Async IO and Coroutines

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Last week we looked at threading.

- Multiple concurrent threads of execution.
- Threads can speed up an I/O bound program.
- Threads may be swapped in and out at any time. We don't control when threads are executed. This can lead to challenging bugs.

But

- We can write programs in a familiar way and add threading to parts of the program that benefit from it.

A different approach: Async IO

Often we write code that blocks, typically while waiting on IO. It generally clear to us where this happens in our code. When using an async IO style of coding, we explicitly mark these points in our code so that execution can be yielded to some other part of the process. This is called *cooperative multitasking*.

Review: Threading

```
import threading
from time import sleep

def slow(x):
    sleep(10)
    return x

def do_threaded_tasks(num):
    threads = []
    for n in range(num):
        t = threading.Thread(target=slow, args=(n,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
```

One key thing here is that the function `slow()` doesn't know or care that it will be run in a seperate thread.

An AsyncIO Version

```
import asyncio

async def slow(n):
    await asyncio.sleep(3)
    return n

async def run_async_tasks(n):
    tasks = [asyncio.create_task(slow(i)) for i in range(n)]
    results = await asyncio.gather(*tasks)
    return results

if __name__ == '__main__':
    results = asyncio.run(run_async_tasks(5))
    print(results)
```

What's going on here?

The main new thing here is the addition of the words `async` and `await`. These became reserved words in Python as of version 3.7.

`async def` defines a *coroutine*. A typical Python function is a *subroutine*. A subroutine runs from start to finish in an uninterrupted manner (threads notwithstanding). A coroutine can suspend and resume execution later from where it left off.

We must use `await` to call a coroutine and only coroutines can call other coroutines. The use of `await` signals the event loop that the calling coroutine can be suspended until the result of the `awaited` call is ready.

Also, `asyncio.run()` provides an event loop in which our coroutines can be run.

On the one hand, this code is simpler that working with threads. Everything runs in one thread and we don't have to worry about race conditions or deadlocks.

On the other hand, you can't just take some IO bound code and speed it up by sprinkling some `awaits` on it. You have to plan for how you're going to use coroutines from the bottom up.

Example, HTTP Requests

```
import asyncio
from aiohttp import ClientSession

async def get_web(session, url):
    resp =  await session.get(url)
    content = await resp.text()
    return content

async def web_async_tasks():
    urls = ['https://google.com', 'https://github.io']
    async with ClientSession() as session:
        tasks = [asyncio.create_task(get_web(session, url))
            for url in urls]
        results = await asyncio.gather(*tasks)
        return results

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

This example isn't particularly harder than a non-async IO version. The key difference is that we needed a library, `aiohttp` made for use in async IO code.

We could have used the excellent `requests` library to make our HTTP calls. But we can't `await` a call to `requests.get()` because it's not a coroutine. We can still use `requests`, but the calls will block

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `23-practical` in your practicals repo.
3. Copy the subdirectory, `23-practical` from the class materials into your repo.
4. See the README for directions.
5. We will discuss results in 20ish minutes.

Another Pattern: Producer/Consumer

A design pattern that works well in an async IO context is *producer/consumer*. It comprises the following parts:

- a set of *producers* that prepares data;
- a buffer (FIFO) that holds the data produced by the producers;
- a set of *consumers* that takes data from the buffer and processes it in some way.

Producer

```
async def produce(q):
    num  = randint(0, 5)
    for i in range(num):
        await asyncio.sleep(2)
        await q.put(i)
```

Consumer

```
async def consume(q):
    while True:
        await asyncio.sleep(1)
        val = await q.get()
        print(val)
        q.task_done()
```

Assembling the Pattern

```
async def main():
    q = asyncio.Queue()
    prods = [asyncio.create_task(produce(q))
        for _ in range(N_PRODUCERS)]
    cons = [asyncio.create_task(consume(q))
        for _ in range(N_CONSUMERS)]
    await asyncio.gather(*prods)
    await q.join()
    for c in cons:
        c.cancel()
```

References

- `asyncio`: (https://docs.python.org/3/library/asyncio.html)
- Libraries/frameworks for use with async io: `https://github.com/timofurrer/awesome-asyncio`