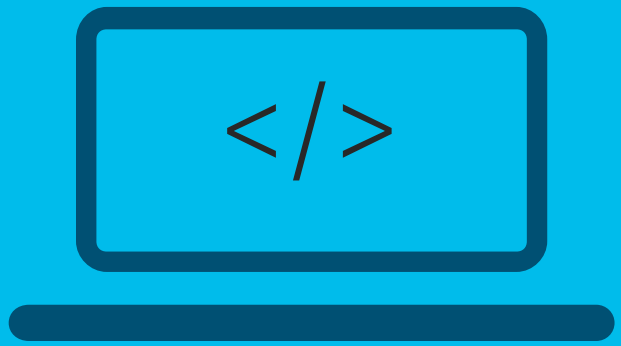
Coding Hour – Cisco EMEAR

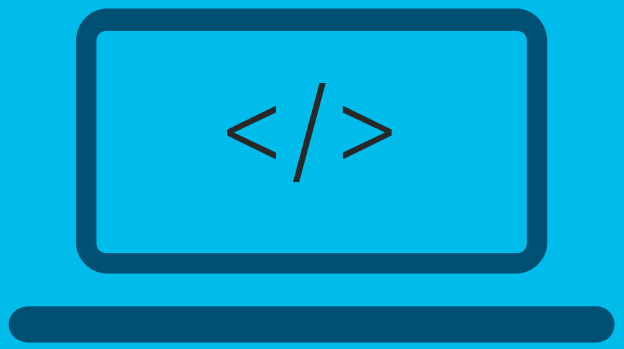
[broadcasting from Frankfurt]



23 July 2020

DEEP PYTHON SERIES

[Asyncio, cooperative multitasking for I/O bound tasks]



Today Speaker

Johannes Krohn @BU

13 August 2020

Agenda

Code, examples available at
https://github.com/jeokrohn/EMEAR_Coding_asyncio

- Cooperative vs Preemtive Multitasking
- Why asyncio
- Asyncio Overview
- Running Coroutines, Scheduling Tasks
- Race Conditions
- Results
- Blocking Calls
- Aiohttp
- Summary

Cooperative vs Preemptive Multitasking

	Preemptive	Cooperative
Task switch	Scheduler switches between tasks	Tasks explicitly pass control
When	Switch can happen at any point	Switch can only happen at explicit points
Risks	race conditions	hogging

Why `asyncio`?

- Threading suffers from...
  - CPU context switching cost
    - Think thousands of tasks
  - Race conditions
  - Deadlocks
  - Resource starvation
- `asyncio` allows to avoid these issues

`asyncio` Programming

- Single-threaded, single-process
- Cooperative multitasking
- “feeling of concurrency”
- What does ”asynchronous” mean?
  - Async code can “pause” (wait for a result or events) and let other code run
  - Quasi concurrent execution of multiple tasks
  - ”Cooperative”: execution is passed on when one task pauses

`asyncio` High-Level Overview

- Event loop: scheduling of asynchronous task
  - Single thread, single process
- Designed for I/O bound tasks
- Cooperative multitasking
- Options
  - Callbacks
  - Async/await
- Keywords: `async` and `await`
  - Coroutine function: `async def`
  - Task switch: `await`

“Event Loop”

- Register tasks to be executed
- Execute, delay, cancel a task
- Handle events related to these operations
- While one task waits for I/O another task is scheduled to run

→ Mechanism to schedule async tasks and get them executed

”Awaitables”

- Coroutines
  - Coroutine function (`async def`)
  - Coroutine object (returned by calling coroutine function)
- Tasks
  - `asyncio.create_task`
- Futures
  - Represents future result of async operation
  - Similar to `concurrent.futures.Future` .. but awaitable

Running Coroutine, Scheduling tasks

- `asyncio.run`: run a coroutine
- `asyncio.create_task`: create and schedule a task

Demo: 00 running.py

Demo: 01 running_details.py

```python
import asyncio


async def main():
    print('inside main!')


if __name__ == '__main__':
    asyncio.run(main())
```

Scheduling tasks

Demo: 02 tasks.py

```
[INFO] __main__: scheduling tasks
[INFO] __main__: Done
[INFO] __main__: wait_some_time(5): before sleep
[INFO] __main__: wait_some_time(4): before sleep
[INFO] __main__: wait_some_time(3): before sleep
[INFO] __main__: wait_some_time(2): before sleep
[INFO] __main__: wait_some_time(1): before sleep

Process finished with exit code 0
```

- `asyncio` tasks don't need to be started
  - `create_task` also schedules tasks for execution
- Coroutines don't have the parent/child relationship like threads:
  - Coroutine terminates before tasks created by that coroutine terminates
- Need to explicitly wait for coroutine termination: `asyncio.wait`
  - ALL_COMPLETED, FIRST_EXCEPTION, FIRST_COMPLETED

Demo: 03 tasks.py

Coroutines Have to be Awaited!

- Creating a coroutine object w/o awaiting it creates a runtime warning
- Look for calls to coroutine functions w/o `await`!!

`asyncio.sleep(wait)`

```
02 tasks.py:14: RuntimeWarning: coroutine 'sleep' was never awaited
  asyncio.sleep(wait)
RuntimeWarning: Enable tracemalloc to get the object allocation traceback
```

A Shortcut For `asyncio.wait`

- `asyncio.wait` expects awaitable objects
- A coroutine object is `await`able
- Instead of scheduling tasks and then `wait`ing for the tasks we can also pass the list of coroutine objects directly

Demo 04 wait_for_coroutines.py

```
await asyncio.wait([wait_some_time(wait=TASKS - i) for i in range(TASKS)])
```

Race Conditions?

Demo: 05 update_counter.py

- Tasks never get interrupted (in contrast to threads)
- No race conditions!
- No need to protect read/update/write operations by locks
  - .. As long as there is no `await` allowing a task switch

Getting Results back from Coroutines

- Remember the pain we had with threads?
  - Passing immutables..
- `concurrent.futures` offered the resolution: futures
- Luckily w/ asyncio we can access the results directly ☺

Getting Results back from Coroutines

Demo: 06 getting results.py

```
coros = [update_counter(_) for _ in range(TASKS)]
done, pending = await asyncio.wait(coros)
for task in done:
    id, val = task.result()
    log.info(f'update_counter({id}) set the counter to {val}')
```

- Simply call result() of a task to get the returned result

Getting Results back from Coroutines

Demo: 07 gather.py

```
coros = [update_counter(_) for _ in range(TASKS)]
results = await asyncio.gather(*coros, return_exceptions=False)
for id, val in results:
    log.info(f'update_counter({id}) set the counter to {val}')
```

- `asyncio.gather` runs awaitable objects and directly returns the results
- `return_exception` controls whether an exception in any of the awaitables should be raised or returned

- asyncio.as_completed allows to iterate over completed awaitables
- Returns a future that can be awaited to get the result of the next completed awaitable

Demo: 08 as_completed.py

Blocking Synchronous Calls

Demo: 09 synchronous_operation.py

- Calls to synchronous code can block asynchronous tasks
- We need a way to schedule synchronous calls from asynchronous tasks

```
2020-08-12 14:36:08,687 [INFO] __main__: scheduling tasks
2020-08-12 14:36:08,687 [INFO] __main__: tasks scheduled
2020-08-12 14:36:08,687 [INFO] __main__: blocking_sync_call: awaiting noop
2020-08-12 14:36:08,687 [INFO] __main__: blocking_sync_call: before blocking call
2020-08-12 14:36:12,692 [INFO] __main__: blocking_sync_call: after blocking call
2020-08-12 14:36:12,692 [INFO] __main__: blocking_sync_call: awaiting noop again
2020-08-12 14:36:12,692 [INFO] __main__: blocking_sync_call: Done!
2020-08-12 14:36:12,693 [INFO] __main__: 0: doing some preparation
2020-08-12 14:36:12,693 [INFO] __main__: 1: doing some preparation
```

`concurrent.futures` .. To the Rescue!

- Abstraction for `threading` (... and `multiprocessing`)
- Allows us to use non-asyncio modules
- `concurrent.futures` is “asyncio friendly”
  - Interoperable w/ asyncio event loop
- Event loop offers `run_in_executor()`

Demo 10 run_in_executor.py

```
with ThreadPoolExecutor() as pool:
    await asyncio.get_running_loop().run_in_executor(pool, slow_sync_call)

await asyncio.get_running_loop().run_in_executor(None, slow_sync_call)
```

`aiohttp`: Async Web Clients

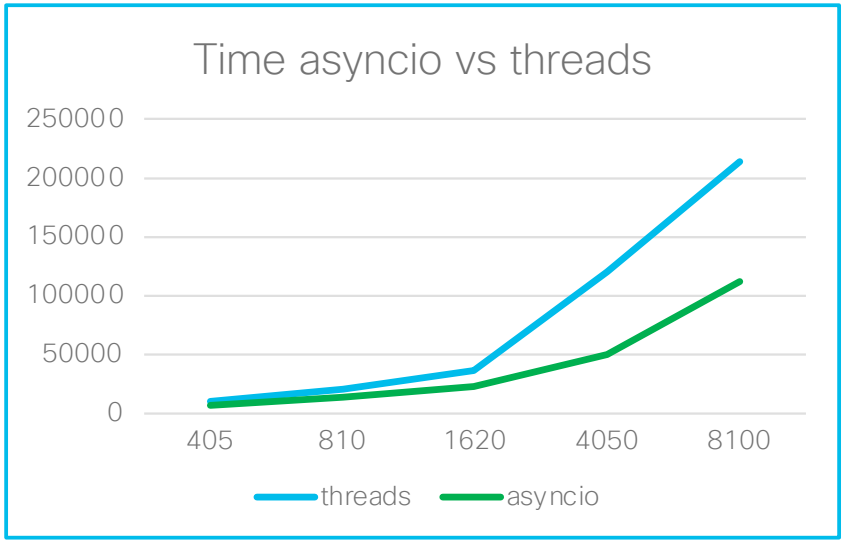
- `asyncio` web client
- Replacement for synchronous `requests` library
- High-Level: `aiohttp.ClientSession`() object
  - Connection pooling
  - Cookie jar
  - Keepalive
  - ...
- Low-Level: `aiohttp.request()`

https://aiohttp.readthedocs.io/en/stable/

Scraping a Web Site

Demo: 11 field notices.py

- Asyncio generally is faster than threading
- Less overhead
- Task switching ”cost” lower than w/ threading



Finding: `asyncio/aiohttp`

- Creating all tasks at the same time:
  - HTTP GET requests sent for all URLs
  - Issues:
    - Servicing the responses takes too long → server timeouts
    - Too many open files
  - Need to throttle the requests (similar to `max_workers` with `concurrent.futures`
  - For example: semaphore

Summary

What Form of Concurrency is “best”?

```
if io_bound:
    if io_very_slow or lots_of_connections:
        print('use asyncio')
    else:
        print('use threads')
else:
    print('use processes')
```

Problems with asyncio

- Need to think “asyncio”
- Hard to migrate existing code
- Some sync libraries don’t have async equivalence (for example `webexteamssdk`)

Other Services

- FTP (server/client): aioftp
- Timeouts: async-timeout
- SSH: asyncssh
- Client WebSockets: aiohttp
- Interaction with network devices: netdev
- Others: https://github.com/aio-libs

References

- asyncio — Asynchronous I/O
- Welcome to AIOHTTP
- Async IO in Python: A Complete Walkthrough
- Guide to Concurrency in Python with Asyncio
- Awesome asyncio - A carefully curated list of awesome Python asyncio frameworks, libraries, software and resources.


CISCO