

There are several useful debugging features built into `asyncio`.

Because applications built on `asyncio` are highly sensitive to greedy coroutines failing to yield control, there is support for detecting slow callbacks built into the event loop. Turn it on by enabling debugging, and control the definition of “slow” by setting the `slow_callback_duration` property of the loop to the number of seconds after which a warning should be emitted.

Finally, if an application using `asyncio` exits without cleaning up some of the coroutines or other resources, that may mean there is a logic error preventing some of the application code from running. Enabling `ResourceWarning` warnings causes these cases to be reported when the program exits.

```
# asyncio_debug.py

import argparse
import asyncio
import logging
import sys
import time
import warnings

parser = argparse.ArgumentParser('debugging asyncio')
parser.add_argument(
    '-v',
    dest='verbose',
    default=False,
    action='store_true',
)
args = parser.parse_args()

logging.basicConfig(
    level=logging.DEBUG,
    format='%(levelname)7s: %(message)s',
    stream=sys.stderr,
)
LOG = logging.getLogger('')

async def inner():
    LOG.info('inner starting')
    # Use a blocking sleep to simulate
    # doing work inside the function.
    time.sleep(0.1)
    LOG.info('inner completed')

async def outer(loop):
    LOG.info('outer starting')
    await asyncio.ensure_future(loop.create_task(inner()))
    LOG.info('outer completed')

event_loop = asyncio.get_event_loop()
if args.verbose:
    LOG.info('enabling debugging')

    # Enable debugging
    event_loop.set_debug(True)
```

```

# Make the threshold for "slow" tasks very very small for
# illustration. The default is 0.1, or 100 milliseconds.
event_loop.slow_callback_duration = 0.001

# Report all mistakes managing asynchronous resources.
warnings.simplefilter('always', ResourceWarning)

LOG.info('entering event loop')
event_loop.run_until_complete(outer(event_loop))

```

When run without debugging enabled, everything looks fine with this application.

```

$ python3 asyncio_debug.py

DEBUG: Using selector: KqueueSelector
INFO: entering event loop
INFO: outer starting
INFO: inner starting
INFO: inner completed
INFO: outer completed

```

Turning on debugging exposes some of the issues it has, including the fact that although `inner()` finishes it takes more time to do so than the `slow_callback_duration` that has been set and that the event loop is not being properly closed when the program exits.

```

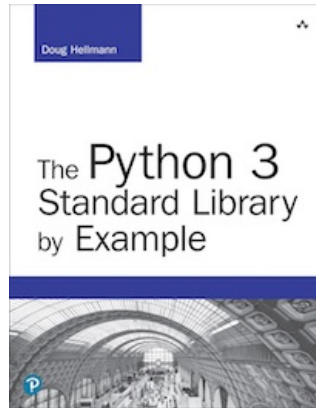
$ python3 asyncio_debug.py -v

DEBUG: Using selector: KqueueSelector
INFO: enabling debugging
INFO: entering event loop
INFO: outer starting
WARNING: Executing <Task pending coro=<outer() running at
asyncio_debug.py:43> wait_for=<Task pending coro=<inner()
running at asyncio_debug.py:33> cb=[<TaskWakeupMethWrapper
object at 0x106e0d288>()] created at asyncio_debug.py:43>
cb=[_run_until_complete_cb() at
.../lib/python3.7/asyncio/base_events.py:158] created at
.../lib/python3.7/asyncio/base_events.py:552> took 0.001 seconds
INFO: inner starting
INFO: inner completed
WARNING: Executing <Task finished coro=<inner() done, defined at
asyncio_debug.py:33> result=None created at asyncio_debug.py:43>
took 0.101 seconds
INFO: outer completed

```

Navigation

- Combining Coroutines with Threads and Processes
- concurrent.futures — Manage Pools of Concurrent Tasks



[Get the book](#)

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for [examples for Python 2?](#)

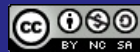
This Site

 [Module Index](#)

 [Index](#)



© Copyright 2019, Doug Hellmann



Other Writing

 [Blog](#)

 [The Python Standard Library By Example](#)