadline 3s {msg}')
        await asyncio.sleep(2)

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))
    await asyncio.gather(save(msg), restart_host(msg))
    event.set()
```

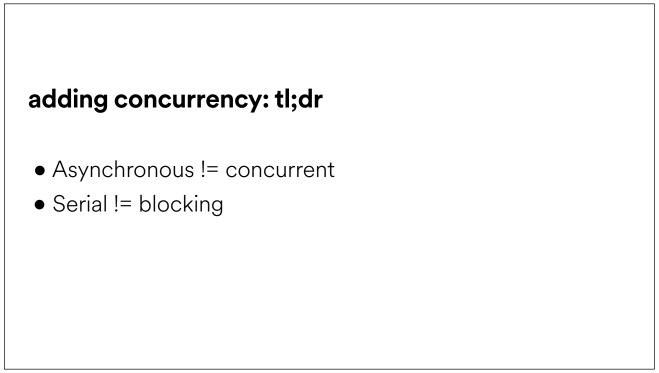
… to make use of the `event.wait` coroutine.



unblocking: tasks monitoring other tasks

```
async def cleanup(msg, event):
    await event.wait()
    await ack_message(msg)
    msg.acked = True
    logging.info(f'Done. Acked {msg}')

async def extend(msg, event):
    while not event.is_set():
        logging.info(f'Extended deadline 3s {msg}')
        await asyncio.sleep(2)

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))
    await asyncio.gather(save(msg), restart_host(msg))
    event.set()
```

Cleanup would be its own task again that you'd "fire and forget" along with `extend`, making it a bit more logical to read, so cleanup isn't called within extend, but from the same place that message-related tasks are controlled.



adding concurrency: tl;dr

- Asynchronous != concurrent
- Serial != blocking

asyncio is pretty easy to use, but being easy to use doesn't automatically mean you're using it correctly. You can't just throw around async and await keywords around blocking code. It's a shift in a mental paradigm. Both with needing to think of what work can be farmed out and let it do its thing, what dependencies there are and where your code might still need to be sequential.

But having steps within your code, having "first A, then B, then C" may seem like it's blocking when it's not. Sequential code can still be asynchronous. I might have to call customer service for something, and wait to be taken off hold to talk to them, but while I wait, I can put the phone on speaker and pet my super needy cat. I might be single-threaded as a person, but I can multi-task like CPUs.

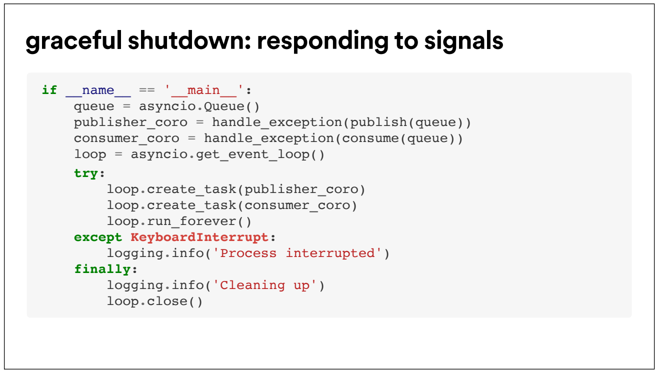
graceful shutdowns

Earlier, we added a try/except/finally around our main event loop code. Often though, you'll want your service to gracefully shutdown if it receives a signal of some sort, e.g. clean up open database connections, stop consuming messages, finish responding to current requests while not accepting new requests, etc.

graceful shutdowns

responding to signals

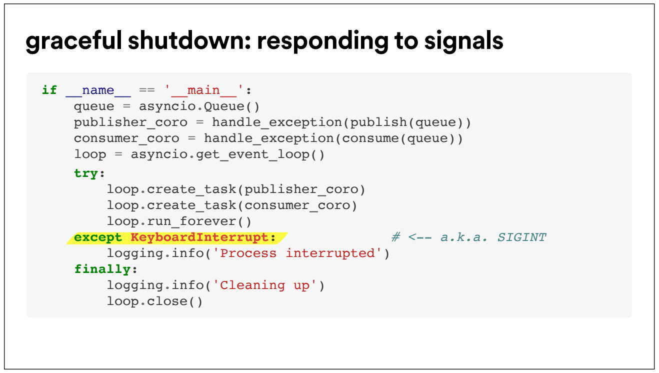
So, if we happen to restart an instance of our own chaos monkey-like service, we should clean up the "mess" we've made before exiting out.



graceful shutdown: responding to signals

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue))
    consumer_coro = handle_exception(consume(queue))
    loop = asyncio.get_event_loop()
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Process interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

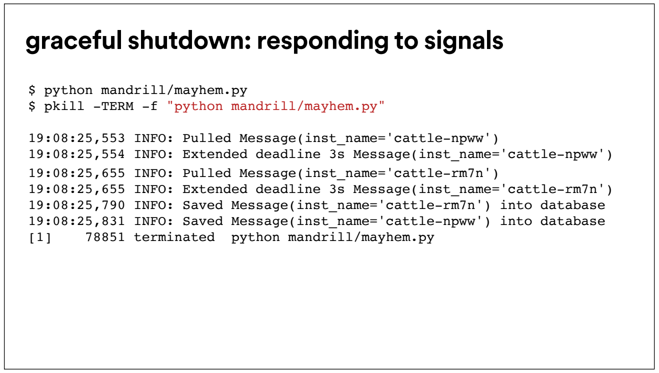
We've been catching the commonly-known…



graceful shutdown: responding to signals

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue))
    consumer_coro = handle_exception(consume(queue))
    loop = asyncio.get_event_loop()
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:              # <-- a.k.a. SIGINT
        logging.info('Process interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

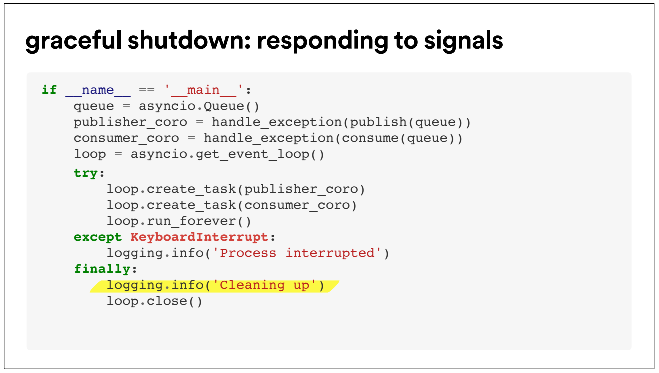
…KeyboardInterrupt exception like many other tutorials and libraries.



graceful shutdown: responding to signals

```
$ python mandrill/mayhem.py
$ pkill -TERM -f "python mandrill/mayhem.py"

19:08:25,553 INFO: Pulled Message(inst_name='cattle-npww')
19:08:25,554 INFO: Extended deadline 3s Message(inst_name='cattle-npww')
19:08:25,655 INFO: Pulled Message(inst_name='cattle-rm7n')
19:08:25,655 INFO: Extended deadline 3s Message(inst_name='cattle-rm7n')
19:08:25,790 INFO: Saved Message(inst_name='cattle-rm7n') into database
19:08:25,831 INFO: Saved Message(inst_name='cattle-npww') into database
[1]    78851 terminated  python mandrill/mayhem.py
```

So if we send our program a signal other than SIGINT, like SIGTERM, We see that we don't reach the finally clause…

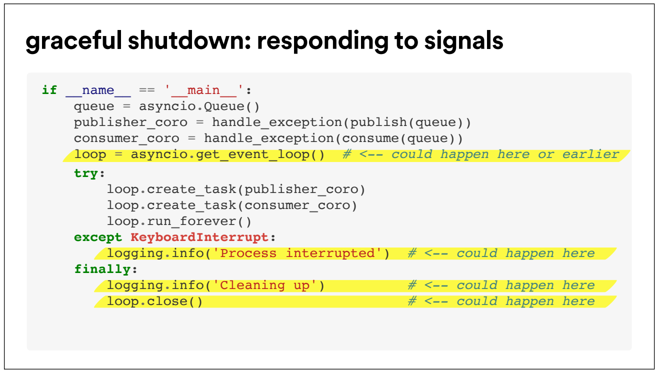


graceful shutdown: responding to signals

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue))
    consumer_coro = handle_exception(consume(queue))
    loop = asyncio.get_event_loop()
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Process interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

where we log that we're cleaning up and close the loop.

It should also be pointed out that – even if we were to only ever expect a KeyboardInterrupt / SIGINT signal –



graceful shutdown: responding to signals

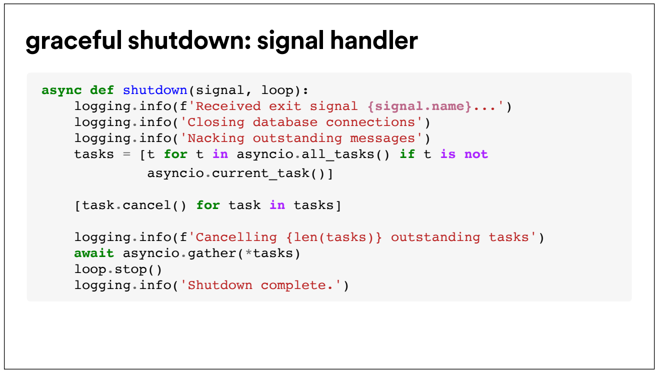
```
if __name__ == '__main__':
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue))
    consumer_coro = handle_exception(consume(queue))
    loop = asyncio.get_event_loop()  # <-- could happen here or earlier
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Process interrupted')  # <-- could happen here
    finally:
        logging.info('Cleaning up')          # <-- could happen here
        loop.close()                         # <-- could happen here
```

it could happen outside the catching of the exception, potentially causing the service to end up in an incomplete or otherwise unknown state.

graceful shutdowns

signal handler

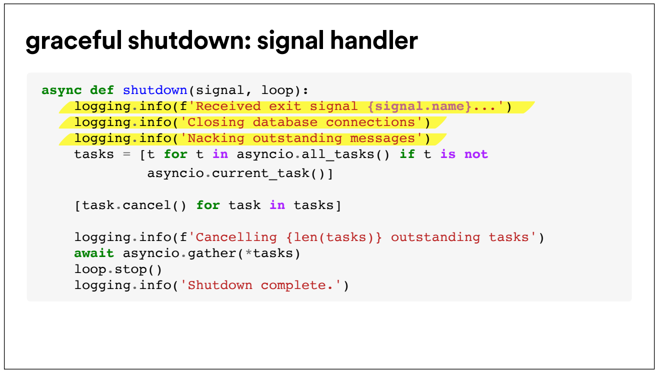
So, instead of catching KeyboardInterrupt, let's attach a signal handler to the loop.



graceful shutdown: signal handler

```
async def shutdown(signal, loop):
    logging.info(f'Received exit signal {signal.name}...')
    logging.info('Closing database connections')
    logging.info('Nacking outstanding messages')
    tasks = [t for t in asyncio.all_tasks() if t is not
             asyncio.current_task()]

    [task.cancel() for task in tasks]

    logging.info(f'Cancelling {len(tasks)} outstanding tasks')
    await asyncio.gather(*tasks)
    loop.stop()
    logging.info('Shutdown complete.')
```

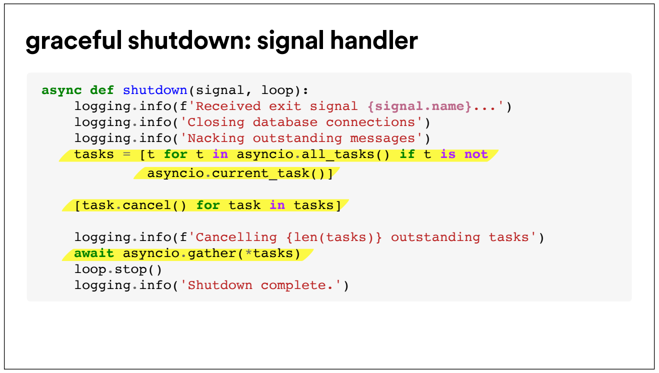
We'll define a shutdown coroutine that will be responsible for doing all of our necessary shutdown tasks.



graceful shutdown: signal handler

```
async def shutdown(signal, loop):
    logging.info(f'Received exit signal {signal.name}...')
    logging.info('Closing database connections')
    logging.info('Nacking outstanding messages')
    tasks = [t for t in asyncio.all_tasks() if t is not
             asyncio.current_task()]

    [task.cancel() for task in tasks]

    logging.info(f'Cancelling {len(tasks)} outstanding tasks')
    await asyncio.gather(*tasks)
    loop.stop()
    logging.info('Shutdown complete.')
```

Here I'm just closing that simulated database connections, returning messages to pub/sub as not acknowledged (so they can be redelivered and not dropped), …

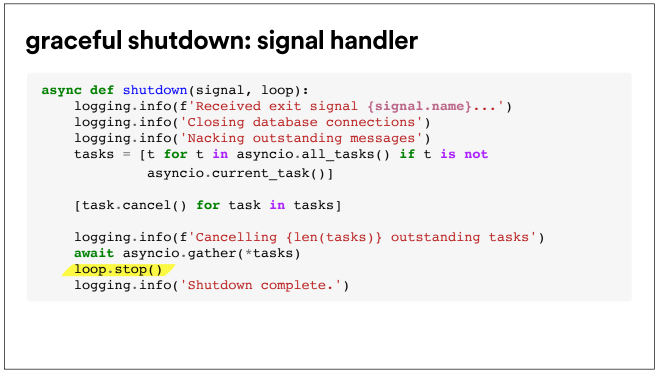


graceful shutdown: signal handler

```
async def shutdown(signal, loop):
    logging.info(f'Received exit signal {signal.name}...')
    logging.info('Closing database connections')
    logging.info('Nacking outstanding messages')
    tasks = [t for t in asyncio.all_tasks() if t is not
             asyncio.current_task()]

    [task.cancel() for task in tasks]

    logging.info(f'Cancelling {len(tasks)} outstanding tasks')
    await asyncio.gather(*tasks)
    loop.stop()
    logging.info('Shutdown complete.')
```

…and then collecting all outstanding tasks - except for the shutdown task, itself - and cancelling them.

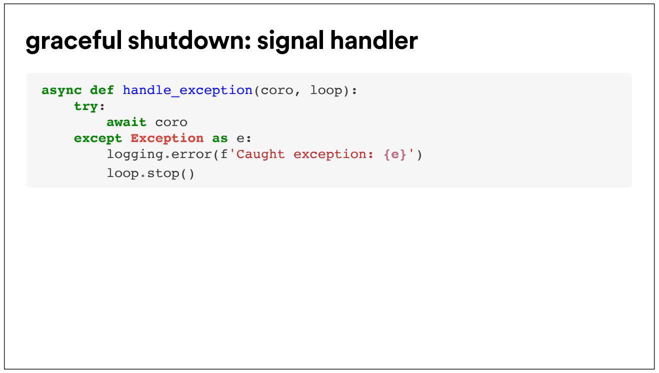
We don't necessarily need to cancel pending tasks; we could just collect and allow them to finish. We may also want to take this opportunity to flush any collected metrics so they're not lost.



graceful shutdown: signal handler

```
async def shutdown(signal, loop):
    logging.info(f'Received exit signal {signal.name}...')
    logging.info('Closing database connections')
    logging.info('Nacking outstanding messages')
    tasks = [t for t in asyncio.all_tasks() if t is not
             asyncio.current_task()]

    [task.cancel() for task in tasks]

    logging.info(f'Cancelling {len(tasks)} outstanding tasks')
    await asyncio.gather(*tasks)
    loop.stop()
    logging.info('Shutdown complete.')
```

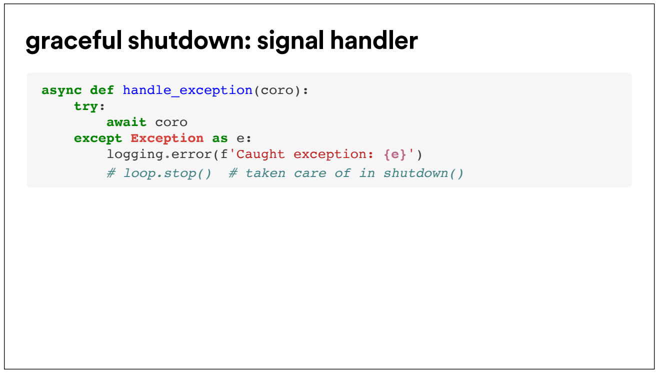
Since we have `loop.stop` here,



graceful shutdown: signal handler

```
async def handle_exception(coro, loop):
    try:
        await coro
    except Exception as e:
        logging.error(f'Caught exception: {e}')
        loop.stop()
```

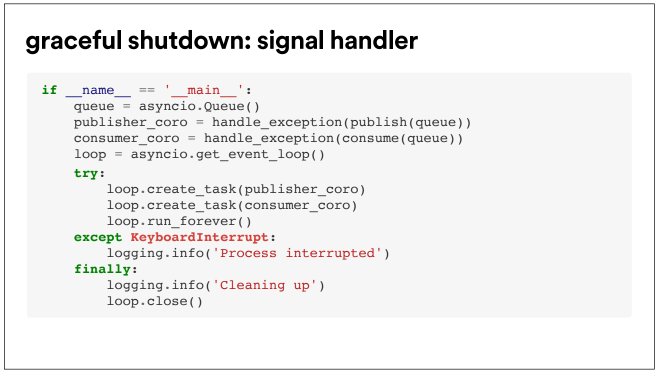
We should remove it…



graceful shutdown: signal handler

```
async def handle_exception(coro):
    try:
        await coro
    except Exception as e:
        logging.error(f'Caught exception: {e}')
        # loop.stop()  # taken care of in shutdown()
```

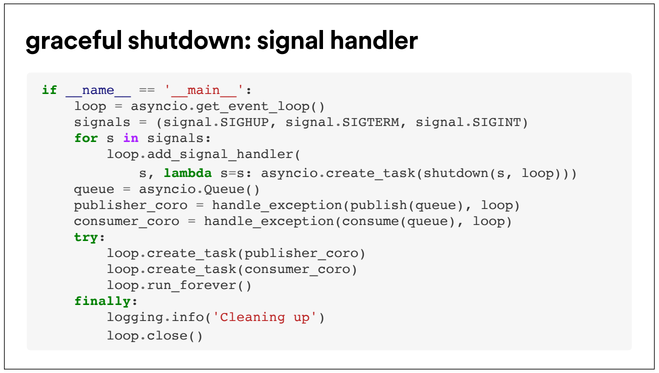
…from our handle_exception coroutine that we defined earlier, and now we no longer need to pass it the `loop` argument.



graceful shutdown: signal handler

```
if __name__ == '__main__':
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue))
    consumer_coro = handle_exception(consume(queue))
    loop = asyncio.get_event_loop()
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    except KeyboardInterrupt:
        logging.info('Process interrupted')
    finally:
        logging.info('Cleaning up')
        loop.close()
```

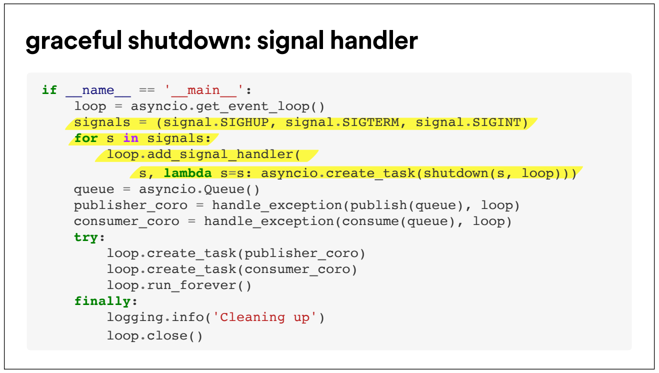
We now need to update our main section.



graceful shutdown: signal handler

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop)))
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue), loop)
    consumer_coro = handle_exception(consume(queue), loop)
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    finally:
        logging.info('Cleaning up')
        loop.close()
```

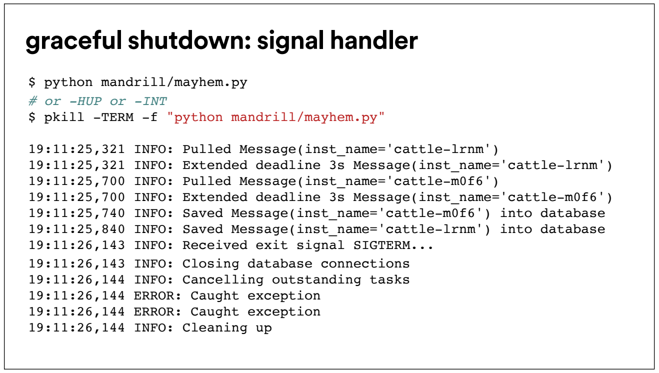
The first thing we do is setup our loop, and add our signal handler….



graceful shutdown: signal handler

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop)))
    queue = asyncio.Queue()
    publisher_coro = handle_exception(publish(queue), loop)
    consumer_coro = handle_exception(consume(queue), loop)
    try:
        loop.create_task(publisher_coro)
        loop.create_task(consumer_coro)
        loop.run_forever()
    finally:
        logging.info('Cleaning up')
        loop.close()
```

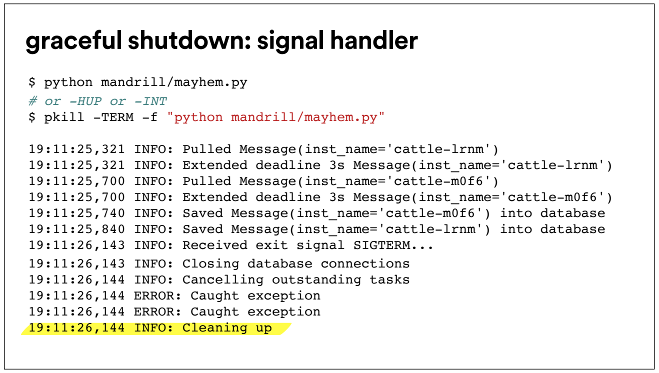
I also removed the KeyboardInterrupt catch since that's now taken care of within the signal handling.



graceful shutdown: signal handler

```
$ python mandrill/mayhem.py
# or -HUP or -INT
$ pkill -TERM -f "python mandrill/mayhem.py"

19:11:25,321 INFO: Pulled Message(inst_name='cattle-lrnm')
19:11:25,321 INFO: Extended deadline 3s Message(inst_name='cattle-lrnm')
19:11:25,700 INFO: Pulled Message(inst_name='cattle-m0f6')
19:11:25,700 INFO: Extended deadline 3s Message(inst_name='cattle-m0f6')
19:11:25,740 INFO: Saved Message(inst_name='cattle-m0f6') into database
19:11:25,840 INFO: Saved Message(inst_name='cattle-lrnm') into database
19:11:26,143 INFO: Received exit signal SIGTERM...
19:11:26,143 INFO: Closing database connections
19:11:26,144 INFO: Cancelling outstanding tasks
19:11:26,144 ERROR: Caught exception
19:11:26,144 ERROR: Caught exception
19:11:26,144 INFO: Cleaning up
```

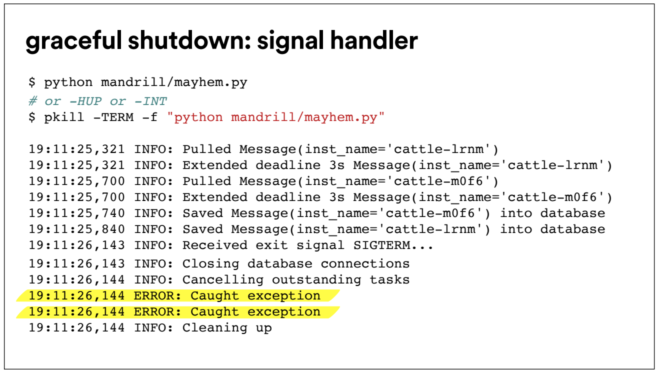
Running this again



graceful shutdown: signal handler

```
$ python mandrill/mayhem.py
# or -HUP or -INT
$ pkill -TERM -f "python mandrill/mayhem.py"

19:11:25,321 INFO: Pulled Message(inst_name='cattle-lrnm')
19:11:25,321 INFO: Extended deadline 3s Message(inst_name='cattle-lrnm')
19:11:25,700 INFO: Pulled Message(inst_name='cattle-m0f6')
19:11:25,700 INFO: Extended deadline 3s Message(inst_name='cattle-m0f6')
19:11:25,740 INFO: Saved Message(inst_name='cattle-m0f6') into database
19:11:25,840 INFO: Saved Message(inst_name='cattle-lrnm') into database
19:11:26,143 INFO: Received exit signal SIGTERM...
19:11:26,143 INFO: Closing database connections
19:11:26,144 INFO: Cancelling outstanding tasks
19:11:26,144 ERROR: Caught exception
19:11:26,144 ERROR: Caught exception
19:11:26,144 INFO: Cleaning up
```

We do get into that “finally” clause.



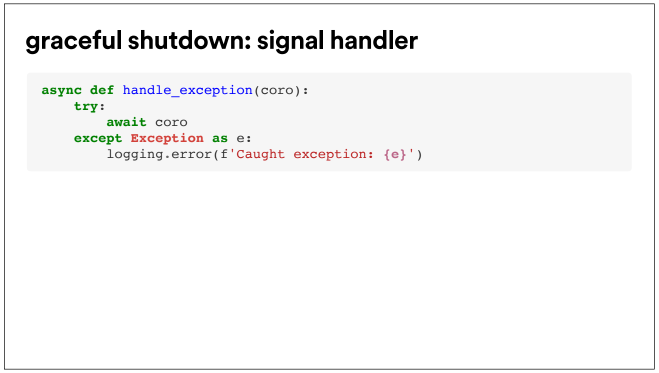
graceful shutdown: signal handler

```
$ python mandrill/mayhem.py
# or -HUP or -INT
$ pkill -TERM -f "python mandrill/mayhem.py"

19:11:25,321 INFO: Pulled Message(inst_name='cattle-lrnm')
19:11:25,321 INFO: Extended deadline 3s Message(inst_name='cattle-lrnm')
19:11:25,700 INFO: Pulled Message(inst_name='cattle-m0f6')
19:11:25,700 INFO: Extended deadline 3s Message(inst_name='cattle-m0f6')
19:11:25,740 INFO: Saved Message(inst_name='cattle-m0f6') into database
19:11:25,840 INFO: Saved Message(inst_name='cattle-lrnm') into database
19:11:26,143 INFO: Received exit signal SIGTERM...
19:11:26,143 INFO: Closing database connections
19:11:26,144 INFO: Cancelling outstanding tasks
19:11:26,144 ERROR: Caught exception
19:11:26,144 ERROR: Caught exception
19:11:26,144 INFO: Cleaning up
```

But it also looks like we hit "Caught exception" twice.

This is because awaiting on cancelled tasks will raise asyncio.CancelledError, which is expected.

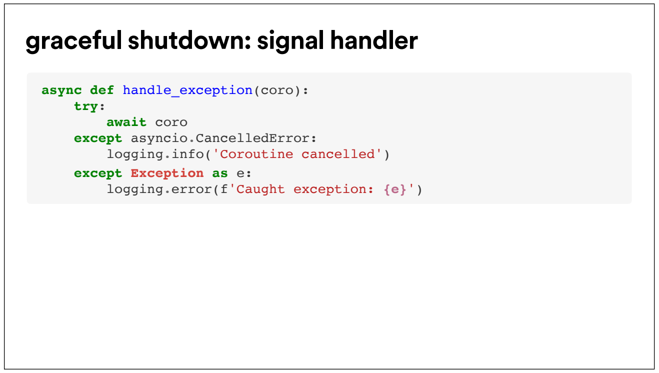
We can handle that better within our handle_exception coroutine as well:



graceful shutdown: signal handler

```
async def handle_exception(coro):
    try:
        await coro
    except Exception as e:
        logging.error(f'Caught exception: {e}')
```

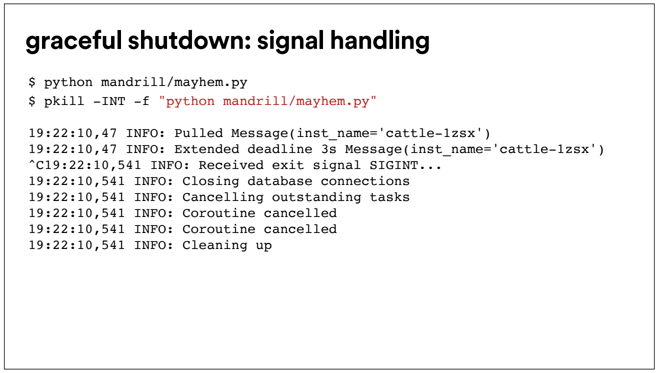
So within this coroutine,



graceful shutdown: signal handler

```
async def handle_exception(coro):
    try:
        await coro
    except asyncio.CancelledError:
        logging.info('Coroutine cancelled')
    except Exception as e:
        logging.error(f'Caught exception: {e}')
```

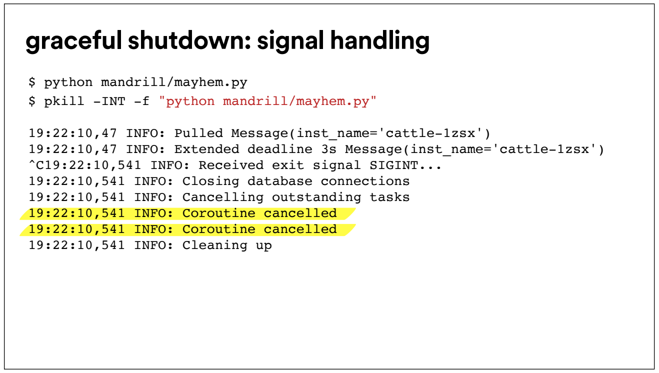
We add a catch for the CancelledError exception, so we can differentiate it from others.



graceful shutdown: signal handling

```
$ python mandrill/mayhem.py
$ pkill -INT -f "python mandrill/mayhem.py"

19:22:10,47 INFO: Pulled Message(inst_name='cattle-1zsx')
19:22:10,47 INFO: Extended deadline 3s Message(inst_name='cattle-1zsx')
^C19:22:10,541 INFO: Received exit signal SIGINT...
19:22:10,541 INFO: Closing database connections
19:22:10,541 INFO: Cancelling outstanding tasks
19:22:10,541 INFO: Coroutine cancelled
19:22:10,541 INFO: Coroutine cancelled
19:22:10,541 INFO: Cleaning up
```

So now we see our coroutines are cancelled



graceful shutdown: signal handling

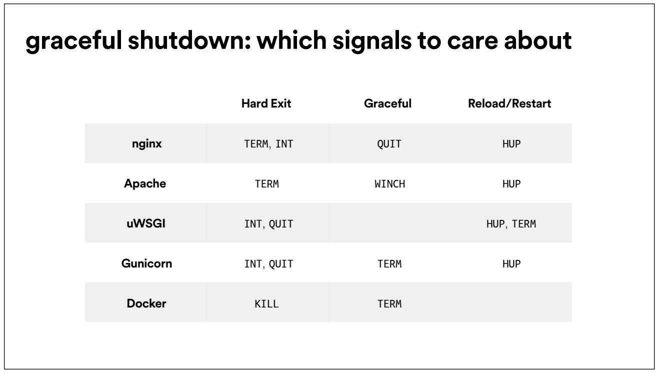
```
$ python mandrill/mayhem.py
$ pkill -INT -f "python mandrill/mayhem.py"

19:22:10,47 INFO: Pulled Message(inst_name='cattle-1zsx')
19:22:10,47 INFO: Extended deadline 3s Message(inst_name='cattle-1zsx')
^C19:22:10,541 INFO: Received exit signal SIGINT...
19:22:10,541 INFO: Closing database connections
19:22:10,541 INFO: Cancelling outstanding tasks
19:22:10,541 INFO: Coroutine cancelled
19:22:10,541 INFO: Coroutine cancelled
19:22:10,541 INFO: Cleaning up
```

and not some random exception.

graceful shutdowns

which signals to care about

You now might be wondering which signals to react to.

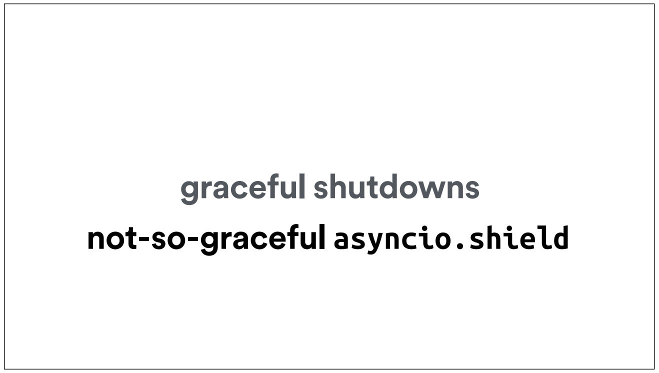


graceful shutdown: which signals to care about

	Hard Exit	Graceful	Reload/Restart
nginx	TERM, INT	QUIT	HUP
Apache	TERM	WINCH	HUP
uWSGI	INT, QUIT		HUP, TERM
Gunicorn	INT, QUIT	TERM	HUP
Docker	KILL	TERM	

And apparently there's no standard.

Basically, you should be aware of how you're running your service, and handle accordingly. It seems like it could get messy with conflicting signals, and when adding docker to the mix.



graceful shutdowns

not-so-graceful `asyncio.shield`

Another misleading API in asyncio is asyncio.shield [1]. The docs say it's a means to shield a future from cancellation. But if you have a coroutine that must not be cancelled during shutdown, asyncio.shield will not help you.

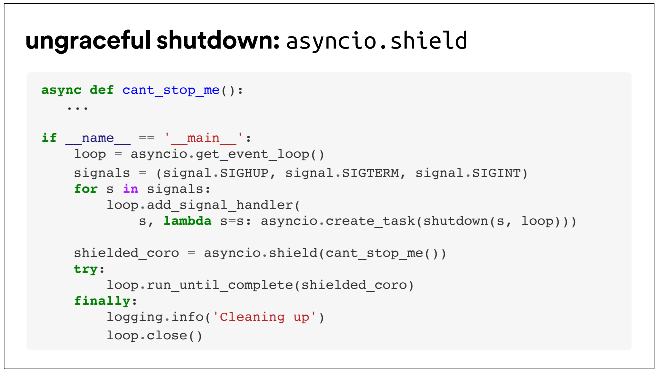
This is because the task that asyncio.shield creates gets included in asyncio.all_tasks, and therefore receives the cancellation signal like the rest of the tasks.

[1] https://docs.python.org/3/library/asyncio-task.html#asyncio.shield

ungraceful shutdown: `asyncio.shield`

```
async def cant_stop_me():
    logging.info('Hold on...')
    await asyncio.sleep(60)
    logging.info('Done!')
```

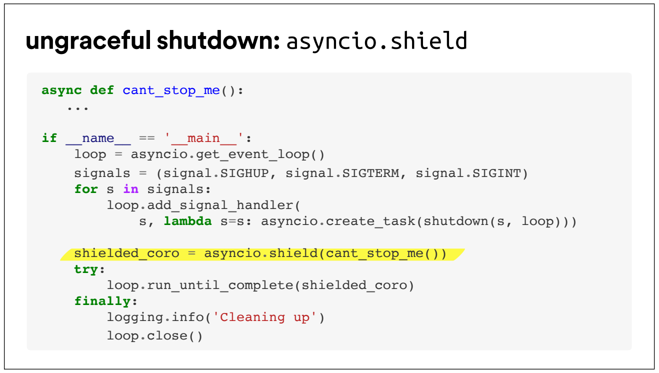
To help illustrate real quick, let's have a simple async function with a long sleep that logs a line saying “Done”.



ungraceful shutdown: `asyncio.shield`

```
async def cant_stop_me():
   ...

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop)))

    shielded_coro = asyncio.shield(cant_stop_me())
    try:
        loop.run_until_complete(shielded_coro)
    finally:
        logging.info('Cleaning up')
        loop.close()
```

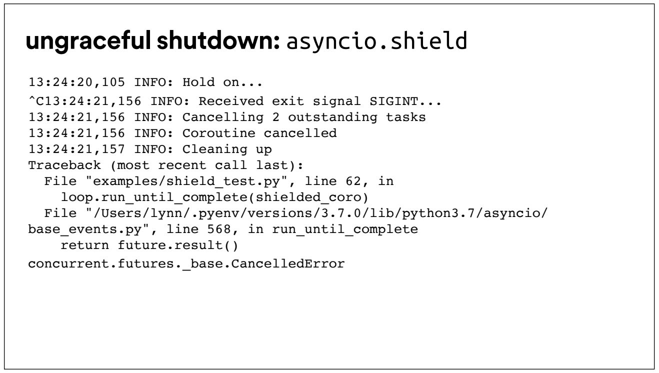
that we want to shield from cancellation



ungraceful shutdown: `asyncio.shield`

```
async def cant_stop_me():
   ...

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(s, loop)))

    shielded_coro = asyncio.shield(cant_stop_me())
    try:
        loop.run_until_complete(shielded_coro)
    finally:
        logging.info('Cleaning up')
        loop.close()
```

With asyncio.shield

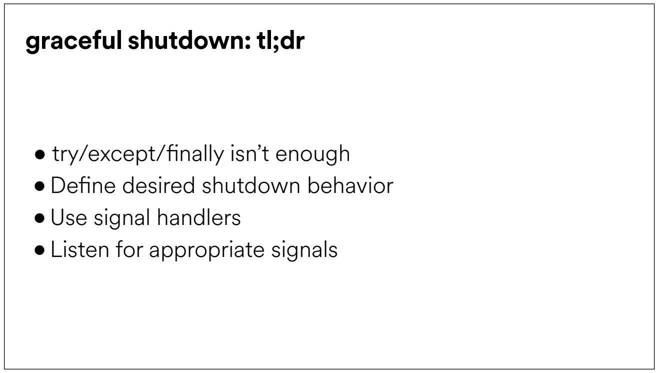


ungraceful shutdown: `asyncio.shield`

```
13:24:20,105 INFO: Hold on...
^C13:24:21,156 INFO: Received exit signal SIGINT...
13:24:21,156 INFO: Cancelling 2 outstanding tasks
13:24:21,156 INFO: Coroutine cancelled
13:24:21,157 INFO: Cleaning up
Traceback (most recent call last):
  File "examples/shield_test.py", line 62, in
    loop.run_until_complete(shielded_coro)
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/
base_events.py", line 568, in run_until_complete
    return future.result()
concurrent.futures._base.CancelledError
```

Running this and cancelling it after a second, we see that we don't get to the "Done" log line, that it's immediately cancelled.

And to be honest, I couldn’t get shield to work under any circumstances…



graceful shutdown: tl;dr

- try/except/finally isn’t enough
- Define desired shutdown behavior
- Use signal handlers
- Listen for appropriate signals

We don't have any nursuries in asyncio core to clean ourselves up; it's up to us to be responsible and close up the connections and files we opened, respond to outstanding requests, basically leave things how we found them.

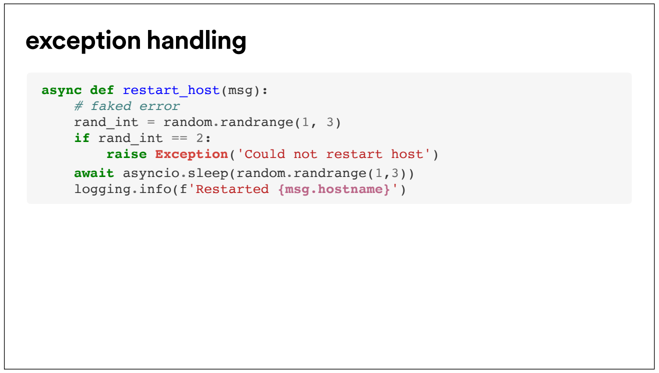
Doing our cleanup in a finally clause isn't enough, though, since a signal could be sent outside of the try/except clause.

So as we construct the loop, we should tell how it should be deconstructed as soon as possible in the program. This ensures that "all our bases are covered", that we're not leaving artifacts anywhere.

And finally, we also need to be aware of when our program should shutdown, which is closely tied to how we run our program. If it's a manually ran script, then SIGINT is fine. But if it's within a daemonized Docker container, then SIGTERM is more appropriate.

exception handling

You may have noticed that, while we're catching exceptions on the top level, we're not paying any mind to exceptions that could be raised from within coroutines like restart_host, save, etc.

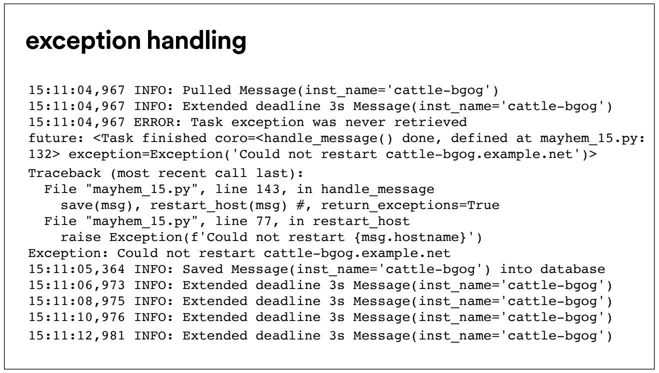


exception handling

```python
async def restart_host(msg):
    # faked error
    rand_int = random.randrange(1, 3)
    if rand_int == 2:
        raise Exception('Could not restart host')
    await asyncio.sleep(random.randrange(1,3))
    logging.info(f'Restarted {msg.hostname}')
```

To show you what I mean, let's fake an error where we can't restart a host.

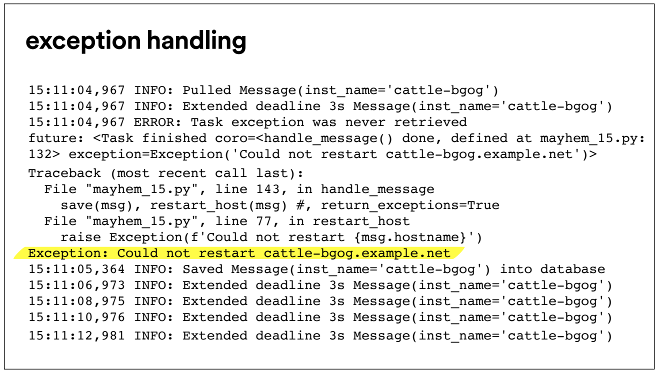
Running it, we see:



exception handling

```
15:11:04,967 INFO: Pulled Message(inst_name='cattle-bgog')
15:11:04,967 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:04,967 ERROR: Task exception was never retrieved
future: <Task finished coro=<handle_message() done, defined at mayhem_15.py:
132> exception=Exception('Could not restart cattle-bgog.example.net')>
Traceback (most recent call last):
  File "mayhem_15.py", line 143, in handle_message
    save(msg), restart_host(msg) #, return_exceptions=True
  File "mayhem_15.py", line 77, in restart_host
    raise Exception(f'Could not restart {msg.hostname}')
Exception: Could not restart cattle-bgog.example.net
15:11:05,364 INFO: Saved Message(inst_name='cattle-bgog') into database
15:11:06,973 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:08,975 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:10,976 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:12,981 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
```

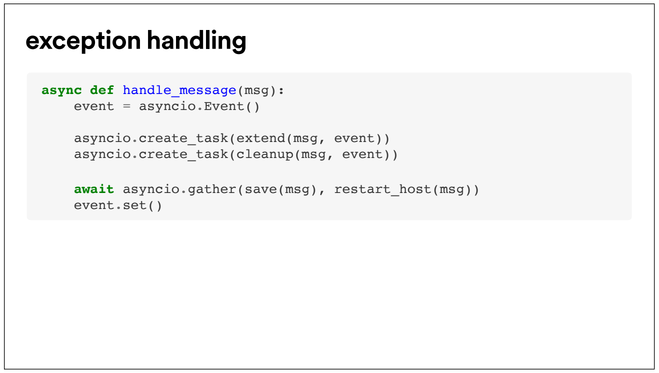
We see that one of the hosts could not be restarted…



exception handling

```
15:11:04,967 INFO: Pulled Message(inst_name='cattle-bgog')
15:11:04,967 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:04,967 ERROR: Task exception was never retrieved
future: <Task finished coro=<handle_message() done, defined at mayhem_15.py:
132> exception=Exception('Could not restart cattle-bgog.example.net')>
Traceback (most recent call last):
  File "mayhem_15.py", line 143, in handle_message
    save(msg), restart_host(msg) #, return_exceptions=True
  File "mayhem_15.py", line 77, in restart_host
    raise Exception(f'Could not restart {msg.hostname}')
Exception: Could not restart cattle-bgog.example.net
15:11:05,364 INFO: Saved Message(inst_name='cattle-bgog') into database
15:11:06,973 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:08,975 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:10,976 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
15:11:12,981 INFO: Extended deadline 3s Message(inst_name='cattle-bgog')
```

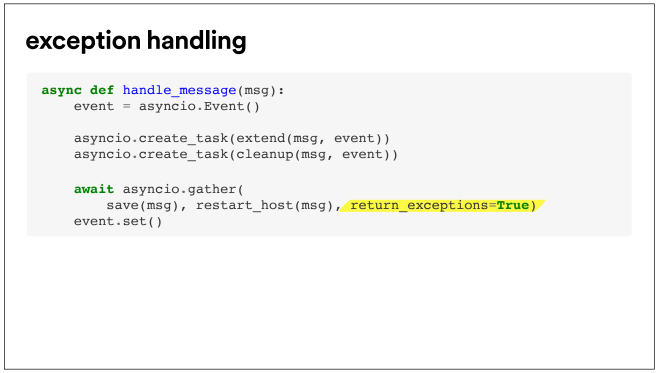
While the service doesn't crash and it did save the message in the database, it did not clean up and ack the message. The extend on the message deadline will also keep spinning, so we've essentially deadlocked on the message.



exception handling

```
async def handle_message(msg):
    event = asyncio.Event()

    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))

    await asyncio.gather(save(msg), restart_host(msg))
    event.set()
```

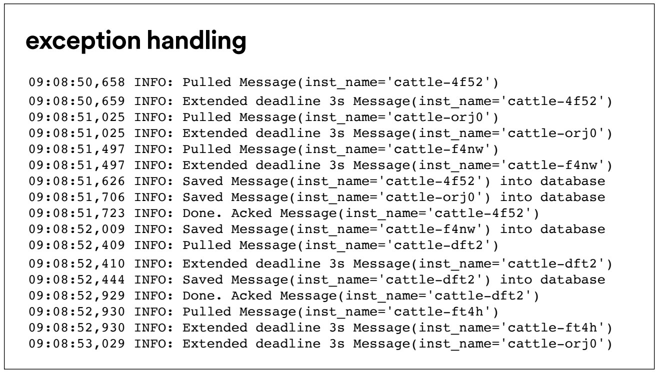
The simple thing to do is - within handle_message -



exception handling

```
async def handle_message(msg):
    event = asyncio.Event()

    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))

    await asyncio.gather(
        save(msg), restart_host(msg), return_exceptions=True)
    event.set()
```

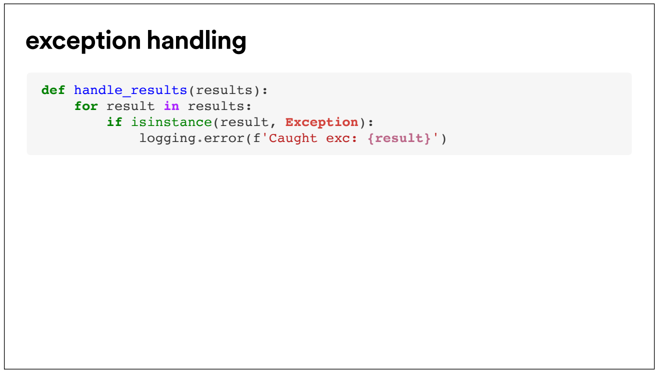
add return_exceptions=True to asyncio.gather, so rather than completely dropping an exception, it's returned along with the successful results:



exception handling

```
09:08:50,658 INFO: Pulled Message(inst_name='cattle-4f52')
09:08:50,659 INFO: Extended deadline 3s Message(inst_name='cattle-4f52')
09:08:51,025 INFO: Pulled Message(inst_name='cattle-orj0')
09:08:51,025 INFO: Extended deadline 3s Message(inst_name='cattle-orj0')
09:08:51,497 INFO: Pulled Message(inst_name='cattle-f4nw')
09:08:51,497 INFO: Extended deadline 3s Message(inst_name='cattle-f4nw')
09:08:51,626 INFO: Saved Message(inst_name='cattle-4f52') into database
09:08:51,706 INFO: Saved Message(inst_name='cattle-orj0') into database
09:08:51,723 INFO: Done. Acked Message(inst_name='cattle-4f52')
09:08:52,009 INFO: Saved Message(inst_name='cattle-f4nw') into database
09:08:52,409 INFO: Pulled Message(inst_name='cattle-dft2')
09:08:52,410 INFO: Extended deadline 3s Message(inst_name='cattle-dft2')
09:08:52,444 INFO: Saved Message(inst_name='cattle-dft2') into database
09:08:52,929 INFO: Done. Acked Message(inst_name='cattle-dft2')
09:08:52,930 INFO: Pulled Message(inst_name='cattle-ft4h')
09:08:52,930 INFO: Extended deadline 3s Message(inst_name='cattle-ft4h')
09:08:53,029 INFO: Extended deadline 3s Message(inst_name='cattle-orj0')
```

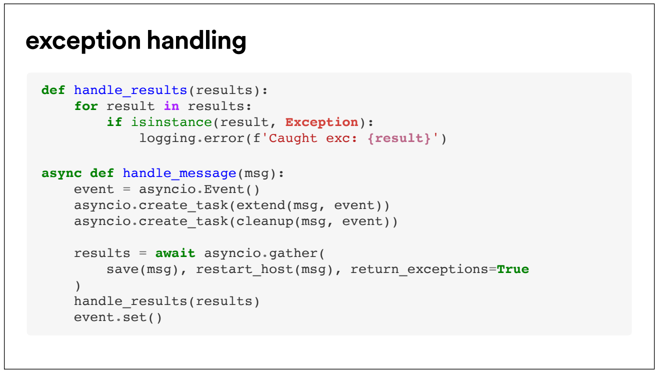
So while we don’t see any tracebacks nor are we purpetually extending messages, we don’t yet see where or when we have encountered errors. It’d be nice to see which tasks finish with an error



exception handling

```python
def handle_results(results):
    for result in results:
        if isinstance(result, Exception):
            logging.error(f'Caught exc: {result}')
```

We could add a callback via add_done_callback to the asyncio.gather future, but as I said, I'm allergic. We can just process the results of the gathering of tasks afterwards….

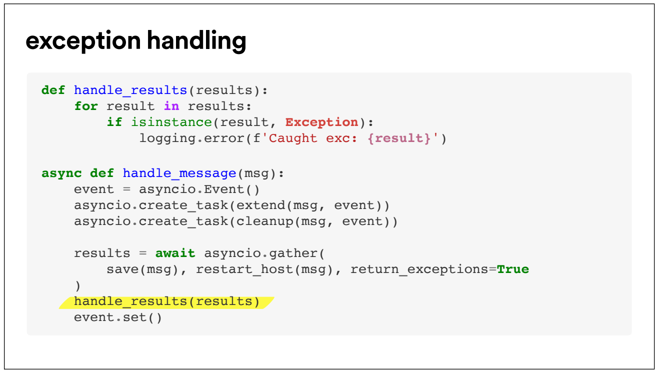


exception handling

```
def handle_results(results):
    for result in results:
        if isinstance(result, Exception):
            logging.error(f'Caught exc: {result}')

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))

    results = await asyncio.gather(
        save(msg), restart_host(msg), return_exceptions=True
    )
    handle_results(results)
    event.set()
```

…which for me, seems easier to understand what's going on.

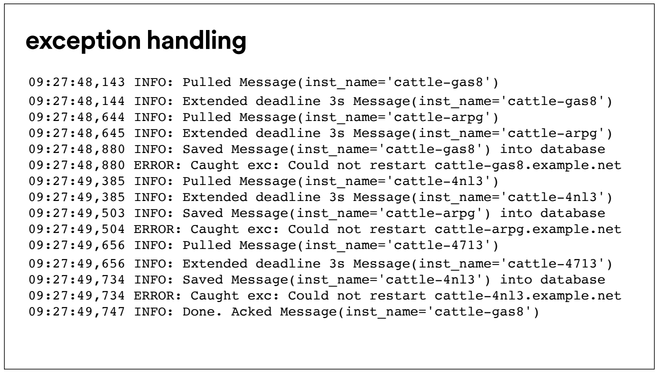
So here after the asyncio.gather returns,…



exception handling

```
def handle_results(results):
    for result in results:
        if isinstance(result, Exception):
            logging.error(f'Caught exc: {result}')

async def handle_message(msg):
    event = asyncio.Event()
    asyncio.create_task(extend(msg, event))
    asyncio.create_task(cleanup(msg, event))

    results = await asyncio.gather(
        save(msg), restart_host(msg), return_exceptions=True
    )
    handle_results(results)
    event.set()
```

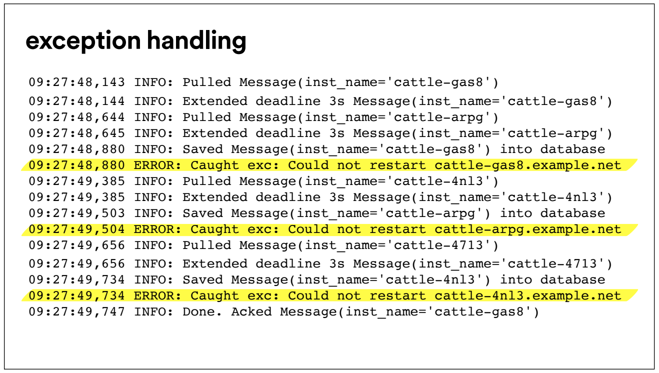
we can separate out the successful results from the errors.



exception handling

```
09:27:48,143 INFO: Pulled Message(inst_name='cattle-gas8')
09:27:48,144 INFO: Extended deadline 3s Message(inst_name='cattle-gas8')
09:27:48,644 INFO: Pulled Message(inst_name='cattle-arpg')
09:27:48,645 INFO: Extended deadline 3s Message(inst_name='cattle-arpg')
09:27:48,880 INFO: Saved Message(inst_name='cattle-gas8') into database
09:27:48,880 ERROR: Caught exc: Could not restart cattle-gas8.example.net
09:27:49,385 INFO: Pulled Message(inst_name='cattle-4nl3')
09:27:49,385 INFO: Extended deadline 3s Message(inst_name='cattle-4nl3')
09:27:49,503 INFO: Saved Message(inst_name='cattle-arpg') into database
09:27:49,504 ERROR: Caught exc: Could not restart cattle-arpg.example.net
09:27:49,656 INFO: Pulled Message(inst_name='cattle-4713')
09:27:49,656 INFO: Extended deadline 3s Message(inst_name='cattle-4713')
09:27:49,734 INFO: Saved Message(inst_name='cattle-4nl3') into database
09:27:49,734 ERROR: Caught exc: Could not restart cattle-4nl3.example.net
09:27:49,747 INFO: Done. Acked Message(inst_name='cattle-gas8')
```

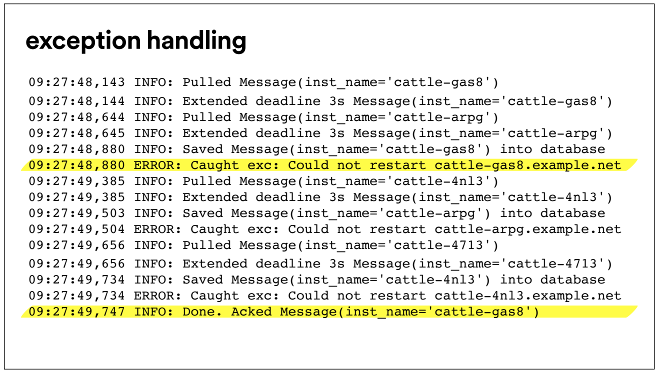
Now running with this,



exception handling

```
09:27:48,143 INFO: Pulled Message(inst_name='cattle-gas8')
09:27:48,144 INFO: Extended deadline 3s Message(inst_name='cattle-gas8')
09:27:48,644 INFO: Pulled Message(inst_name='cattle-arpg')
09:27:48,645 INFO: Extended deadline 3s Message(inst_name='cattle-arpg')
09:27:48,880 INFO: Saved Message(inst_name='cattle-gas8') into database
09:27:48,880 ERROR: Caught exc: Could not restart cattle-gas8.example.net
09:27:49,385 INFO: Pulled Message(inst_name='cattle-4nl3')
09:27:49,385 INFO: Extended deadline 3s Message(inst_name='cattle-4nl3')
09:27:49,503 INFO: Saved Message(inst_name='cattle-arpg') into database
09:27:49,504 ERROR: Caught exc: Could not restart cattle-arpg.example.net
09:27:49,656 INFO: Pulled Message(inst_name='cattle-4713')
09:27:49,656 INFO: Extended deadline 3s Message(inst_name='cattle-4713')
09:27:49,734 INFO: Saved Message(inst_name='cattle-4nl3') into database
09:27:49,734 ERROR: Caught exc: Could not restart cattle-4nl3.example.net
09:27:49,747 INFO: Done. Acked Message(inst_name='cattle-gas8')
```

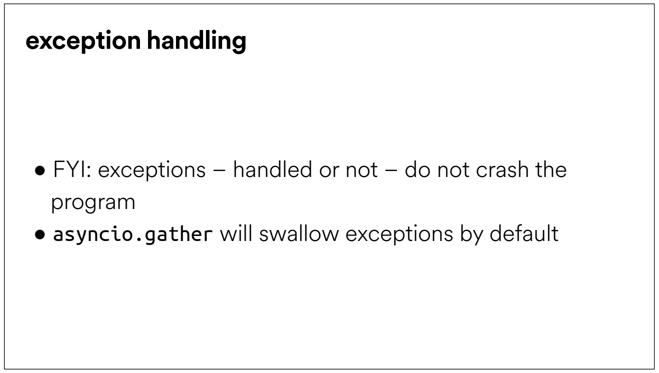
We indeed see errors logged.



exception handling

```
09:27:48,143 INFO: Pulled Message(inst_name='cattle-gas8')
09:27:48,144 INFO: Extended deadline 3s Message(inst_name='cattle-gas8')
09:27:48,644 INFO: Pulled Message(inst_name='cattle-arpg')
09:27:48,645 INFO: Extended deadline 3s Message(inst_name='cattle-arpg')
09:27:48,880 INFO: Saved Message(inst_name='cattle-gas8') into database
09:27:48,880 ERROR: Caught exc: Could not restart cattle-gas8.example.net
09:27:49,385 INFO: Pulled Message(inst_name='cattle-4nl3')
09:27:49,385 INFO: Extended deadline 3s Message(inst_name='cattle-4nl3')
09:27:49,503 INFO: Saved Message(inst_name='cattle-arpg') into database
09:27:49,504 ERROR: Caught exc: Could not restart cattle-arpg.example.net
09:27:49,656 INFO: Pulled Message(inst_name='cattle-4713')
09:27:49,656 INFO: Extended deadline 3s Message(inst_name='cattle-4713')
09:27:49,734 INFO: Saved Message(inst_name='cattle-4nl3') into database
09:27:49,734 ERROR: Caught exc: Could not restart cattle-4nl3.example.net
09:27:49,747 INFO: Done. Acked Message(inst_name='cattle-gas8')
```

But again, even though a message's task errored out, it doesn't block from being cleaned up.



exception handling

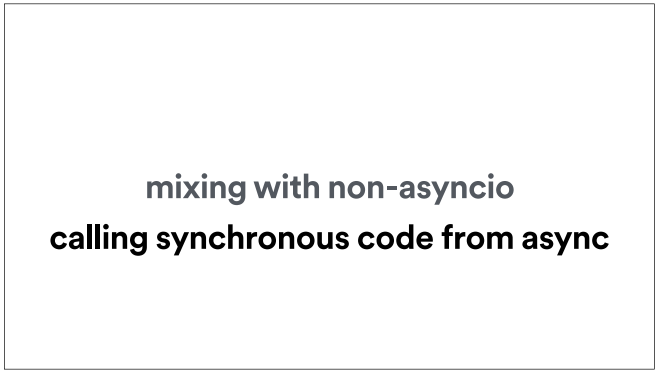
- FYI: exceptions – handled or not – do not crash the program
- `asyncio.gather` will swallow exceptions by default

Exceptions will not crash the system - unlike non-asyncio programs. and they might go unnoticed. So we need to account for that.

I personally like using asyncio.gather because the order of the returned results are deterministic, but it's easy to get tripped up with it. By default, it will swallow exceptions but happily continue working on the other tasks that were given. If an exception is never returned, weird behavior can happen, like spinning around an event.

mixing with non-asyncio

I'm sure that as folks have started to use asyncio, they've realized that async/await starts infecting everything around the codebase; everything needs to be async. This isn't necessarily a bad thing; it just forces a shift in perspective.

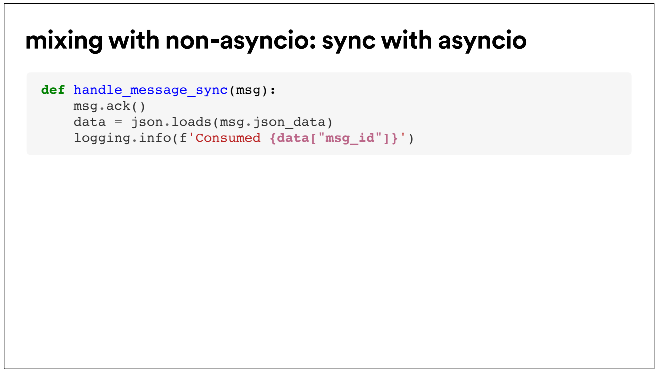


mixing with non-asyncio

calling synchronous code from async

But sometimes, third-party code throws a wrench at you...

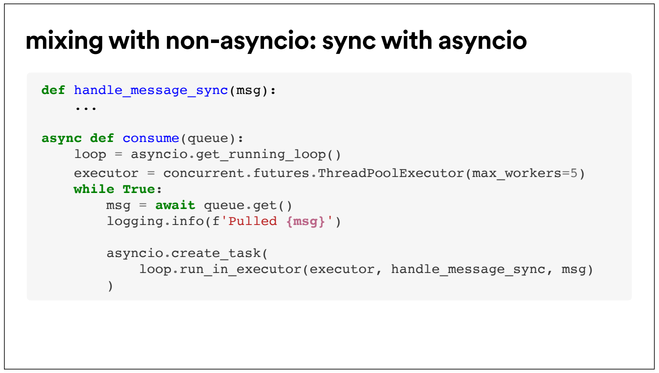
If you're lucky, you'll be faced with a third-party library that is multi-threaded and blocking. For example, Google Pub/Sub's Python library makes use of gPRC under the hood via threading, but is also blocks when we're opening up a subscription. The library also requires a non-async callback (:grimace:) for when a message is received.



mixing with non-asyncio: sync with asyncio

```
def handle_message_sync(msg):
    msg.ack()
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
```

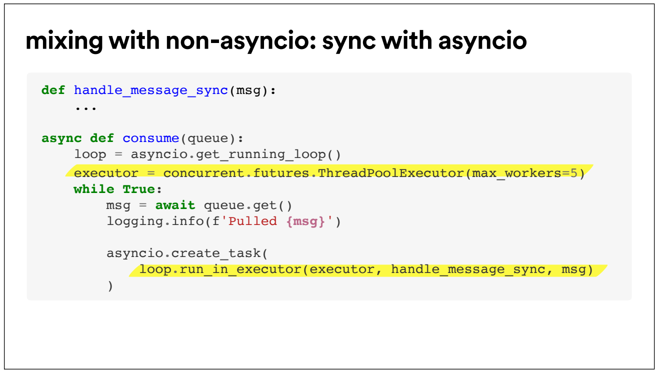
Let’s say we have a non-async function that we need to call from our asynchronous code. It
So for our code to work with this, we need to rework our async consumer.



mixing with non-asyncio: sync with asyncio

```
def handle_message_sync(msg):
    ...

async def consume(queue):
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=5)
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(
            loop.run_in_executor(executor, handle_message_sync, msg)
        )
```

Not much needed actually.



mixing with non-asyncio: sync with asyncio

```
def handle_message_sync(msg):
    ...

async def consume(queue):
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=5)
    while True:
        msg = await queue.get()
        logging.info(f'Pulled {msg}')

        asyncio.create_task(
            loop.run_in_executor(executor, handle_message_sync, msg)
        )
```

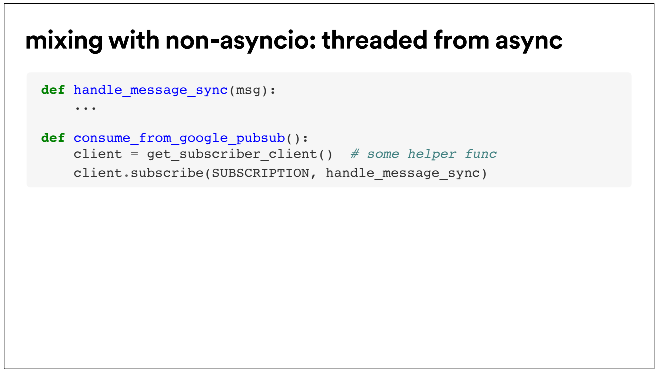
Here I'm still making use of our own asynchronous consumer coroutine to call the non-async consumer by running the synchronous code in a threadpool executor.

mixing with non-asyncio

threaded code from async

But sometimes, third-party code throws a wrench at you...

If you're lucky, you'll be faced with a third-party library that is multi-threaded and blocking. For example, Google Pub/Sub's Python library makes use of gPRC under the hood via threading, but is also blocks when we're opening up a subscription. The library also requires a non-async callback (:grimace:) for when a message is received.

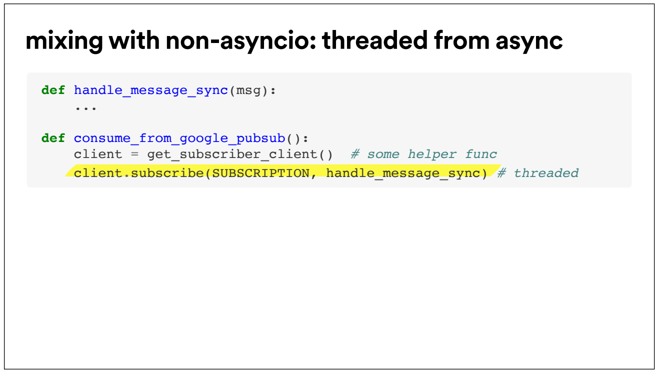


mixing with non-asyncio: threaded from async

```
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    client = get_subscriber_client()  # some helper func
    client.subscribe(SUBSCRIPTION, handle_message_sync)
```

In typical Google fashion, they'll stuff some uber-cool technology in a difficult to work-with library. This future that's returned, it will make use of gRPC for bidirectional communication and remove our need to periodically pull for messages as well as manage message deadlines.

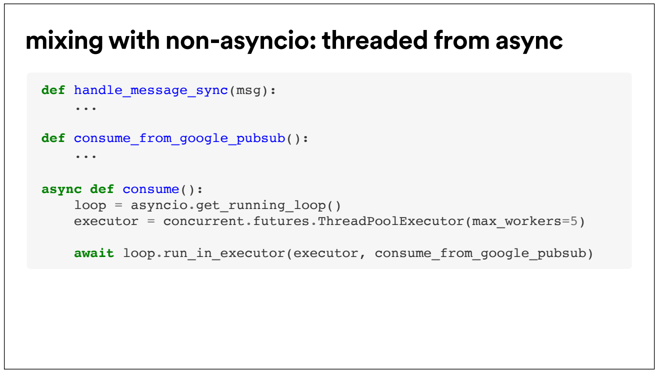
To illustrate, here's how we can use loop.run_in_executor for this blocking code



mixing with non-asyncio: threaded from async

```
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    client = get_subscriber_client()  # some helper func
    client.subscribe(SUBSCRIPTION, handle_message_sync) # threaded
```

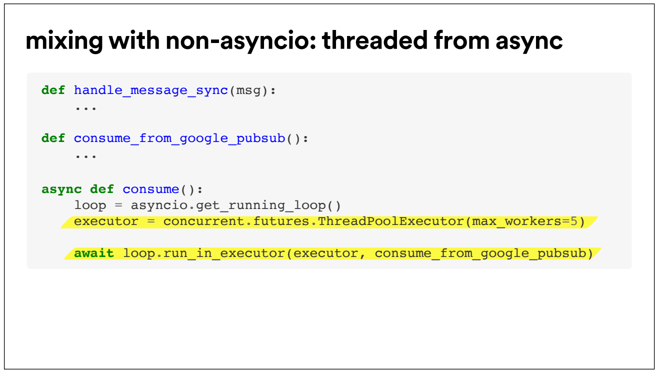
Here, the client.subscribe from google’s python library is multithreaded



mixing with non-asyncio: threaded from async

```
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    ...

async def consume():
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=5)

    await loop.run_in_executor(executor, consume_from_google_pubsub)
```

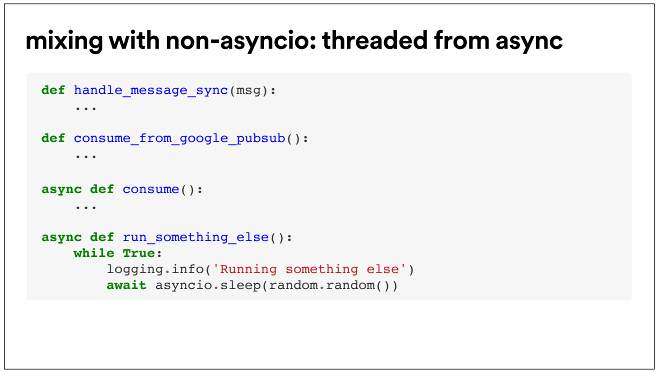
And now some small tweaks to our consume function - it no longer needs to take a `queue` for an argument, since the `consume_from_google_pubsub` is pulling from an external queue.



mixing with non-asyncio: threaded from async

```
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    ...

async def consume():
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=5)

    await loop.run_in_executor(executor, consume_from_google_pubsub)
```

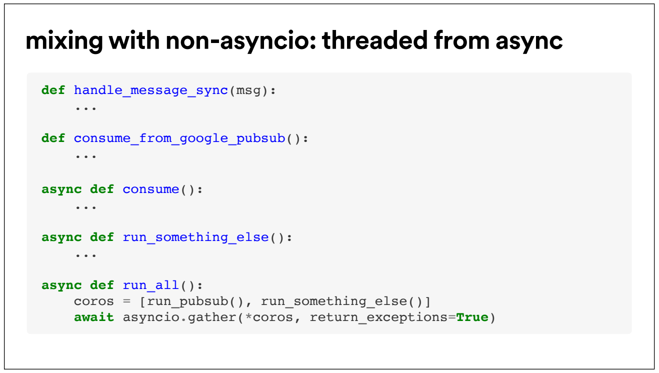
The `loop.run_in_executor` returns a future object, which we can just await on



mixing with non-asyncio: threaded from async

```python
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    ...

async def consume():
    ...

async def run_something_else():
    while True:
        logging.info('Running something else')
        await asyncio.sleep(random.random())
```

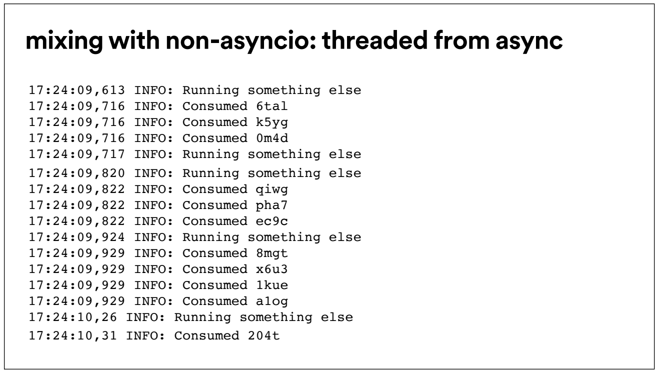
I'd like to also prove that this is now non-blocking, so let's add a dummy coroutine function to be ran alongside our consume coroutine.



mixing with non-asyncio: threaded from async

```python
def handle_message_sync(msg):
    ...

def consume_from_google_pubsub():
    ...

async def consume():
    ...

async def run_something_else():
    ...

async def run_all():
    coros = [run_pubsub(), run_something_else()]
    await asyncio.gather(*coros, return_exceptions=True)
```

So then sort of a main run coroutine that's called from our main section to run forever.

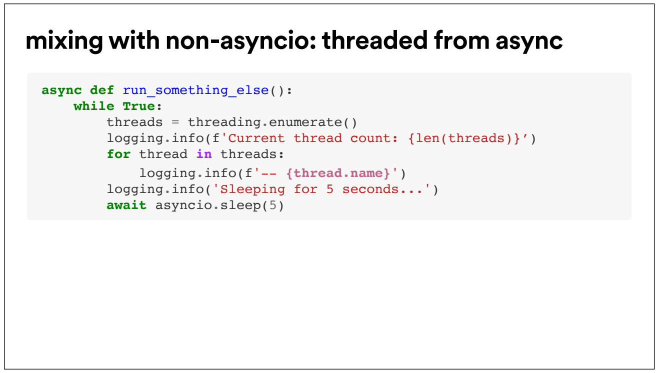


mixing with non-asyncio: threaded from async

```
17:24:09,613 INFO: Running something else
17:24:09,716 INFO: Consumed 6tal
17:24:09,716 INFO: Consumed k5yg
17:24:09,716 INFO: Consumed 0m4d
17:24:09,717 INFO: Running something else
17:24:09,820 INFO: Running something else
17:24:09,822 INFO: Consumed qiwg
17:24:09,822 INFO: Consumed pha7
17:24:09,822 INFO: Consumed ec9c
17:24:09,924 INFO: Running something else
17:24:09,929 INFO: Consumed 8mgt
17:24:09,929 INFO: Consumed x6u3
17:24:09,929 INFO: Consumed 1kue
17:24:09,929 INFO: Consumed a1og
17:24:10,26 INFO: Running something else
17:24:10,31 INFO: Consumed 204t
```

Now running it will show things being consumed, and something else being ran

<TODO: skip below?>
As I mentioned, that google client is threaded to handle the regular polling of the messages, and message deadline extensions, which is great

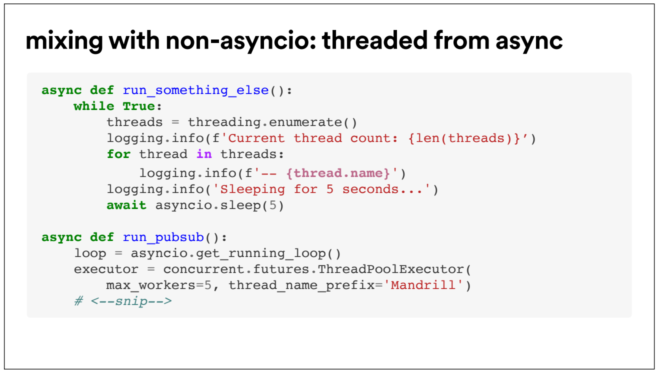


mixing with non-asyncio: threaded from async

```python
async def run_something_else():
    while True:
        threads = threading.enumerate()
        logging.info(f'Current thread count: {len(threads)}’)
        for thread in threads:
            logging.info(f'-- {thread.name}')
        logging.info('Sleeping for 5 seconds...')
        await asyncio.sleep(5)
```

<TODO: skip?>

But, it does introduce a bunch of threads. To see what’s going on underneath the hood, we can reuse the `run_something_else` coroutine function to get some periodic stats on our threads.

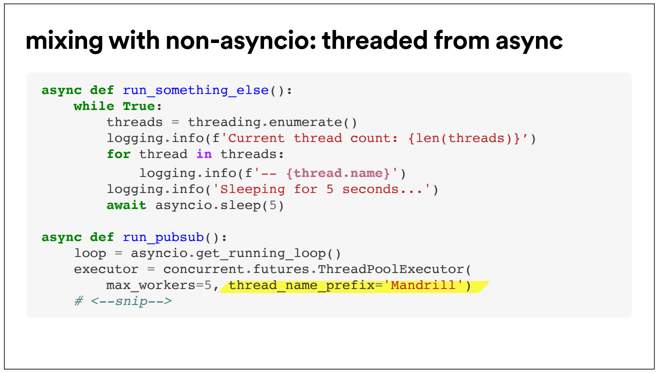


mixing with non-asyncio: threaded from async

```
async def run_something_else():
    while True:
        threads = threading.enumerate()
        logging.info(f'Current thread count: {len(threads)}’)
        for thread in threads:
            logging.info(f'-- {thread.name}')
        logging.info('Sleeping for 5 seconds...')
        await asyncio.sleep(5)

async def run_pubsub():
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(
        max_workers=5, thread_name_prefix='Mandrill')
    # <--snip-->
```

<TODO: skip?>

I'm also going to use a prefix our own threads

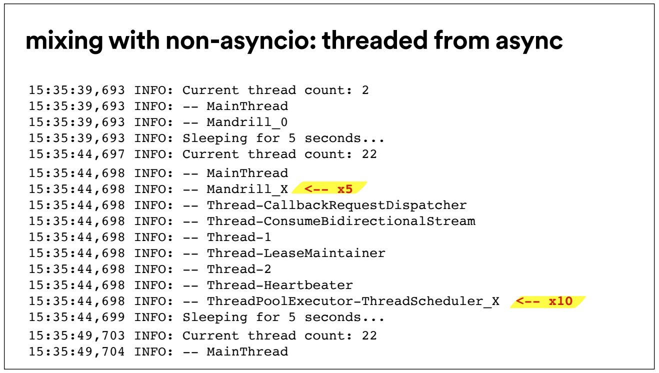


mixing with non-asyncio: threaded from async

```
async def run_something_else():
    while True:
        threads = threading.enumerate()
        logging.info(f'Current thread count: {len(threads)}’)
        for thread in threads:
            logging.info(f'-- {thread.name}')
        logging.info('Sleeping for 5 seconds...')
        await asyncio.sleep(5)

async def run_pubsub():
    loop = asyncio.get_running_loop()
    executor = concurrent.futures.ThreadPoolExecutor(
        max_workers=5, thread_name_prefix='Mandrill')
    # <--snip-->
```

<TODO: skip?>

from our threadpool executor so I can easily tell which threads I created versus others.

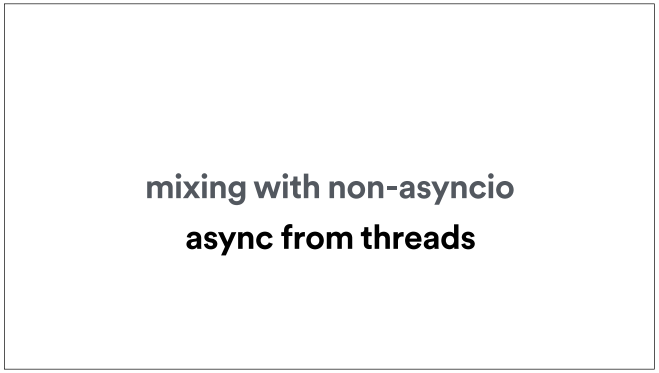


mixing with non-asyncio: threaded from async

```
15:35:39,693 INFO: Current thread count: 2
15:35:39,693 INFO: -- MainThread
15:35:39,693 INFO: -- Mandrill_0
15:35:39,693 INFO: Sleeping for 5 seconds...
15:35:44,697 INFO: Current thread count: 22
15:35:44,698 INFO: -- MainThread
15:35:44,698 INFO: -- Mandrill_X  <-- x5
15:35:44,698 INFO: -- Thread-CallbackRequestDispatcher
15:35:44,698 INFO: -- Thread-ConsumeBidirectionalStream
15:35:44,698 INFO: -- Thread-1
15:35:44,698 INFO: -- Thread-LeaseMaintainer
15:35:44,698 INFO: -- Thread-2
15:35:44,698 INFO: -- Thread-Heartbeater
15:35:44,698 INFO: -- ThreadPoolExecutor-ThreadScheduler_X  <-- x10
15:35:44,699 INFO: Sleeping for 5 seconds...
15:35:49,703 INFO: Current thread count: 22
15:35:49,704 INFO: -- MainThread
```

We see we have the MainThread which is the asyncio event loop. There's also five Mandrill_-prefixed threads that were created by our threadpool executor. There's five because we limited the number of workers when creating the executor. It looks as if the subscription client has its own threadpool executor named ThreadPoolExecutor-ThreadScheduler; Thread-MonitorBatchPublisher is from the publisher; and some gRPC/bidirectional streaming going on with the rest of the threads (heart beater, lease maintainer, etc).

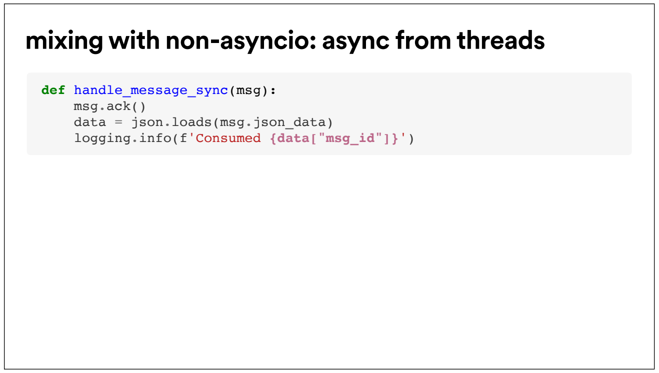
All in all, though, the approach to threaded code isn't any different than the non-async code.



mixing with non-asyncio

async from threads

Until you release you need to call asynchronous code from a non-async function that's within a thread.



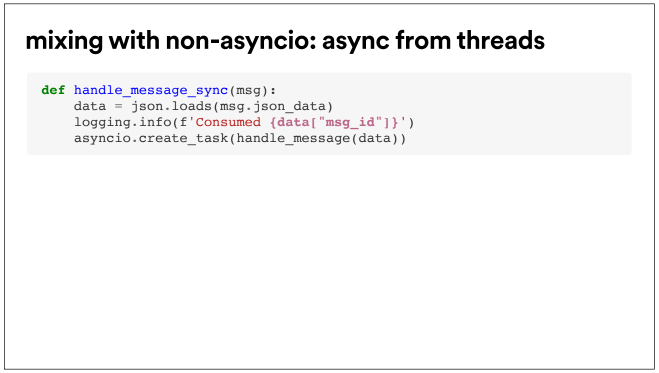
mixing with non-asyncio: async from threads

```python
def handle_message_sync(msg):
    msg.ack()
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
```

We currently just ack the message once we receive it, but that’s not what we want.

We need to restart the required host and save the message in our database. We have to somehow call asynchronous code from a non-async function, from a separate thread.

This is pretty embarrassing, bare with me.

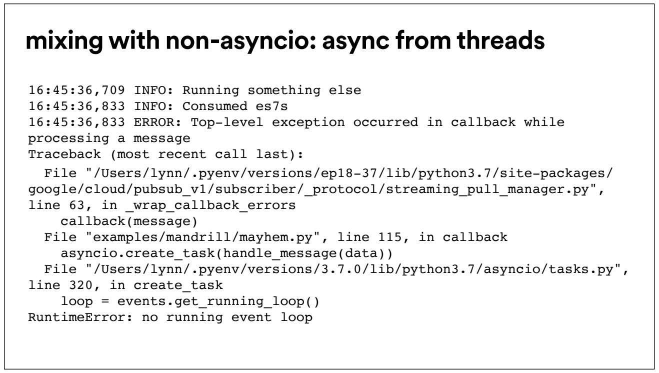


mixing with non-asyncio: async from threads

```python
def handle_message_sync(msg):
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
    asyncio.create_task(handle_message(data))
```

Let's first attempt to use the asyncio API that we're familar with, and update our synchronous callback function with creating a task via asyncio.create_task from the handle_message coroutine we defined earlier.

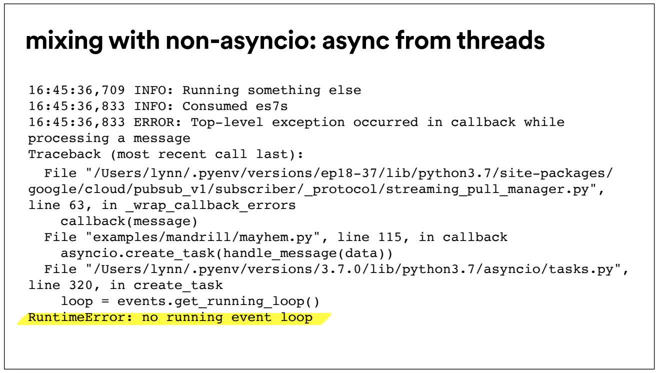
Note that `msg.ack` got removed since it’s taken care of in our handle_message coroutine



mixing with non-asyncio: async from threads

```
16:45:36,709 INFO: Running something else
16:45:36,833 INFO: Consumed es7s
16:45:36,833 ERROR: Top-level exception occurred in callback while
processing a message
Traceback (most recent call last):
  File "/Users/lynn/.pyenv/versions/ep18-37/lib/python3.7/site-packages/
google/cloud/pubsub_v1/subscriber/_protocol/streaming_pull_manager.py",
line 63, in _wrap_callback_errors
    callback(message)
  File "examples/mandrill/mayhem.py", line 115, in callback
    asyncio.create_task(handle_message(data))
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/tasks.py",
line 320, in create_task
    loop = events.get_running_loop()
RuntimeError: no running event loop
```

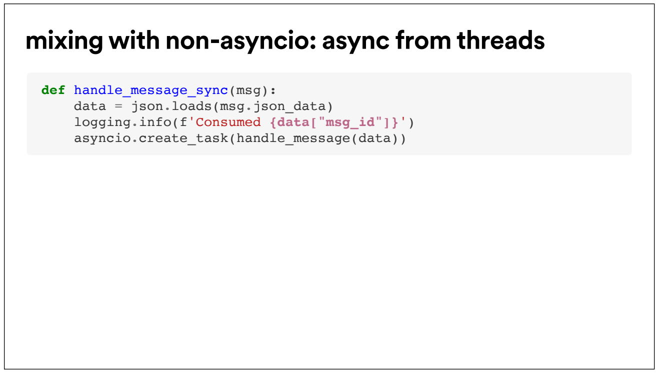
Ugh; sure, yes this of course makes sense.



mixing with non-asyncio: async from threads

```
16:45:36,709 INFO: Running something else
16:45:36,833 INFO: Consumed es7s
16:45:36,833 ERROR: Top-level exception occurred in callback while
processing a message
Traceback (most recent call last):
  File "/Users/lynn/.pyenv/versions/ep18-37/lib/python3.7/site-packages/
google/cloud/pubsub_v1/subscriber/_protocol/streaming_pull_manager.py",
line 63, in _wrap_callback_errors
    callback(message)
  File "examples/mandrill/mayhem.py", line 115, in callback
    asyncio.create_task(handle_message(data))
  File "/Users/lynn/.pyenv/versions/3.7.0/lib/python3.7/asyncio/tasks.py",
line 320, in create_task
    loop = events.get_running_loop()
RuntimeError: no running event loop
```

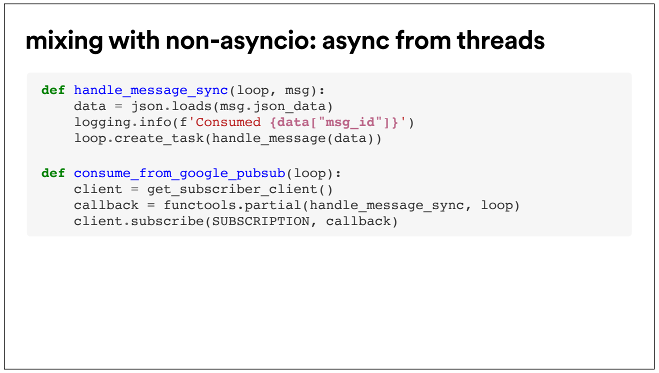
At this point, we're in another thread and there is no loop running for that thread, only in the main thread.



mixing with non-asyncio: async from threads

```
def handle_message_sync(msg):
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
    asyncio.create_task(handle_message(data))
```

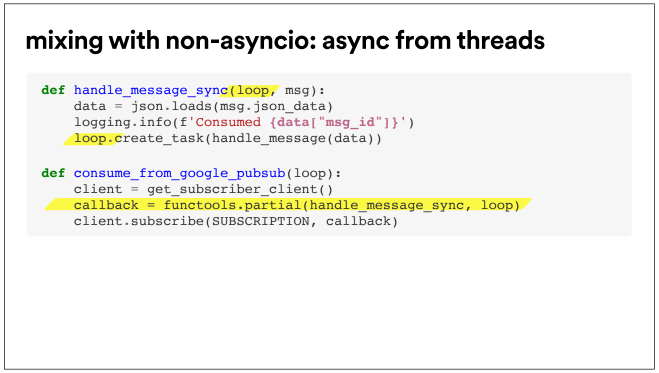
So if we take what we have right now,…



mixing with non-asyncio: async from threads

```
def handle_message_sync(loop, msg):
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(handle_message(data))

def consume_from_google_pubsub(loop):
    client = get_subscriber_client()
    callback = functools.partial(handle_message_sync, loop)
    client.subscribe(SUBSCRIPTION, callback)
```

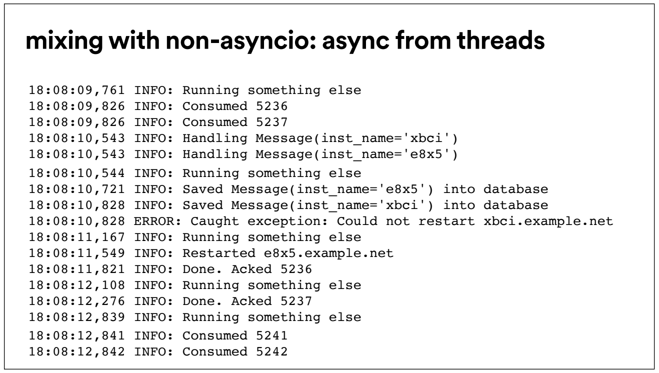
And update our function to…



mixing with non-asyncio: async from threads

```python
def handle_message_sync(loop, msg):
    data = json.loads(msg.json_data)
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(handle_message(data))

def consume_from_google_pubsub(loop):
    client = get_subscriber_client()
    callback = functools.partial(handle_message_sync, loop)
    client.subscribe(SUBSCRIPTION, callback)
```

…gave it the main event loop we're using?

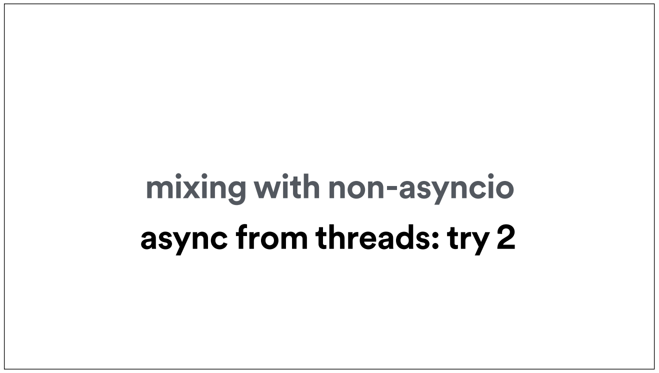


mixing with non-asyncio: async from threads

```
18:08:09,761 INFO: Running something else
18:08:09,826 INFO: Consumed 5236
18:08:09,826 INFO: Consumed 5237
18:08:10,543 INFO: Handling Message(inst_name='xbci')
18:08:10,543 INFO: Handling Message(inst_name='e8x5')
18:08:10,544 INFO: Running something else
18:08:10,721 INFO: Saved Message(inst_name='e8x5') into database
18:08:10,828 INFO: Saved Message(inst_name='xbci') into database
18:08:10,828 ERROR: Caught exception: Could not restart xbci.example.net
18:08:11,167 INFO: Running something else
18:08:11,549 INFO: Restarted e8x5.example.net
18:08:11,821 INFO: Done. Acked 5236
18:08:12,108 INFO: Running something else
18:08:12,276 INFO: Done. Acked 5237
18:08:12,839 INFO: Running something else
18:08:12,841 INFO: Consumed 5241
18:08:12,842 INFO: Consumed 5242
```

So yes! Look at that! It worked!

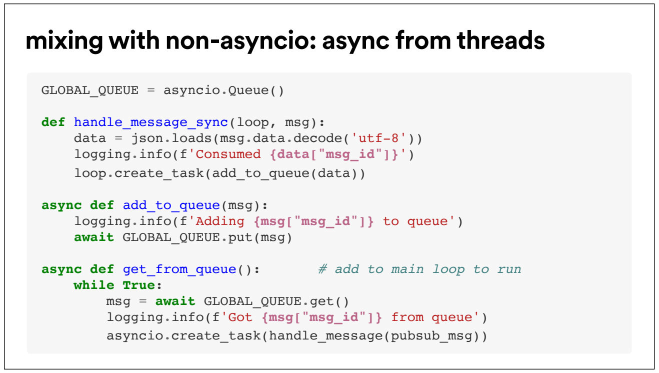
But this is actually deceptive. We're lucky it works.



mixing with non-asyncio

async from threads: try 2

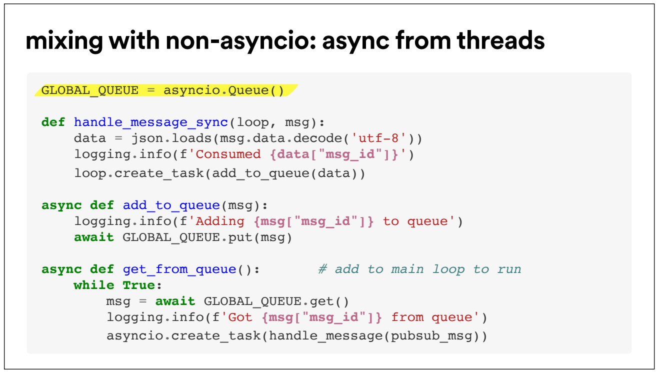
I’ll show you why -



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

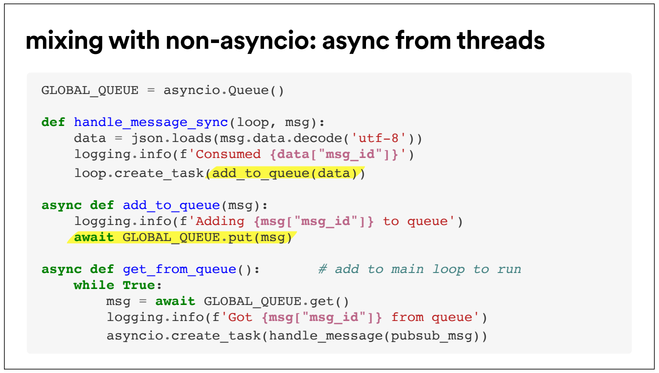
we’ll share some object between the threaded code in the callback, and the asynchronous code when handling the message, we see that we've shot ourselves in the foot.



mixing with non-asyncio: async from threads

```python
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

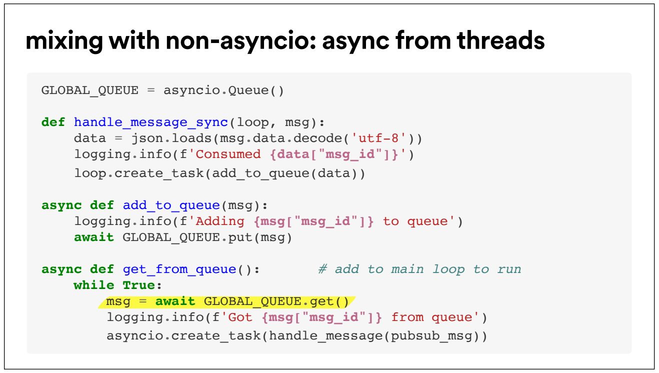
Let's create a shared queue



mixing with non-asyncio: async from threads

```python
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

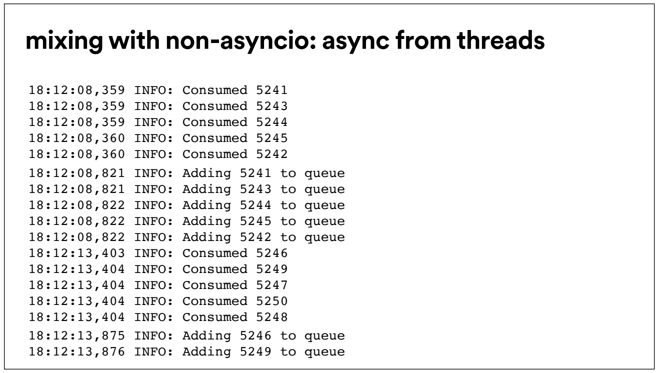
that our synchronous function will add to from one thread, and then



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

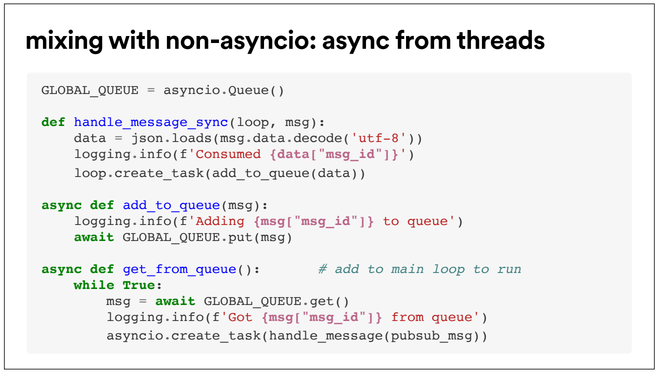
we’ll read off in another thread to continue processing it.



mixing with non-asyncio: async from threads

```
18:12:08,359 INFO: Consumed 5241
18:12:08,359 INFO: Consumed 5243
18:12:08,359 INFO: Consumed 5244
18:12:08,360 INFO: Consumed 5245
18:12:08,360 INFO: Consumed 5242
18:12:08,821 INFO: Adding 5241 to queue
18:12:08,821 INFO: Adding 5243 to queue
18:12:08,822 INFO: Adding 5244 to queue
18:12:08,822 INFO: Adding 5245 to queue
18:12:08,822 INFO: Adding 5242 to queue
18:12:13,403 INFO: Consumed 5246
18:12:13,404 INFO: Consumed 5249
18:12:13,404 INFO: Consumed 5247
18:12:13,404 INFO: Consumed 5250
18:12:13,404 INFO: Consumed 5248
18:12:13,875 INFO: Adding 5246 to queue
18:12:13,876 INFO: Adding 5249 to queue
```

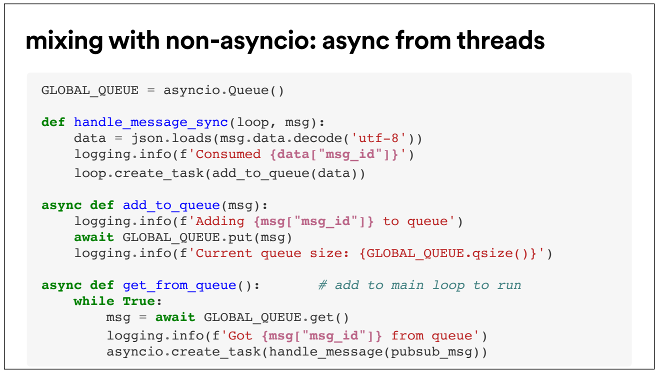
Running it, we see something funky. The log line that gets from the global queue never shows; it doesn't ever look like we consume from our global queue.



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

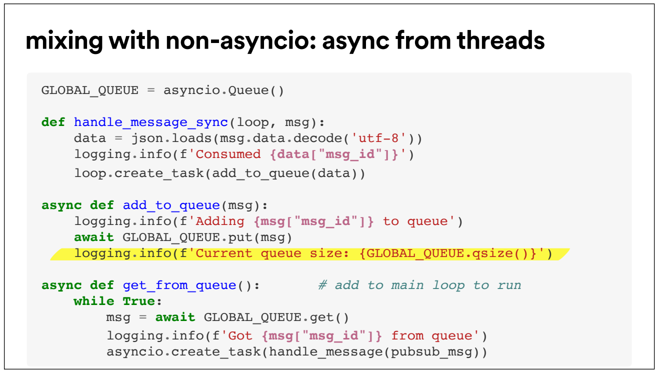
If we add a line in our



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)
    logging.info(f'Current queue size: {GLOBAL_QUEUE.qsize()}')

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

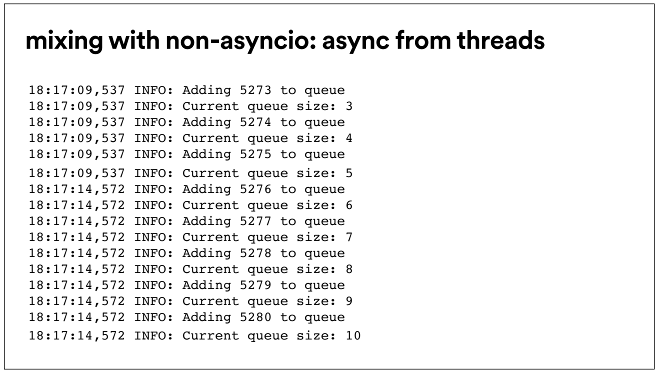
add_to_queue coroutine



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)
    logging.info(f'Current queue size: {GLOBAL_QUEUE.qsize()}')

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

to see the queue size:

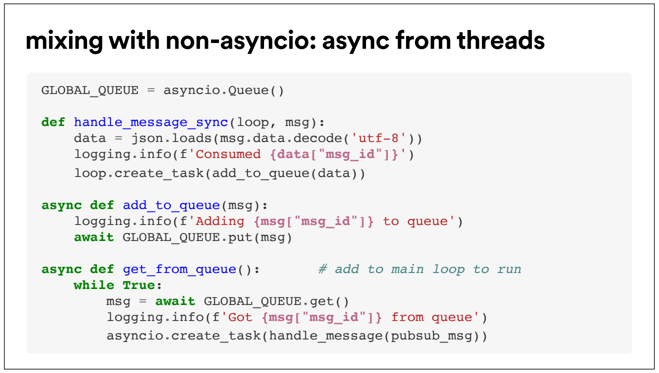


mixing with non-asyncio: async from threads

```
18:17:09,537 INFO: Adding 5273 to queue
18:17:09,537 INFO: Current queue size: 3
18:17:09,537 INFO: Adding 5274 to queue
18:17:09,537 INFO: Current queue size: 4
18:17:09,537 INFO: Adding 5275 to queue
18:17:09,537 INFO: Current queue size: 5
18:17:14,572 INFO: Adding 5276 to queue
18:17:14,572 INFO: Current queue size: 6
18:17:14,572 INFO: Adding 5277 to queue
18:17:14,572 INFO: Current queue size: 7
18:17:14,572 INFO: Adding 5278 to queue
18:17:14,572 INFO: Current queue size: 8
18:17:14,572 INFO: Adding 5279 to queue
18:17:14,572 INFO: Current queue size: 9
18:17:14,572 INFO: Adding 5280 to queue
18:17:14,572 INFO: Current queue size: 10
```

We can see that the queue is ever-growing, and in fact we're not reading from it.

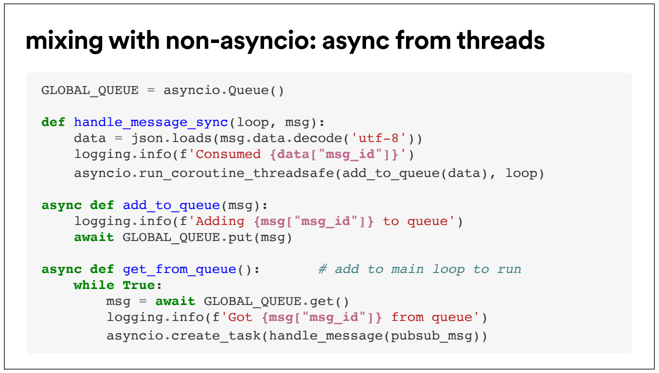
I'm sure a lot of you see what's going on here. We're not thread safe. /facepalm/



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    loop.create_task(add_to_queue(data))

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

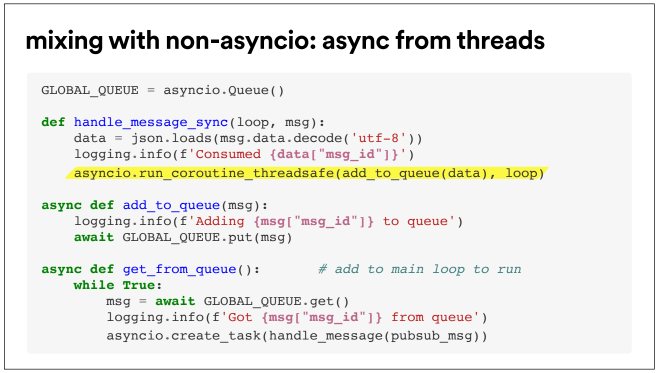
So instead of creating a task on the main event loop from a different thread



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    asyncio.run_coroutine_threadsafe(add_to_queue(data), loop)

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

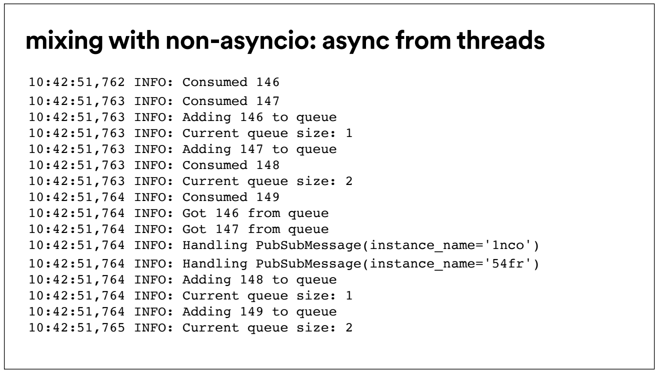
Let's make use of



mixing with non-asyncio: async from threads

```
GLOBAL_QUEUE = asyncio.Queue()

def handle_message_sync(loop, msg):
    data = json.loads(msg.data.decode('utf-8'))
    logging.info(f'Consumed {data["msg_id"]}')
    asyncio.run_coroutine_threadsafe(add_to_queue(data), loop)

async def add_to_queue(msg):
    logging.info(f'Adding {msg["msg_id"]} to queue')
    await GLOBAL_QUEUE.put(msg)

async def get_from_queue():       # add to main loop to run
    while True:
        msg = await GLOBAL_QUEUE.get()
        logging.info(f'Got {msg["msg_id"]} from queue')
        asyncio.create_task(handle_message(pubsub_msg))
```

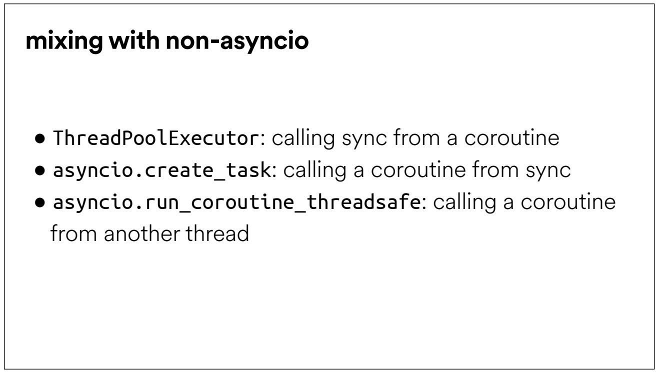
asyncio.run_coroutine_threadsafe to see what happens; the “threadsafe” in the API name should give a clue



mixing with non-asyncio: async from threads

```
10:42:51,762 INFO: Consumed 146
10:42:51,763 INFO: Consumed 147
10:42:51,763 INFO: Adding 146 to queue
10:42:51,763 INFO: Current queue size: 1
10:42:51,763 INFO: Adding 147 to queue
10:42:51,763 INFO: Consumed 148
10:42:51,763 INFO: Current queue size: 2
10:42:51,764 INFO: Consumed 149
10:42:51,764 INFO: Got 146 from queue
10:42:51,764 INFO: Got 147 from queue
10:42:51,764 INFO: Handling PubSubMessage(instance_name='1nco')
10:42:51,764 INFO: Handling PubSubMessage(instance_name='54fr')
10:42:51,764 INFO: Adding 148 to queue
10:42:51,764 INFO: Current queue size: 1
10:42:51,764 INFO: Adding 149 to queue
10:42:51,765 INFO: Current queue size: 2
```

And finally we have it!



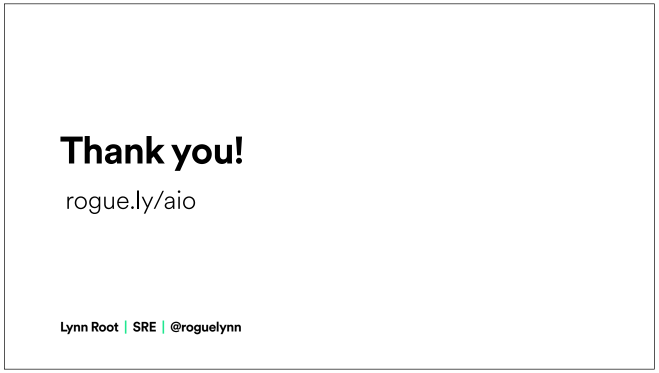
mixing with non-asyncio

- `ThreadPoolExecutor`: calling sync from a coroutine
- `asyncio.create_task`: calling a coroutine from sync
- `asyncio.run_coroutine_threadsafe`: calling a coroutine from another thread

<TODO: flesh out more>

In my opinion, it isn't difficult to work with synchronous code with asyncio.

However, it is difficult to work with threads, particularly with asyncio. If you must, use the _threadsafe APIs that asyncio gives you.



Thank you!

rogue.ly/aio

Lynn Root | SRE | @roguelynn

So in essence, this talk is something I would have liked a year ago; so I'm speaking to past Lynn here. But I'm hoping there are others that benefit from a use case that's not a web crawler.

Thanks!