Appendix C. Something Completely Different: Async

Our first two appendixes were for beginning programmers, but this one is for those who are a bit advanced.

Like most programming languages, Python has been *synchronous*. It runs through code linearly, a line at a time, from top to bottom. When you call a function, Python jumps into its code, and the caller waits until the function returns before resuming what it was doing.

Your CPU can do only one thing at a time, so synchronous execution makes perfect sense. But it turns out that often a program is not actually running any code, but waiting for something, like data from a file or a network service. This is like us staring at a browser screen while waiting for a site to load. If we could avoid this "busy waiting," we might shorten the total time of our programs. This is also called improving *throughput*.

In Chapter 15, you saw that if you want some concurrency, your choices included threads, processes, or a third-party solution like `gevent` or `twisted`. But there are now a growing number of *asynchronous* answers, both built in to Python and third-party solutions. These coexist with the usual synchronous Python code, but, to borrow a Ghostbusters warning, you can't cross the streams. I'll show you how to avoid any ectoplasmic side effects.

Coroutines and Event Loops

In Python 3.4, Python added a standard *asynchronous* module called `asyncio`. Python 3.5 then added the keywords `async` and `await`. These implement some new concepts:

- *Coroutines* are functions that pause at various points
- An *event loop* that schedules and runs coroutines

These let us write asynchronous code that looks something like the normal synchronous code that we're used to. Otherwise, we'd need to use one of the methods mentioned in Chapter 15 and Chapter 17, and summarized later in "Async Versus…".

Normal multitasking is what your operating system does to your processes. It decides what's fair, who's being a CPU hog, when to open the I/O spigots, and so on. The event loop, however, provides *cooperative multitasking*, in which coroutines indicate when they're able to start and stop. They run in a single thread, so you don't have the potential issues that I mentioned in "Threads".

You *define* a coroutine by putting `async` before its initial `def`. You *call* a coroutine by:

- Putting `await` before it, which quietly adds the coroutine to an existing event loop. You can do this only within another coroutine.
- Or by using `asyncio.run()`, which explicitly starts an event loop.
- Or by using `asyncio.create_task()` or `asyncio.ensure_future()`.

This example uses the first two calling methods:

```
>>> import asyncio
>>>
>>> async def wicked():
...     print("Surrender,")
...     await asyncio.sleep(2)
...     print("Dorothy!")
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

These was a dramatic two-second wait in there that you can't see on a printed page. To prove that we didn't cheat (see Chapter 19 for `timeit` details):

```
>>> from timeit import timeit
>>> timeit("asyncio.run(wicked())", globals=globals(), number=1)
```

```
Surrender,
Dorothy!
2.005701574998966
```

That `asyncio.sleep(2)` call was itself a coroutine, just an example here to fake something time consuming like an API call.

The line `asyncio.run(wicked())` is a way of running a coroutine from synchronous Python code (here, the top level of the program).

The difference from a standard synchronous counterpart (using `time.sleep()`) is that the caller of `wicked()` is not blocked for two seconds while it runs.

The third way to run a coroutine is to create a *task* and `await` it. This example shows the task approach along with the previous two methods:

```
>>> import asyncio
>>>
>>> async def say(phrase, seconds):
...     print(phrase)
...     await asyncio.sleep(seconds)
...
>>> async def wicked():
...     task_1 = asyncio.create_task(say("Surrender,", 2))
...     task_2 = asyncio.create_task(say("Dorothy!", 0))
...     await task_1
...     await task_2
...
>>> asyncio.run(wicked())
Surrender,
Dorothy!
```

If you run this, you'll see that there was no delay between the two lines printing this time. That's because they were separate tasks. `task_1` paused two seconds after printing `Surrender`, but that didn't affect `task_2`.

An `await` is similar to a `yield` in a generator, but rather than returning a value, it marks a spot where the event loop can pause it if needed.

There's lots more where this came from in the docs. Synchronous and asynchronous code can coexist in the same program. Just remember to put `async` before the `def` and `await` before the call of your asynchronous function.

Some more information:

- A list of `asyncio` links.
- Code for an `asyncio` web crawler.

Asyncio Alternatives

Although `asyncio` is a standard Python package, you can use `async` and `await` without it. Coroutines and the event loop are independent. The design of `asyncio` is sometimes criticized, and third-party alternatives have appeared:

- `curio`
- `trio`

Let's show a real example using `trio` and `asks` (an async web framework, modeled on the `requests` API). Example C-1 shows a concurrent web-crawling example using `trio` and `asks`, adapted from a stackoverflow answer. To run this, first `pip install` both `trio` and `asks`.

Example C-1. trio_asks_sites.py

```
import time

import asks
import trio

asks.init("trio")

urls = [
    'https://boredomtherapy.com/bad-taxidermy/',
    'http://www.badtaxidermy.com/',
    'https://crappytaxidermy.com/',
    'https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reign',
]

async def get_one(url, t1):
    r = await asks.get(url)
    t2 = time.time()
    print(f"{(t2-t1):.04}\t{len(r.content)}\t{url}")

async def get_sites(sites):
    t1 = time.time()
    async with trio.open_nursery() as nursery:
        for url in sites:
            nursery.start_soon(get_one, url, t1)

if __name__ == "__main__":
    print("seconds\tbytes\turl")
    trio.run(get_sites, urls)
```

Here's what I got:

```
$ python trio_asks_sites.py
seconds bytes   url
0.1287  5735    https://boredomtherapy.com/bad-taxidermy/
0.2134  146082  https://www.ranker.com/list/bad-taxidermy-pictures/ashley-reigr
0.215   11029   http://www.badtaxidermy.com/
0.3813  52385   https://crappytaxidermy.com/
```

You'll notice that `trio` did not use `asyncio.run()`, but instead its own `trio.open_nursery()`. If you're curious, you can read an essay and discussion of the design decisions behind `trio`.

A new package called `AnyIO` provides a single interface to `asyncio`, `curio`, and `trio`.

In the future, you can expect more async approaches, both in standard Python and from third-party developers.

Async Versus…

As you've seen in many places in this book, there are many techniques for concurrency. How does the async stuff compare with them?

Processes
: This is a good solution if you want to use all the CPU cores on your machine, or multiple machines. But processes are heavy, take a while to start, and require serialization for interprocess communication.

Threads
: Although threads were designed as a "lightweight" alternative to processes, each thread uses a good chunk of memory. Coroutines are much lighter than threads; you can create hundreds of thousands of coroutines on a machine that might only support a few thousand threads.

Green threads
: Green threads like `gevent` work well and look like synchronous code, but they require *monkey-patching* standard Python functions, such as socket libraries.

Callbacks
: Libraries like `twisted` rely on *callbacks*: functions that are called when when certain events occur. This is familiar to GUI and JavaScript programmers.

Queues—These tend to be a large-scale solution, when your data or processes really need more than one machine.

Async Frameworks and Servers

The async additions to Python are recent, and it's taking time for developers to create async versions of frameworks like Flask.

The ASGI standard is an async version of WSGI, discussed further here.

Here are some ASGI web servers:

- `hypercorn`

- `sanic`
- `uvicorn`

And some async web frameworks:

- `aiohttp` —Client *and* server
- `api_hour`
- `asks` —Like `requests`
- `blacksheep`
- `bocadillo`
- `channels`
- `fastapi` —Uses type annotations
- `muffin`
- `quart`
- `responder`
- `sanic`
- `starlette`
- `tornado`
- `vibora`

Finally, some async database interfaces:

- `aiomysql`
- `aioredis`
- `asyncpg`