Combining Coroutines with Threads and Processes

A lot of existing libraries are not ready to be used with `asyncio` natively. They may block, or depend on concurrency features not available through the module. It is still possible to use those libraries in an application based on `asyncio` by using an *executor* from `concurrent.futures` to run the code either in a separate thread or a separate process.

Threads

The `run_in_executor()` method of the event loop takes an executor instance, a regular callable to invoke, and any arguments to be passed to the callable. It returns a `Future` that can be used to wait for the function to finish its work and return something. If no executor is passed in, a `ThreadPoolExecutor` is created. This example explicitly creates an executor to limit the number of worker threads it will have available.

A `ThreadPoolExecutor` starts its worker threads and then calls each of the provided functions once in a thread. This example shows how to combine `run_in_executor()` and `wait()` to have a coroutine yield control to the event loop while blocking functions run in separate threads, and then wake back up when those functions are finished.

```
# asyncio_executor_thread.py

import asyncio
import concurrent.futures
import logging
import sys
import time


def blocks(n):
    log = logging.getLogger('blocks({})'.format(n))
    log.info('running')
    time.sleep(0.1)
    log.info('done')
    return n ** 2


async def run_blocking_tasks(executor):
    log = logging.getLogger('run_blocking_tasks')
    log.info('starting')

    log.info('creating executor tasks')
    loop = asyncio.get_event_loop()
    blocking_tasks = [
        loop.run_in_executor(executor, blocks, i)
        for i in range(6)
    ]
    log.info('waiting for executor tasks')
    completed, pending = await asyncio.wait(blocking_tasks)
    results = [t.result() for t in completed]
    log.info('results: {!r}'.format(results))

    log.info('exiting')


if __name__ == '__main__':
    # Configure logging to show the name of the thread
    # where the log message originates.
    logging.basicConfig(
        level=logging.INFO,
        format='%(threadName)10s %(name)18s: %(message)s',
        stream=sys.stderr,
    )

    # Create a limited thread pool.
    executor = concurrent.futures.ThreadPoolExecutor(
```

```
    executor = concurrent.futures.ThreadPoolExecutor(
        max_workers=3,
    )

    event_loop = asyncio.get_event_loop()
    try:
        event_loop.run_until_complete(
            run_blocking_tasks(executor)
        )
    finally:
        event_loop.close()
```

asyncio_executor_thread.py uses logging to conveniently indicate which thread and function are producing each log message. Because a separate logger is used in each call to `blocks()`, the output clearly shows the same threads being reused to call multiple copies of the function with different arguments.

```
$ python3 asyncio_executor_thread.py

MainThread run_blocking_tasks: starting
MainThread run_blocking_tasks: creating executor tasks
ThreadPoolExecutor-0_0          blocks(0): running
ThreadPoolExecutor-0_1          blocks(1): running
ThreadPoolExecutor-0_2          blocks(2): running
MainThread run_blocking_tasks: waiting for executor tasks
ThreadPoolExecutor-0_0          blocks(0): done
ThreadPoolExecutor-0_1          blocks(1): done
ThreadPoolExecutor-0_2          blocks(2): done
ThreadPoolExecutor-0_0          blocks(3): running
ThreadPoolExecutor-0_1          blocks(4): running
ThreadPoolExecutor-0_2          blocks(5): running
ThreadPoolExecutor-0_0          blocks(3): done
ThreadPoolExecutor-0_2          blocks(5): done
ThreadPoolExecutor-0_1          blocks(4): done
MainThread run_blocking_tasks: results: [0, 9, 16, 25, 1, 4]
MainThread run_blocking_tasks: exiting
```

Processes

A `ProcessPoolExecutor` works in much the same way, creating a set of worker processes instead of threads. Using separate processes requires more system resources, but for computationally-intensive operations it can make sense to run a separate task on each CPU core.

```
# asyncio_executor_process.py

# changes from asyncio_executor_thread.py

if __name__ == '__main__':
    # Configure logging to show the id of the process
    # where the log message originates.
    logging.basicConfig(
        level=logging.INFO,
        format='PID %(process)5s %(name)18s: %(message)s',
        stream=sys.stderr,
    )

    # Create a limited process pool.
    executor = concurrent.futures.ProcessPoolExecutor(
        max_workers=3,
    )

    event_loop = asyncio.get_event_loop()
    try:
        event_loop.run_until_complete(
            run_blocking_tasks(executor)
        )
    finally:
        event_loop.close()
```

The only change needed to move from threads to processes is to create a different type of executor. This example also changes the logging format string to include the process id instead of the thread name, to demonstrate that the tasks are in

changes the logging format string to include the process id instead of the thread name, to demonstrate that the tasks are in fact running in separate processes.

```
$ python3 asyncio_executor_process.py

PID 40498 run_blocking_tasks: starting
PID 40498 run_blocking_tasks: creating executor tasks
PID 40498 run_blocking_tasks: waiting for executor tasks
PID 40499           blocks(0): running
PID 40500           blocks(1): running
PID 40501           blocks(2): running
PID 40499           blocks(0): done
PID 40500           blocks(1): done
PID 40501           blocks(2): done
PID 40500           blocks(3): running
PID 40499           blocks(4): running
PID 40501           blocks(5): running
PID 40499           blocks(4): done
PID 40500           blocks(3): done
PID 40501           blocks(5): done
PID 40498 run_blocking_tasks: results: [1, 4, 9, 0, 16, 25]
PID 40498 run_blocking_tasks: exiting
```

Receiving Unix Signals

Debugging with asyncio

Quick Links

Threads
Processes

This page was last updated 2018-03-18.

Navigation

Receiving Unix Signals
Debugging with asyncio

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

Other Writing

- Blog
- The Python Standard Library By Example