CHAPTER 18

Concurrency with asyncio

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

Not the same, but related.

One is about structure, one is about execution.

Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.[1]

— Rob Pike
Co-inventor of the Go language

Professor Imre Simon[2] liked to say there are two major sins in science: using different words to mean the same thing and using one word to mean different things. If you do any research on concurrent or parallel programming you will find different definitions for "concurrency" and "parallelism." I will adopt the informal definitions by Rob Pike, quoted above.

For real parallelism, you must have multiple cores. A modern laptop has four CPU cores but is routinely running more than 100 processes at any given time under normal, casual use. So, in practice, most processing happens concurrently and not in parallel. The computer is constantly dealing with 100+ processes, making sure each has an opportunity to make progress, even if the CPU itself can't do more than four things at once. Ten years ago we used machines that were also able to handle 100 processes concurrently, but on a single core. That's why Rob Pike titled that talk "Concurrency Is Not Parallelism (It's Better)."

1. Slide 5 of the talk "Concurrency Is Not Parallelism (It's Better)".
2. Imre Simon (1943–2009) was a pioneer of computer science in Brazil who made seminal contributions to Automata Theory and started the field of Tropical Mathematics. He was also an advocate of free software and free culture. I was fortunate to study, work, and hang out with him.

This chapter introduces asyncio, a package that implements concurrency with coroutines driven by an event loop. It's one of the largest and most ambitious libraries ever added to Python. Guido van Rossum developed asyncio outside of the Python repository and gave the project a code name of "Tulip"—so you'll see references to that flower when researching this topic online. For example, the main discussion group is still called python-tulip.

Tulip was renamed to asyncio when it was added to the standard library in Python 3.4. It's also compatible with Python 3.3—you can find it on PyPI under the new official name. Because it uses `yield from` expressions extensively, asyncio is incompatible with older versions of Python.

> The Trollius project—also named after a flower—is a backport of `asyncio` to Python 2.6 and newer, replacing `yield from` with `yield` and clever callables named `From` and `Return`. A `yield from` … expression becomes `yield From(…)`; and when a coroutine needs to return a result, you write `raise Return(result)` instead of `return result`. Trollius is led by Victor Stinner, who is also an `asyncio` core developer, and who kindly agreed to review this chapter as this book was going into production.

In this chapter we'll see:

- A comparison between a simple threaded program and the `asyncio` equivalent, showing the relationship between threads and asynchronous tasks
- How the `asyncio.Future` class differs from `concurrent.futures.Future`
- Asynchronous versions of the flag download examples from Chapter 17
- How asynchronous programming manages high concurrency in network applications, without using threads or processes
- How coroutines are a major improvement over callbacks for asynchronous programming
- How to avoid blocking the event loop by offloading blocking operations to a thread pool
- Writing `asyncio` servers, and how to rethink web applications for high concurrency
- Why `asyncio` is poised to have a big impact in the Python ecosystem

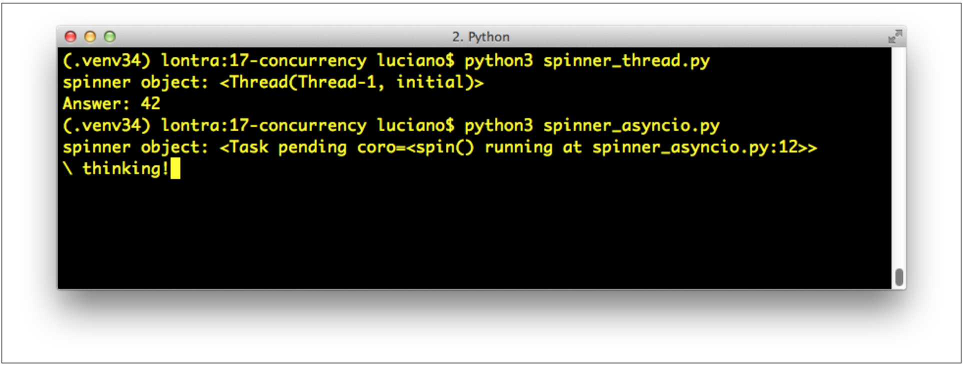
Let's get started with the simple example contrasting `threading` and `asyncio`.

Thread Versus Coroutine: A Comparison

During a discussion about threads and the GIL, Michele Simionato posted a simple but fun example using `multiprocessing` to display an animated spinner made with the ASCII characters `"|/-\"` on the console while some long computation is running.

I adapted Simionato's example to use a thread with the `Threading` module and then a coroutine with `asyncio`, so you can see the two examples side by side and understand how to code concurrent behavior without threads.

The output shown in Examples 18-1 and 18-2 is animated, so you really should run the scripts to see what happens. If you're in the subway (or somewhere else without a WiFi connection), take a look at Figure 18-1 and imagine the \ bar before the word "thinking" is spinning.



```
(.venv34) lontra:17-concurrency luciano$ python3 spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.venv34) lontra:17-concurrency luciano$ python3 spinner_asyncio.py
spinner object: <Task pending coro=<spin() running at spinner_asyncio.py:12>>
\ thinking!
```

Figure 18-1. The scripts spinner_thread.py and spinner_asyncio.py produce similar output: the repr of a spinner object and the text Answer: 42. In the screenshot, spinner_asyncio.py is still running, and the spinner message \ thinking! is shown; when the script ends, that line will be replaced by the Answer: 42.

Let's review the *spinner_thread.py* script first (Example 18-1).

Example 18-1. spinner_thread.py: animating a text spinner with a thread

```
import threading
import itertools
import time
import sys


class Signal:  ❶
    go = True


def spin(msg, signal):  ❷
```

```
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):  ❸
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))  ❹
        time.sleep(.1)
        if not signal.go:  ❺
            break
    write(' ' * len(status) + '\x08' * len(status))  ❻


def slow_function():  ❼
    # pretend waiting a long time for I/O
    time.sleep(3)  ❽
    return 42


def supervisor():  ❾
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)  ❿
    spinner.start()  ⓫
    result = slow_function()  ⓬
    signal.go = False  ⓭
    spinner.join()  ⓮
    return result


def main():
    result = supervisor()  ⓯
    print('Answer:', result)


if __name__ == '__main__':
    main()
```

❶ This class defines a simple mutable object with a `go` attribute we'll use to control the thread from outside.

❷ This function will run in a separate thread. The `signal` argument is an instance of the `Signal` class just defined.

❸ This is actually an infinite loop because `itertools.cycle` produces items cycling from the given sequence forever.

❹ The trick to do text-mode animation: move the cursor back with backspace characters (`\x08`).

❺ If the `go` attribute is no longer `True`, exit the loop.

❻ Clear the status line by overwriting with spaces and moving the cursor back to the beginning.

❼ Imagine this is some costly computation.

❽ Calling `sleep` will block the main thread, but crucially, the GIL will be released so the secondary thread will proceed.

❾ This function sets up the secondary thread, displays the thread object, runs the slow computation, and kills the thread.

❿ Display the secondary thread object. The output looks like `<Thread(Thread-1, initial)>`.

⓫ Start the secondary thread.

⓬ Run `slow_function`; this blocks the main thread. Meanwhile, the spinner is animated by the secondary thread.

⓭ Change the state of the `signal`; this will terminate the `for` loop inside the `spin` function.

⓮ Wait until the `spinner` thread finishes.

⓯ Run the `supervisor` function.

Note that, by design, there is no API for terminating a thread in Python. You must send it a message to shut down. Here I used the `signal.go` attribute: when the main thread sets it to false, the `spinner` thread will eventually notice and exit cleanly.

Now let's see how the same behavior can be achieved with an `@asyncio.coroutine` instead of a thread.

As noted in the "Chapter Summary" on page 498 (Chapter 16), `asyncio` uses a stricter definition of "coroutine." A coroutine suitable for use with the `asyncio` API must use `yield from` and not `yield` in its body. Also, an `asyncio` coroutine should be driven by a caller invoking it through `yield from` or by passing the coroutine to one of the `asyncio` functions such as `asyncio.async(…)` and others covered in this chapter. Finally, the `@asyncio.coroutine` decorator should be applied to coroutines, as shown in the examples.

Take a look at Example 18-2.

Example 18-2. spinner_asyncio.py: animating a text spinner with a coroutine

```
import asyncio
import itertools
import sys
```

```
@asyncio.coroutine  ❶
def spin(msg):  ❷
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            yield from asyncio.sleep(.1)  ❸
        except asyncio.CancelledError:  ❹
            break
    write(' ' * len(status) + '\x08' * len(status))


@asyncio.coroutine
def slow_function():  ❺
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3)  ❻
    return 42


@asyncio.coroutine
def supervisor():  ❼
    spinner = asyncio.async(spin('thinking!'))  ❽
    print('spinner object:', spinner)  ❾
    result = yield from slow_function()  ❿
    spinner.cancel()  ⓫
    return result


def main():
    loop = asyncio.get_event_loop()  ⓬
    result = loop.run_until_complete(supervisor())  ⓭
    loop.close()
    print('Answer:', result)


if __name__ == '__main__':
    main()
```

❶ Coroutines intended for use with `asyncio` should be decorated with `@asyncio.coroutine`. This not mandatory, but is highly advisable. See explanation following this listing.

❷ Here we don't need the `signal` argument that was used to shut down the thread in the `spin` function of Example 18-1.

❸ Use `yield from asyncio.sleep(.1)` instead of just `time.sleep(.1)`, to sleep without blocking the event loop.

❹ If `asyncio.CancelledError` is raised after `spin` wakes up, it's because cancellation was requested, so exit the loop.

❺ `slow_function` is now a coroutine, and uses `yield from` to let the event loop proceed while this coroutine pretends to do I/O by sleeping.

❻ The `yield from asyncio.sleep(3)` expression handles the control flow to the main loop, which will resume this coroutine after the sleep delay.

❼ `supervisor` is now a coroutine as well, so it can drive `slow_function` with `yield from`.

❽ `asyncio.async(…)` schedules the `spin` coroutine to run, wrapping it in a `Task` object, which is returned immediately.

❾ Display the `Task` object. The output looks like `<Task pending coro=<spin() running at spinner_asyncio.py:12>>`.

❿ Drive the `slow_function()`. When that is done, get the returned value. Meanwhile, the event loop will continue running because `slow_function` ultimately uses `yield from asyncio.sleep(3)` to hand control back to the main loop.

⓫ A `Task` object can be cancelled; this raises `asyncio.CancelledError` at the `yield` line where the coroutine is currently suspended. The coroutine may catch the exception and delay or even refuse to cancel.

⓬ Get a reference to the event loop.

⓭ Drive the `supervisor` coroutine to completion; the return value of the coroutine is the return value of this call.

Never use `time.sleep(…)` in `asyncio` coroutines unless you want to block the main thread, therefore freezing the event loop and probably the whole application as well. If a coroutine needs to spend some time doing nothing, it should `yield from asyncio.sleep(DELAY)`.

The use of the `@asyncio.coroutine` decorator is not mandatory, but highly recommended: it makes the coroutines stand out among regular functions, and helps with debugging by issuing a warning when a coroutine is garbage collected without being yielded from—which means some operation was left unfinished and is likely a bug. This is not a *priming decorator*.

Note that the line count of *spinner_thread.py* and *spinner_asyncio.py* is nearly the same. The `supervisor` functions are the heart of these examples. Let's compare them in detail. Example 18-3 lists only the `supervisor` from the `Threading` example.

Example 18-3. spinner_thread.py: the threaded supervisor function

```
def supervisor():
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)
    spinner.start()
    result = slow_function()
    signal.go = False
    spinner.join()
    return result
```

For comparison, Example 18-4 shows the `supervisor` coroutine.

Example 18-4. spinner_asyncio.py: the asynchronous supervisor coroutine

```
@asyncio.coroutine
def supervisor():
    spinner = asyncio.async(spin('thinking!'))
    print('spinner object:', spinner)
    result = yield from slow_function()
    spinner.cancel()
    return result
```

Here is a summary of the main differences to note between the two `supervisor` implementations:

- An `asyncio.Task` is roughly the equivalent of a `threading.Thread`. Victor Stinner, special technical reviewer for this chapter, points out that "a `Task` is like a green thread in libraries that implement cooperative multitasking, such as `gevent`."
- A `Task` drives a coroutine, and a `Thread` invokes a callable.
- You don't instantiate `Task` objects yourself, you get them by passing a coroutine to `asyncio.async(…)` or `loop.create_task(…)`.
- When you get a `Task` object, it is already scheduled to run (e.g., by `asyncio.async`); a `Thread` instance must be explicitly told to run by calling its `start` method.
- In the threaded `supervisor`, the `slow_function` is a plain function and is directly invoked by the thread. In the `asyncio` `supervisor`, `slow_function` is a coroutine driven by `yield from`.
- There's no API to terminate a thread from the outside, because a thread could be interrupted at any point, leaving the system in an invalid state. For tasks, there is

the Task.cancel() instance method, which raises CancelledError inside the coroutine. The coroutine can deal with this by catching the exception in the yield where it's suspended.

- The supervisor coroutine must be executed with loop.run_until_complete in the main function.

This comparison should help you understand how concurrent jobs are orchestrated with asyncio, in contrast to how it's done with the more familiar Threading module.

One final point related to threads versus coroutines: if you've done any nontrivial programming with threads, you know how challenging it is to reason about the program because the scheduler can interrupt a thread at any time. You must remember to hold locks to protect the critical sections of your program, to avoid getting interrupted in the middle of a multistep operation—which could leave data in an invalid state.

With coroutines, everything is protected against interruption by default. You must explicitly yield to let the rest of the program run. Instead of holding locks to synchronize the operations of multiple threads, you have coroutines that are "synchronized" by definition: only one of them is running at any time. And when you want to give up control, you use yield or yield from to give control back to the scheduler. That's why it is possible to safely cancel a coroutine: by definition, a coroutine can only be cancelled when it's suspended at a yield point, so you can perform cleanup by handling the CancelledError exception.

We'll now see how the asyncio.Future class differs from the concurrent.futures.Future class we saw in Chapter 17.

asyncio.Future: Nonblocking by Design

The asyncio.Future and the concurrent.futures.Future classes have mostly the same interface, but are implemented differently and are not interchangeable. PEP-3156 — Asynchronous IO Support Rebooted: the "asyncio" Module has this to say about this unfortunate situation:

> In the future (pun intended) we may unify asyncio.Future and concurrent.futures.Future (e.g., by adding an __iter__ method to the latter that works with yield from).

As mentioned in "Where Are the Futures?" on page 511, futures are created only as the result of scheduling something for execution. In asyncio, BaseEventLoop.create_task(…) takes a coroutine, schedules it to run, and returns an asyncio.Task instance—which is also an instance of asyncio.Future because Task is a subclass of Future designed to wrap a coroutine. This is analogous to how we create concurrent.futures.Future instances by invoking Executor.submit(…).

Like its `concurrent.futures.Future` counterpart, the `asyncio.Future` class provides `.done()`, `.add_done_callback(…)`, and `.results()` methods, among others. The first two methods work as described in "Where Are the Futures?" on page 511, but `.result()` is very different.

In `asyncio.Future`, the `.result()` method takes no arguments, so you can't specify a timeout. Also, if you call `.result()` and the future is not done, it does not block waiting for the result. Instead, an `asyncio.InvalidStateError` is raised.

However, the usual way to get the result of an `asyncio.Future` is to `yield from` it, as we'll see in Example 18-8.

Using `yield from` with a future automatically takes care of waiting for it to finish, without blocking the event loop—because in `asyncio`, `yield from` is used to give control back to the event loop.

Note that using `yield from` with a future is the coroutine equivalent of the functionality offered by `add_done_callback`: instead of triggering a callback, when the delayed operation is done, the event loop sets the result of the future, and the `yield from` expression produces a return value inside our suspended coroutine, allowing it to resume.

In summary, because `asyncio.Future` is designed to work with `yield from`, these methods are often not needed:

- You don't need `my_future.add_done_callback(…)` because you can simply put whatever processing you would do after the future is done in the lines that follow `yield from my_future` in your coroutine. That's the big advantage of having coroutines: functions that can be suspended and resumed.
- You don't need `my_future.result()` because the value of a `yield from` expression on a future is the result (e.g., `result = yield from my_future`).

Of course, there are situations in which `.done()`, `.add_done_callback(…)`, and `.results()` are useful. But in normal usage, `asyncio` futures are driven by `yield from`, not by calling those methods.

We'll now consider how `yield from` and the `asyncio` API brings together futures, tasks, and coroutines.

Yielding from Futures, Tasks, and Coroutines

In `asyncio`, there is a close relationship between futures and coroutines because you can get the result of an `asyncio.Future` by yielding from it. This means that `res = yield from foo()` works if `foo` is a coroutine function (therefore it returns a coroutine object when called) or if `foo` is a plain function that returns a `Future` or `Task` instance.

This is one of the reasons why coroutines and futures are interchangeable in many parts of the asyncio API.

In order to execute, a coroutine must be scheduled, and then it's wrapped in an asyncio.Task. Given a coroutine, there are two main ways of obtaining a Task:

`asyncio.async(coro_or_future, *, loop=None)`
: This function unifies coroutines and futures: the first argument can be either one. If it's a Future or Task, it's returned unchanged. If it's a coroutine, async calls loop.create_task(…) on it to create a Task. An optional event loop may be passed as the loop= keyword argument; if omitted, async gets the loop object by calling asyncio.get_event_loop().

`BaseEventLoop.create_task(coro)`
: This method schedules the coroutine for execution and returns an asyncio.Task object. If called on a custom subclass of BaseEventLoop, the object returned may be an instance of some other Task-compatible class provided by an external library (e.g., Tornado).

BaseEventLoop.create_task(…) is only available in Python 3.4.2 or later. If you're using an older version of Python 3.3 or 3.4, you need to use asyncio.async(…), or install a more recent version of asyncio from PyPI.

Several asyncio functions accept coroutines and wrap them in asyncio.Task objects automatically, using asyncio.async internally. One example is BaseEventLoop.run_until_complete(…).

If you want to experiment with futures and coroutines on the Python console or in small tests, you can use the following snippet:[3]

```
>>> import asyncio
>>> def run_sync(coro_or_future):
...     loop = asyncio.get_event_loop()
...     return loop.run_until_complete(coro_or_future)
...
>>> a = run_sync(some_coroutine())
```

The relationship between coroutines, futures, and tasks is documented in section 18.5.3. Tasks and coroutines of the asyncio documentation, where you'll find this note:

> In this documentation, some methods are documented as coroutines, even if they are plain Python functions returning a Future. This is intentional to have a freedom of tweaking the implementation of these functions in the future.

3. Suggested by Petr Viktorin in a September 11, 2014, message to the Python-ideas list.

Having covered these fundamentals, we'll now study the code for the asynchronous flag download script *flags_asyncio.py* demonstrated along with the sequential and thread pool scripts in Example 17-1 (Chapter 17).

Downloading with asyncio and aiohttp

As of Python 3.4, `asyncio` only supports TCP and UDP directly. For HTTP or any other protocol, we need third-party packages; `aiohttp` is the one everyone seems to be using for `asyncio` HTTP clients and servers at this time.

Example 18-5 is the full listing for the flag downloading script *flags_asyncio.py*. Here is a high-level view of how it works:

1. We start the process in `download_many` by feeding the event loop with several coroutine objects produced by calling `download_one`.
2. The `asyncio` event loop activates each coroutine in turn.
3. When a client coroutine such as `get_flag` uses `yield from` to delegate to a library coroutine—such as `aiohttp.request`—control goes back to the event loop, which can execute another previously scheduled coroutine.
4. The event loop uses low-level APIs based on callbacks to get notified when a blocking operation is completed.
5. When that happens, the main loop sends a result to the suspended coroutine.
6. The coroutine then advances to the next yield, for example, `yield from resp.read()` in `get_flag`. The event loop takes charge again. Steps 4, 5, and 6 repeat until the event loop is terminated.

This is similar to the example we looked at in "The Taxi Fleet Simulation" on page 490, where a main loop started several taxi processes in turn. As each taxi process yielded, the main loop scheduled the next event for that taxi (to happen in the future), and proceeded to activate the next taxi in the queue. The taxi simulation is much simpler, and you can easily understand its main loop. But the general flow is the same as in `asyncio`: a single-threaded program where a main loop activates queued coroutines one by one. Each coroutine advances a few steps, then yields control back to the main loop, which then activates the next coroutine in the queue.

Now let's review Example 18-5 play by play.

Example 18-5. flags_asyncio.py: asynchronous download script with asyncio and aiohttp

```
import asyncio

import aiohttp    ❶
```

```
from flags import BASE_URL, save_flag, show, main  ❷


@asyncio.coroutine  ❸
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)  ❹
    image = yield from resp.read()  ❺
    return image


@asyncio.coroutine
def download_one(cc):  ❻
    image = yield from get_flag(cc)  ❼
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    loop = asyncio.get_event_loop()  ❽
    to_do = [download_one(cc) for cc in sorted(cc_list)]  ❾
    wait_coro = asyncio.wait(to_do)  ❿
    res, _ = loop.run_until_complete(wait_coro)  ⓫
    loop.close()  ⓬

    return len(res)


if __name__ == '__main__':
    main(download_many)
```

❶ `aiohttp` must be installed—it's not in the standard library.

❷ Reuse some functions from the `flags` module (Example 17-2).

❸ Coroutines should be decorated with `@asyncio.coroutine`.

❹ Blocking operations are implemented as coroutines, and your code delegates to them via `yield from` so they run asynchronously.

❺ Reading the response contents is a separate asynchronous operation.

❻ `download_one` must also be a coroutine, because it uses `yield from`.

❼ The only difference from the sequential implementation of `download_one` are the words `yield from` in this line; the rest of the function body is exactly as before.

❽ Get a reference to the underlying event-loop implementation.

❾ Build a list of generator objects by calling the `download_one` function once for each flag to be retrieved.

❿ Despite its name, `wait` is not a blocking function. It's a coroutine that completes when all the coroutines passed to it are done (that's the default behavior of `wait`; see explanation after this example).

⓫ Execute the event loop until `wait_coro` is done; this is where the script will block while the event loop runs. We ignore the second item returned by `run_until_complete`. The reason is explained next.

⓬ Shut down the event loop.

> It would be nice if event loop instances were context managers, so we could use a `with` block to make sure the loop is closed. However, the situation is complicated by the fact that client code never creates the event loop directly, but gets a reference to it by calling `asyncio.get_event_loop()`. Sometimes our code does not "own" the event loop, so it would be wrong to close it. For example, when using an external GUI event loop with a package like Quamash, the Qt library is responsible for shutting down the loop when the application quits.

The `asyncio.wait(…)` coroutine accepts an iterable of futures or coroutines; `wait` wraps each coroutine in a `Task`. The end result is that all objects managed by `wait` become instances of `Future`, one way or another. Because it is a coroutine function, calling `wait(…)` returns a coroutine/generator object; this is what the `wait_coro` variable holds. To drive the coroutine, we pass it to `loop.run_until_complete(…)`.

The `loop.run_until_complete` function accepts a future or a coroutine. If it gets a coroutine, `run_until_complete` wraps it into a `Task`, similar to what `wait` does. Coroutines, futures, and tasks can all be driven by `yield from`, and this is what `run_until_complete` does with the `wait_coro` object returned by the `wait` call. When `wait_coro` runs to completion, it returns a 2-tuple where the first item is the set of completed futures, and the second is the set of those not completed. In Example 18-5, the second set will always be empty—that's why we explicitly ignore it by assigning to `_`. But `wait` accepts two keyword-only arguments that may cause it to return even if some of the futures are not complete: `timeout` and `return_when`. See the `asyncio.wait` documentation for details.

Note that in Example 18-5 I could not reuse the `get_flag` function from *flags.py* (Example 17-2) because that uses the `requests` library, which performs blocking I/O. To leverage `asyncio`, we must replace every function that hits the network with an asynchronous version that is invoked with `yield from`, so that control is given back to the event loop. Using `yield from` in `get_flag` means that it must be driven as a coroutine.

That's why I could not reuse the `download_one` function from *flags_threadpool.py* (Example 17-3) either. The code in Example 18-5 drives `get_flag` with `yield_from`, so `download_one` is itself also a coroutine. For each request, a `download_one` coroutine object is created in `download_many`, and they are all driven by the `loop.run_until_complete` function, after being wrapped by the `asyncio.wait` coroutine.

There are a lot of new concepts to grasp in `asyncio` but the overall logic of Example 18-5 is easy to follow if you employ a trick suggested by Guido van Rossum himself: squint and pretend the `yield from` keywords are not there. If you do that, you'll notice that the code is as easy to read as plain old sequential code.

For example, imagine that the body of this coroutine…

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    image = yield from resp.read()
    return image
```

…works like the following function, except that it never blocks:

```
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url)
    image = resp.read()
    return image
```

Using the `yield from foo` syntax avoids blocking because the current coroutine is suspended (i.e., the delegating generator where the `yield from` code is), but the control flow goes back to the event loop, which can drive other coroutines. When the `foo` future or coroutine is done, it returns a result to the suspended coroutine, resuming it.

At the end of the section "Using yield from" on page 477, I stated two facts about every usage of `yield from`. Here they are, summarized:

- Every arrangement of coroutines chained with `yield from` must be ultimately driven by a caller that is not a coroutine, which invokes `next(…)` or `.send(…)` on the outermost delegating generator, explicitly or implicitly (e.g., in a `for` loop).
- The innermost subgenerator in the chain must be a simple generator that uses just yield—or an iterable object.

When using `yield from` with the `asyncio` API, both facts remain true, with the following specifics:

- The coroutine chains we write are always driven by passing our outermost delegating generator to an `asyncio` API call, such as `loop.run_until_complete(…)`.

In other words, when using `asyncio` our code doesn't drive a coroutine chain by calling `next(…)` or `.send(…)` on it—the `asyncio` event loop does that.

- The coroutine chains we write always end by delegating with `yield from` to some `asyncio` coroutine function or coroutine method (e.g., `yield from asyncio.sleep(…)` in Example 18-2) or coroutines from libraries that implement higher-level protocols (e.g., `resp = yield from aiohttp.request('GET', url)` in the `get_flag` coroutine of Example 18-5).

  In other words, the innermost subgenerator will be a library function that does the actual I/O, not something we write.

To summarize: as we use `asyncio`, our asynchronous code consists of coroutines that are delegating generators driven by `asyncio` itself and that ultimately delegate to `asyncio` library coroutines—possibly by way of some third-party library such as `aiohttp`. This arrangement creates pipelines where the `asyncio` event loop drives—through our coroutines—the library functions that perform the low-level asynchronous I/O.

We are now ready to answer one question raised in Chapter 17:

- How can *flags_asyncio.py* perform 5× faster than *flags.py* when both are single threaded?

Running Circling Around Blocking Calls

Ryan Dahl, the inventor of Node.js, introduces the philosophy of his project by saying "We're doing I/O completely wrong.[4]" He defines a *blocking function* as one that does disk or network I/O, and argues that we can't treat them as we treat nonblocking functions. To explain why, he presents the numbers in the first two columns of Table 18-1.

Table 18-1. Modern computer latency for reading data from different devices; third column shows proportional times in a scale easier to understand for us slow humans

Device	CPU cycles	Proportional "human" scale
L1 cache	3	3 seconds
L2 cache	14	14 seconds
RAM	250	250 seconds
disk	41,000,000	1.3 years
network	240,000,000	7.6 years

4. Video: Introduction to Node.js at 4:55.

To make sense of Table 18-1, bear in mind that modern CPUs with GHz clocks run billions of cycles per second. Let's say that a CPU runs exactly 1 billion cycles per second. That CPU can make 333,333,333 L1 cache reads in one second, or 4 (four!) network reads in the same time. The third column of Table 18-1 puts those numbers in perspective by multiplying the second column by a constant factor. So, in an alternate universe, if one read from L1 cache took 3 seconds, then a network read would take 7.6 years!

There are two ways to prevent blocking calls to halt the progress of the entire application:

- Run each blocking operation in a separate thread.
- Turn every blocking operation into a nonblocking asynchronous call.

Threads work fine, but the memory overhead for each OS thread—the kind that Python uses—is on the order of megabytes, depending on the OS. We can't afford one thread per connection if we are handling thousands of connections.

Callbacks are the traditional way to implement asynchronous calls with low memory overhead. They are a low-level concept, similar to the oldest and most primitive concurrency mechanism of all: hardware interrupts. Instead of waiting for a response, we register a function to be called when something happens. In this way, every call we make can be nonblocking. Ryan Dahl advocates callbacks for their simplicity and low overhead.

Of course, we can only make callbacks work because the event loop underlying our asynchronous applications can rely on infrastructure that uses interrupts, threads, polling, background processes, etc. to ensure that multiple concurrent requests make progress and they eventually get done.[5] When the event loop gets a response, it calls back our code. But the single main thread shared by the event loop and our application code is never blocked—if we don't make mistakes.

When used as coroutines, generators provide an alternative way to do asynchronous programming. From the perspective of the event loop, invoking a callback or calling `.send()` on a suspended coroutine is pretty much the same. There is a memory overhead for each suspended coroutine, but it's orders of magnitude smaller than the overhead for each thread. And they avoid the dreaded "callback hell," which we'll discuss in "From Callbacks to Futures and Coroutines" on page 562.

Now the five-fold performance advantage of *flags_asyncio.py* over *flags.py* should make sense: *flags.py* spends billions of CPU cycles waiting for each download, one after the other. The CPU is actually doing a lot meanwhile, just not running your program. In contrast, when `loop_until_complete` is called in the `download_many` function of

5. In fact, although Node.js does not support user-level threads written in JavaScript, behind the scenes it implements a thread pool in C with the `libeio` library, to provide its callback-based file APIs—because as of 2014 there are no stable and portable asynchronous file handling APIs for most OSes.

flags_asyncio.py, the event loop drives each `download_one` coroutine to the first `yield from`, and this in turn drives each `get_flag` coroutine to the first `yield from`, calling `aiohttp.request(…)`. None of these calls are blocking, so all requests are started in a fraction of a second.

As the `asyncio` infrastructure gets the first response back, the event loop sends it to the waiting `get_flag` coroutine. As `get_flag` gets a response, it advances to the next `yield from`, which calls `resp.read()` and yields control back to the main loop. Other responses arrive in close succession (because they were made almost at the same time). As each `get_flag` returns, the delegating generator `download_flag` resumes and saves the image file.

For maximum performance, the `save_flag` operation should be asynchronous, but `asyncio` does not provide an asynchronous filesystem API at this time—as Node does. If that becomes a bottleneck in your application, you can use the `loop.run_in_executor` function to run `save_flag` in a thread pool. Example 18-9 will show how.

Because the asynchronous operations are interleaved, the total time needed to download many images concurrently is much less than doing it sequentially. When making 600 HTTP requests with `asyncio` I got all results back more than 70 times faster than with a sequential script.

Now let's go back to the HTTP client example to see how we can display an animated progress bar and perform proper error handling.

Enhancing the asyncio downloader Script

Recall from "Downloads with Progress Display and Error Handling" on page 520 that the `flags2` set of examples share the same command-line interface. This includes the *flags2_asyncio.py* we will analyze in this section. For instance, Example 18-6 shows how to get 100 flags (`-al 100`) from the `ERROR` server, using 100 concurrent requests (`-m 100`).

Example 18-6. Running flags2_asyncio.py

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

Act Responsibly When Testing Concurrent Clients

Even if the overall download time is not different between the threaded and asyncio HTTP clients, asyncio can send requests faster, so it's even more likely that the server will suspect a DOS attack. To really exercise these concurrent clients at full speed, set up a local HTTP server for testing, as explained in the *README.rst* inside the *17-futures/countries/* directory of the *Fluent Python* code repository.

Now let's see how *flags2_asyncio.py* is implemented.

Using asyncio.as_completed

In Example 18-5, I passed a list of coroutines to `asyncio.wait`, which—when driven by `loop.run_until.complete`—would return the results of the downloads when all were done. But to update a progress bar we need to get results as they are done. Fortunately, there is an `asyncio` equivalent of the `as_completed` generator function we used in the thread pool example with the progress bar (Example 17-14).

Writing a `flags2` example to leverage `asyncio` entails rewriting several functions that the `concurrent.future` version could reuse. That's because there's only one main thread in an `asyncio` program and we can't afford to have blocking calls in that thread, as it's the same thread that runs the event loop. So I had to rewrite `get_flag` to use `yield from` for all network access. Now `get_flag` is a coroutine, so `download_one` must drive it with `yield from`, therefore `download_one` itself becomes a coroutine. Previously, in Example 18-5, `download_one` was driven by `download_many`: the calls to `download_one` were wrapped in an `asyncio.wait` call and passed to `loop.run_until_complete`. Now we need finer control for progress reporting and error handling, so I moved most of the logic from `download_many` into a new `downloader_coro` coroutine, and use `download_many` just to set up the event loop and schedule `downloader_coro`.

Example 18-7 shows the top of the *flags2_asyncio.py* script where the `get_flag` and `download_one` coroutines are defined. Example 18-8 lists the rest of the source, with `downloader_coro` and `download_many`.

Example 18-7. flags2_asyncio.py: Top portion of the script; remaining code is in Example 18-8

```
import asyncio
import collections

import aiohttp
from aiohttp import web
import tqdm
```

```
from flags2_common import main, HTTPStatus, Result, save_flag

# default set low to avoid errors from remote site, such as
# 503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000


class FetchError(Exception):  ❶
    def __init__(self, country_code):
        self.country_code = country_code


@asyncio.coroutine
def get_flag(base_url, cc):  ❷
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    if resp.status == 200:
        image = yield from resp.read()
        return image
    elif resp.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.HttpProcessingError(
            code=resp.status, message=resp.reason,
            headers=resp.headers)


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):  ❸
    try:
        with (yield from semaphore):  ❹
            image = yield from get_flag(base_url, cc)  ❺
    except web.HTTPNotFound:  ❻
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc  ❼
    else:
        save_flag(image, cc.lower() + '.gif')  ❽
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ This custom exception will be used to wrap other HTTP or network exceptions and carry the `country_code` for error reporting.

❷ get_flag will either return the bytes of the image downloaded, raise web.HTTPNotFound if the HTTP response status is 404, or raise an aiohttp.HttpProcessingError for other HTTP status codes.

❸ The semaphore argument is an instance of asyncio.Semaphore, a synchronization device that limits the number of concurrent requests.

❹ A semaphore is used as a context manager in a yield from expression so that the system as whole is not blocked: only this coroutine is blocked while the semaphore counter is at the maximum allowed number.

❺ When this with statement exits, the semaphore counter is decremented, unblocking some other coroutine instance that may be waiting for the same semaphore object.

❻ If the flag was not found, just set the status for the Result accordingly.

❼ Any other exception will be reported as a FetchError with the country code and the original exception chained using the raise X from Y syntax introduced in PEP 3134 — Exception Chaining and Embedded Tracebacks.

❽ This function call actually saves the flag image to disk.

In Example 18-7, you can see that the code for get_flag and download_one changed significantly from the sequential version because these functions are now coroutines using yield from to make asynchronous calls.

Network client code of the sort we are studying should always use some throttling mechanism to avoid pounding the server with too many concurrent requests—the overall performance of the system may degrade if the server is overloaded. In *flags2_threadpool.py* (Example 17-14), the throttling was done by instantiating the ThreadPoolExecutor with the required max_workers argument set to concur_req in the download_many function, so only concur_req threads are started in the pool. In *flags2_asyncio.py*, I used an asyncio.Semaphore, which is created by the download er_coro function (shown next, in Example 18-8) and is passed as the semaphore argument to download_one in Example 18-7.[6]

A Semaphore is an object that holds an internal counter that is decremented whenever we call the .acquire() coroutine method on it, and incremented when we call the .release() coroutine method. The initial value of the counter is set when the Semaphore is instantiated, as in this line of downloader_coro:

```
semaphore = asyncio.Semaphore(concur_req)
```

6. Thanks to Guto Maia who noted that Semaphore was not explained in the book draft.

Calling .acquire() does not block when the counter is greater than zero, but if the counter is zero, .acquire() will block the calling coroutine until some other coroutine calls .release() on the same Semaphore, thus incrementing the counter. In Example 18-7, I don't call .acquire() or .release(), but use the semaphore as a context manager in this block of code inside download_one:

```
        with (yield from semaphore):
            image = yield from get_flag(base_url, cc)
```

That snippet guarantees that no more than concur_req instances of get_flags coroutines will be started at any time.

Now let's take a look at the rest of the script in Example 18-8. Note that most functionality of the old download_many function is now in a coroutine, downloader_coro. This was necessary because we must use yield from to retrieve the results of the futures yielded by asyncio.as_completed, therefore as_completed must be invoked in a coroutine. However, I couldn't simply turn download_many into a coroutine, because I must pass it to the main function from flags2_common in the last line of the script, and that main function is not expecting a coroutine, just a plain function. Therefore I created downloader_coro to run the as_completed loop, and now download_many simply sets up the event loop and schedules downloader_coro by passing it to loop.run_until_complete.

Example 18-8. flags2_asyncio.py: Script continued from Example 18-7

```
@asyncio.coroutine
def downloader_coro(cc_list, base_url, verbose, concur_req):   ❶
    counter = collections.Counter()
    semaphore = asyncio.Semaphore(concur_req)  ❷
    to_do = [download_one(cc, base_url, semaphore, verbose)
             for cc in sorted(cc_list)]  ❸

    to_do_iter = asyncio.as_completed(to_do)  ❹
    if not verbose:
        to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list))  ❺
    for future in to_do_iter:  ❻
        try:
            res = yield from future  ❼
        except FetchError as exc:  ❽
            country_code = exc.country_code  ❾
            try:
                error_msg = exc.__cause__.args[0]  ❿
            except IndexError:
                error_msg = exc.__cause__.__class__.__name__  ⓫
            if verbose and error_msg:
                msg = '*** Error for {}: {}'
                print(msg.format(country_code, error_msg))
            status = HTTPStatus.error
        else:
                status = res.status

            counter[status] += 1  ⓬

        return counter  ⓭


def download_many(cc_list, base_url, verbose, concur_req):
    loop = asyncio.get_event_loop()
    coro = downloader_coro(cc_list, base_url, verbose, concur_req)
    counts = loop.run_until_complete(coro)  ⓮
    loop.close()  ⓯

    return counts


if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ The coroutine receives the same arguments as `download_many`, but it cannot be invoked directly from `main` precisely because it's a coroutine function and not a plain function like `download_many`.

❷ Create an `asyncio.Semaphore` that will allow up to `concur_req` active coroutines among those using this semaphore.

❸ Create a list of coroutine objects, one per call to the `download_one` coroutine.

❹ Get an iterator that will return futures as they are done.

❺ Wrap the iterator in the `tqdm` function to display progress.

❻ Iterate over the completed futures; this loop is very similar to the one in `down load_many` in Example 17-14; most changes have to do with exception handling because of differences in the HTTP libraries (`requests` versus `aiohttp`).

❼ The easiest way to retrieve the result of an `asyncio.Future` is using `yield from` instead of calling `future.result()`.

❽ Every exception in `download_one` is wrapped in a `FetchError` with the original exception chained.

❾ Get the country code where the error occurred from the `FetchError` exception.

❿ Try to retrieve the error message from the original exception (`__cause__`).

⓫ If the error message cannot be found in the original exception, use the name of the chained exception class as the error message.

⓬ Tally outcomes.

⓭ Return the counter, as done in the other scripts.

⓮ `download_many` simply instantiates the coroutine and passes it to the event loop with `run_until_complete`.

⓯ When all work is done, shut down the event loop and return `counts`.

In Example 18-8, we could not use the mapping of futures to country codes we saw in Example 17-14 because the futures returned by `asyncio.as_completed` are not necessarily the same futures we pass into the `as_completed` call. Internally, the `asyncio` machinery replaces the future objects we provide with others that will, in the end, produce the same results.[7]

Because I could not use the futures as keys to retrieve the country code from a `dict` in case of failure, I implemented the custom `FetchError` exception (shown in Example 18-7). `FetchError` wraps a network exception and holds the country code associated with it, so the country code can be reported with the error in verbose mode. If there is no error, the country code is available as the result of the `yield from future` expression at the top of the `for` loop.

This wraps up the discussion of an `asyncio` example functionally equivalent to the *flags2_threadpool.py* we saw earlier. Next, we'll implement enhancements to *flags2_asyncio.py* that will let us explore `asyncio` further.

While discussing Example 18-7, I noted that `save_flag` performs disk I/O and should be executed asynchronously. The following section shows how.

Using an Executor to Avoid Blocking the Event Loop

In the Python community, we tend to overlook the fact that local filesystem access is blocking, rationalizing that it doesn't suffer from the higher latency of network access (which is also dangerously unpredictable). In contrast, Node.js programmers are constantly reminded that all filesystem functions are blocking because their signatures require a callback. Recall from Table 18-1 that blocking for disk I/O wastes millions of CPU cycles, and this may have a significant impact on the performance of the application.

In Example 18-7, the blocking function is `save_flag`. In the threaded version of the script (Example 17-14), `save_flag` blocks the thread that's running the `download_one` function, but that's only one of several worker threads. Behind the scenes, the blocking I/O call releases the GIL, so another thread can proceed. But in *flags2_asyncio.py*, `save_flag` blocks the single thread our code shares with the `asyncio` event loop, there-

7. A detailed discussion about this can be found in a thread I started in the python-tulip group, titled "Which other futures my come out of asyncio.as_completed?". Guido responds, and gives insight on the implementation of `as_completed` as well as the close relationship between futures and coroutines in `asyncio`.

fore the whole application freezes while the file is being saved. The solution to this problem is the `run_in_executor` method of the event loop object.

Behind the scenes, the `asyncio` event loop has a thread pool executor, and you can send callables to be executed by it with `run_in_executor`. To use this feature in our example, only a few lines need to change in the `download_one` coroutine, as shown in Example 18-9.

Example 18-9. flags2_asyncio_executor.py: Using the default thread pool executor to run save_flag

```
@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):
            image = yield from get_flag(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        loop = asyncio.get_event_loop()  ❶
        loop.run_in_executor(None,  ❷
                save_flag, image, cc.lower() + '.gif')  ❸
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ Get a reference to the event loop object.

❷ The first argument to `run_in_executor` is an executor instance; if `None`, the default thread pool executor of the event loop is used.

❸ The remaining arguments are the callable and its positional arguments.

When I tested Example 18-9, there was no noticeable change in performance for using `run_in_executor` to save the image files because they are not large (13 KB each, on average). But you'll see an effect if you edit the `save_flag` function in *flags2_common.py* to save 10 times as many bytes on each file—just by coding `fp.write(img*10)` instead of `fp.write(img)`. With an average download size of 130 KB, the advantage of using `run_in_executor` becomes clear. If you're downloading megapixel images, the speedup will be significant.

The advantage of coroutines over callbacks becomes evident when we need to coordinate asynchronous requests, and not just make completely independent requests. The next section explains the problem and the solution.

From Callbacks to Futures and Coroutines

Event-oriented programming with coroutines requires some effort to master, so it's good to be clear on how it improves on the classic callback style. This is the theme of this section.

Anyone with some experience in callback-style event-oriented programming knows the term "callback hell": the nesting of callbacks when one operation depends on the result of the previous operation. If you have three asynchronous calls that must happen in succession, you need to code callbacks nested three levels deep. Example 18-10 is an example in JavaScript.

Example 18-10. Callback hell in JavaScript: nested anonymous functions, a.k.a. Pyramid of Doom

```
api_call1(request1, function (response1) {
    // stage 1
    var request2 = step1(response1);

    api_call2(request2, function (response2) {
        // stage 2
        var request3 = step2(response2);

        api_call3(request3, function (response3) {
            // stage 3
            step3(response3);
        });
    });
});
```

In Example 18-10, `api_call1`, `api_call2`, and `api_call3` are library functions your code uses to retrieve results asynchronously—perhaps `api_call1` goes to a database and `api_call2` gets data from a web service, for example. Each of these take a callback function, which in JavaScript are often anonymous functions (they are named `stage1`, `stage2`, and `stage3` in the following Python example). The `step1`, `step2`, and `step3` here represent regular functions of your application that process the responses received by the callbacks.

Example 18-11 shows what callback hell looks like in Python.

Example 18-11. Callback hell in Python: chained callbacks

```
def stage1(response1):
    request2 = step1(response1)
    api_call2(request2, stage2)
```

```
def stage2(response2):
    request3 = step2(response2)
    api_call3(request3, stage3)


def stage3(response3):
    step3(response3)


api_call1(request1, stage1)
```

Although the code in Example 18-11 is arranged very differently from Example 18-10, they do exactly the same thing, and the JavaScript example could be written using the same arrangement (but the Python code can't be written in the JavaScript style because of the syntactic limitations of `lambda`).

Code organized as Example 18-10 or Example 18-11 is hard to read, but it's even harder to write: each function does part of the job, sets up the next callback, and returns, to let the event loop proceed. At this point, all local context is lost. When the next callback (e.g., `stage2`) is executed, you don't have the value of `request2` any more. If you need it, you must rely on closures or external data structures to store it between the different stages of the processing.

That's where coroutines really help. Within a coroutine, to perform three asynchronous actions in succession, you `yield` three times to let the event loop continue running. When a result is ready, the coroutine is activated with a `.send()` call. From the perspective of the event loop, that's similar to invoking a callback. But for the users of a coroutine-style asynchronous API, the situation is vastly improved: the entire sequence of three operations is in one function body, like plain old sequential code with local variables to retain the context of the overall task under way. See Example 18-12.

Example 18-12. Coroutines and yield from enable asynchronous programming without callbacks

```
@asyncio.coroutine
def three_stages(request1):
    response1 = yield from api_call1(request1)
    # stage 1
    request2 = step1(response1)
    response2 = yield from api_call2(request2)
    # stage 2
    request3 = step2(response2)
    response3 = yield from api_call3(request3)
    # stage 3
    step3(response3)
```

```
loop.create_task(three_stages(request1))  # must explicitly schedule execution
```

Example 18-12 is much easier to follow the previous JavaScript and Python examples: the three stages of the operation appear one after the other inside the same function. This makes it trivial to use previous results in follow-up processing. It also provides a context for error reporting through exceptions.

Suppose in Example 18-11 the processing of the call `api_call2(request2, stage2)` raises an I/O exception (that's the last line of the `stage1` function). The exception cannot be caught in `stage1` because `api_call2` is an asynchronous call: it returns immediately, before any I/O is performed. In callback-based APIs, this is solved by registering two callbacks for each asynchronous call: one for handling the result of successful operations, another for handling errors. Work conditions in callback hell quickly deteriorate when error handling is involved.

In contrast, in Example 18-12, all the asynchronous calls for this three-stage operation are inside the same function, `three_stages`, and if the asynchronous calls `api_call1`, `api_call2`, and `api_call3` raise exceptions we can handle them by putting the respective `yield from` lines inside `try/except` blocks.

This is a much better place than callback hell, but I wouldn't call it coroutine heaven because there is a price to pay. Instead of regular functions, you must use coroutines and get used to `yield from`, so that's the first obstacle. Once you write `yield from` in a function, it's now a coroutine and you can't simply call it, like we called `api_call1(request1, stage1)` in Example 18-11 to start the callback chain. You must explicitly schedule the execution of the coroutine with the event loop, or activate it using `yield from` in another coroutine that is scheduled for execution. Without the call `loop.create_task(three_stages(request1))` in the last line, nothing would happen in Example 18-12.

The next example puts this theory into practice.

Doing Multiple Requests for Each Download

Suppose you want to save each country flag with the name of the country and the country code, instead of just the country code. Now you need to make two HTTP requests per flag: one to get the flag image itself, the other to get the *metadata.json* file in the same directory as the image: that's where the name of the country is recorded.

Articulating multiple requests in the same task is easy in the threaded script: just make one request then the other, blocking the thread twice, and keeping both pieces of data (country code and name) in local variables, ready to use when saving the files. If you need to do the same in an asynchronous script with callbacks, you start to smell the sulfur of callback hell: the country code and name will need to be passed around in a closure or held somewhere until you can save the file because each callback runs in a different local context. Coroutines and `yield from` provide relief from that. The solution is not as simple as with threads, but more manageable than chained or nested callbacks.

Example 18-13 shows code from the third variation of the `asyncio` flag downloading script, using the country name to save each flag. The `download_many` and `downloader_coro` are unchanged from *flags2_asyncio.py* (Examples 18-7 and 18-8). The changes are:

`download_one`
: This coroutine now uses `yield from` to delegate to `get_flag` and the new `get_country` coroutine.

`get_flag`
: Most code from this coroutine was moved to a new `http_get` coroutine so it can also be used by `get_country`.

`get_country`
: This coroutine fetches the *metadata.json* file for the country code, and gets the name of the country from it.

`http_get`
: Common code for getting a file from the Web.

Example 18-13. flags3_asyncio.py: more coroutine delegation to perform two requests per flag

```
@asyncio.coroutine
def http_get(url):
    res = yield from aiohttp.request('GET', url)
    if res.status == 200:
        ctype = res.headers.get('Content-type', '').lower()
        if 'json' in ctype or url.endswith('json'):
            data = yield from res.json()   ❶
        else:
            data = yield from res.read()   ❷
        return data

    elif res.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.errors.HttpProcessingError(
            code=res.status, message=res.reason,
            headers=res.headers)


@asyncio.coroutine
def get_country(base_url, cc):
    url = '{}/{cc}/metadata.json'.format(base_url, cc=cc.lower())
    metadata = yield from http_get(url)   ❸
    return metadata['country']


@asyncio.coroutine
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    return (yield from http_get(url))  ❹


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):  ❺
            image = yield from get_flag(base_url, cc)
        with (yield from semaphore):
            country = yield from get_country(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        country = country.replace(' ', '_')
        filename = '{}-{}.gif'.format(country, cc)
        loop = asyncio.get_event_loop()
        loop.run_in_executor(None, save_flag, image, filename)
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ If the content type has `'json'` in it or the `url` ends with `.json`, use the response `.json()` method to parse it and return a Python data structure—in this case, a `dict`.

❷ Otherwise, use `.read()` to fetch the bytes as they are.

❸ `metadata` will receive a Python `dict` built from the JSON contents.

❹ The outer parentheses here are required because the Python parser gets confused and produces a syntax error when it sees the keywords `return yield from` lined up like that.

❺ I put the calls to `get_flag` and `get_country` in separate `with` blocks controlled by the `semaphore` because I want to keep it acquired for the shortest possible time.

The `yield from` syntax appears nine times in Example 18-13. By now you should be getting the hang of how this construct is used to delegate from one coroutine to another without blocking the event loop.

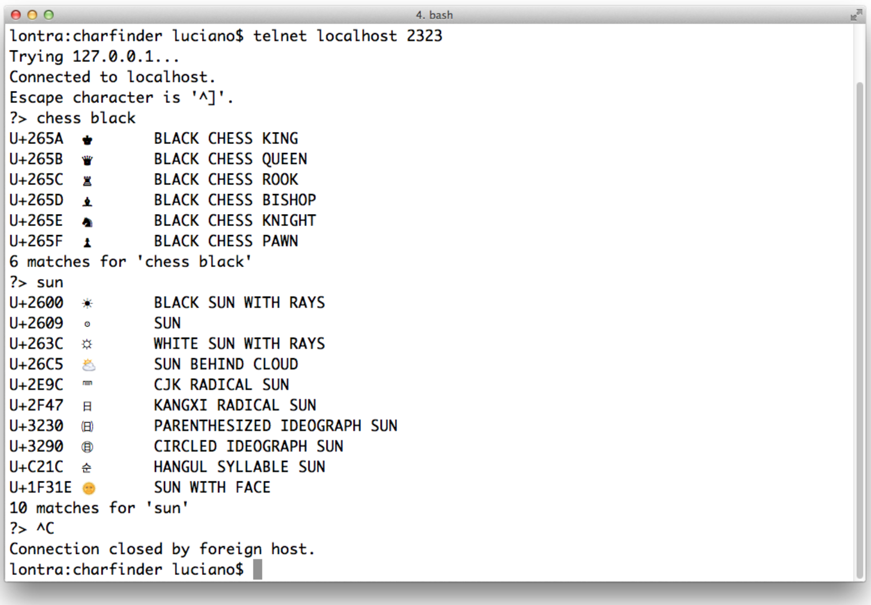
The challenge is to know when you have to use `yield from` and when you can't use it. The answer in principle is easy, you `yield from` coroutines and `asyncio.Future` instances—including tasks. But some APIs are tricky, mixing coroutines and plain functions in seemingly arbitrary ways, like the `StreamWriter` class we'll use in one of the servers in the next section.

Example 18-13 wraps up the `flags2` set of examples. I encourage you to play with them to develop an intuition of how concurrent HTTP clients perform. Use the `-a`, `-e`, and `-l` command-line options to control the number of downloads, and the `-m` option to set the number of concurrent downloads. Run tests against the `LOCAL`, `REMOTE`, `DELAY`, and `ERROR` servers. Discover the optimum number of concurrent downloads to maximize throughput against each server. Tweak the settings of the *vaurien_error_delay.sh* script to add or remove errors and delays.

We'll now go from client scripts to writing servers with `asyncio`.

Writing asyncio Servers

The classic toy example of a TCP server is an echo server. We'll build slightly more interesting toys: Unicode character finders, first using plain TCP, then using HTTP. These servers will allow clients to query for Unicode characters based on words in their canonical names, using the `unicodedata` module we discussed in "The Unicode Database" on page 127. A Telnet session with the TCP character finder server, searching for chess pieces and characters with the word "sun" is shown in Figure 18-2.



```
lontra:charfinder luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> chess black
U+265A  ♚       BLACK CHESS KING
U+265B  ♛       BLACK CHESS QUEEN
U+265C  ♜       BLACK CHESS ROOK
U+265D  ♝       BLACK CHESS BISHOP
U+265E  ♞       BLACK CHESS KNIGHT
U+265F  ♟       BLACK CHESS PAWN
6 matches for 'chess black'
?> sun
U+2600  ☀       BLACK SUN WITH RAYS
U+2609  ☉       SUN
U+263C  ☼       WHITE SUN WITH RAYS
U+26C5  ⛅       SUN BEHIND CLOUD
U+2E9C  ⺜       CJK RADICAL SUN
U+2F47  ⽇       KANGXI RADICAL SUN
U+3230  ㈰       PARENTHESIZED IDEOGRAPH SUN
U+3290  ㊐       CIRCLED IDEOGRAPH SUN
U+C21C  순       HANGUL SYLLABLE SUN
U+1F31E 🌞      SUN WITH FACE
10 matches for 'sun'
?> ^C
Connection closed by foreign host.
lontra:charfinder luciano$ 
```

Figure 18-2. A Telnet session with the tcp_charfinder.py server: querying for "chess black" and "sun".

Now, on to the implementations.

An asyncio TCP Server

Most of the logic in these examples is in the *charfinder.py* module, which has nothing concurrent about it. You can use *charfinder.py* as a command-line character finder, but more importantly, it was designed to provide content for our `asyncio` servers. The code for *charfinder.py* is in the *Fluent Python* code repository.

The `charfinder` module indexes each word that appears in character names in the Unicode database bundled with Python, and creates an inverted index stored in a `dict`. For example, the inverted index entry for the key `'SUN'` contains a `set` with the 10 Unicode characters that have that word in their names. The inverted index is saved in a local *charfinder_index.pickle* file. If multiple words appear in the query, `charfinder` computes the intersection of the sets retrieved from the index.

We'll now focus on the *tcp_charfinder.py* script that is answering the queries in Figure 18-2. Because I have a lot to say about this code, I've split it into two parts: Example 18-14 and Example 18-15.

Example 18-14. tcp_charfinder.py: a simple TCP server using asyncio.start_server; code for this module continues in Example 18-15

```
import sys
import asyncio

from charfinder import UnicodeNameIndex    ❶

CRLF = b'\r\n'
PROMPT = b'?> '

index = UnicodeNameIndex()   ❷

@asyncio.coroutine
def handle_queries(reader, writer):   ❸
    while True:   ❹
        writer.write(PROMPT)  # can't yield from!   ❺
        yield from writer.drain()  # must yield from!   ❻
        data = yield from reader.readline()   ❼
        try:
            query = data.decode().strip()
        except UnicodeDecodeError:   ❽
            query = '\x00'
        client = writer.get_extra_info('peername')   ❾
        print('Received from {}: {!r}'.format(client, query))   ❿
        if query:
            if ord(query[:1]) < 32:   ⓫
                break
            lines = list(index.find_description_strs(query))  ⓬
            if lines:
                writer.writelines(line.encode() + CRLF for line in lines)  ⓭
            writer.write(index.status(query, len(lines)).encode() + CRLF)  ⓮

            yield from writer.drain()   ⓯
            print('Sent {} results'.format(len(lines)))   ⓰

    print('Close the client socket')   ⓱
    writer.close()   ⓲
```

❶ `UnicodeNameIndex` is the class that builds the index of names and provides querying methods.

❷ When instantiated, `UnicodeNameIndex` uses `charfinder_index.pickle`, if available, or builds it, so the first run may take a few seconds longer to start.[8]

❸ This is the coroutine we need to pass to `asyncio_startserver`; the arguments received are an `asyncio.StreamReader` and an `asyncio.StreamWriter`.

❹ This loop handles a session that lasts until any control character is received from the client.

❺ The `StreamWriter.write` method is not a coroutine, just a plain function; this line sends the `?>` prompt.

❻ `StreamWriter.drain` flushes the `writer` buffer; it is a coroutine, so it must be called with `yield from`.

❼ `StreamWriter.readline` is a coroutine; it returns `bytes`.

❽ A `UnicodeDecodeError` may happen when the Telnet client sends control characters; if that happens, we pretend a null character was sent, for simplicity.

❾ This returns the remote address to which the socket is connected.

❿ Log the query to the server console.

⓫ Exit the loop if a control or null character was received.

⓬ This returns a generator that yields strings with the Unicode codepoint, the actual character and its name (e.g., `U+0039\t9\tDIGIT NINE`); for simplicity, I build a `list` from it.

⓭ Send the `lines` converted to `bytes` using the default `UTF-8` encoding, appending a carriage return and a line feed to each; note that the argument is a generator expression.

⓮ Write a status line such as `627 matches for 'digit'`.

⓯ Flush the output buffer.

⓰ Log the response to the server console.

⓱ Log the end of the session to the server console.

⓲ Close the `StreamWriter`.

The `handle_queries` coroutine has a plural name because it starts an interactive session and handles multiple queries from each client.

8. Leonardo Rochael pointed out that building the `UnicodeNameIndex` could be delegated to another thread using `loop.run_with_executor()` in the `main` function of Example 18-15, so the server would be ready to take requests immediately while the index is built. That is true, but querying the index is the only thing this app does, so that would not be a big win. It's an interesting exercise to do as Leo suggests, though. Go ahead and do it, if you like.

Note that all I/O in Example 18-14 is in `bytes`. We need to decode the strings received from the network, and encode strings sent out. In Python 3, the default encoding is UTF-8, and that's what we are using implicitly.

One caveat is that some of the I/O methods are coroutines and must be driven with `yield from`, while others are simple functions. For example, `StreamWriter.write` is a plain function, on the assumption that most of the time it does not block because it writes to a buffer. On the other hand, `StreamWriter.drain`, which flushes the buffer and performs the actual I/O is a coroutine, as is `Streamreader.readline`. While I was writing this book, a major improvement to the `asyncio` API docs was the clear labeling of coroutines as such.

Example 18-15 lists the main function for the module started in Example 18-14.

Example 18-15. tcp_charfinder.py (continued from Example 18-14): the main function sets up and tears down the event loop and the socket server

```
def main(address='127.0.0.1', port=2323):  ❶
    port = int(port)
    loop = asyncio.get_event_loop()
    server_coro = asyncio.start_server(handle_queries, address, port,
                                loop=loop)  ❷
    server = loop.run_until_complete(server_coro)  ❸

    host = server.sockets[0].getsockname()  ❹
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))  ❺
    try:
        loop.run_forever()  ❻
    except KeyboardInterrupt:  # CTRL+C pressed
        pass

    print('Server shutting down.')
    server.close()  ❼
    loop.run_until_complete(server.wait_closed())  ❽
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:])  ❿
```

❶ The `main` function can be called with no arguments.

❷ When completed, the coroutine object returned by `asyncio.start_server` returns an instance of `asyncio.Server`, a TCP socket server.

❸ Drive `server_coro` to bring up the `server`.

❹ Get address and port of the first socket of the server and…

❺ …display it on the server console. This is the first output generated by this script on the server console.

❻ Run the event loop; this is where `main` will block until killed when `CTRL-C` is pressed on the server console.

❼ Close the server.

❽ `server.wait_closed()` returns a future; use `loop.run_until_complete` to let the future do its job.

❾ Terminate the event loop.

❿ This is a shortcut for handling optional command-line arguments: explode `sys.argv[1:]` and pass it to a `main` function with suitable default arguments.

Note how `run_until_complete` accepts either a coroutine (the result of `start_server`) or a `Future` (the result of `server.wait_closed`). If `run_until_complete` gets a coroutine as argument, it wraps the coroutine in a `Task`.

You may find it easier to understand how control flows in *tcp_charfinder.py* if you take a close look at the output it generates on the server console, listed in Example 18-16.

Example 18-16. tcp_charfinder.py: this is the server side of the session depicted in Figure 18-2

```
$ python3 tcp_charfinder.py
Serving on ('127.0.0.1', 2323). Hit CTRL-C to stop.  ❶
Received from ('127.0.0.1', 62910): 'chess black'  ❷
Sent 6 results
Received from ('127.0.0.1', 62910): 'sun'  ❸
Sent 10 results
Received from ('127.0.0.1', 62910): '\x00'   ❹
Close the client socket  ❺
```

❶ This is output by `main`.

❷ First iteration of the `while` loop in `handle_queries`.

❸ Second iteration of the `while` loop.

❹ The user hit `CTRL-C`; the server receives a control character and closes the session.

❺ The client socket is closed but the server is still running, ready to service another client.

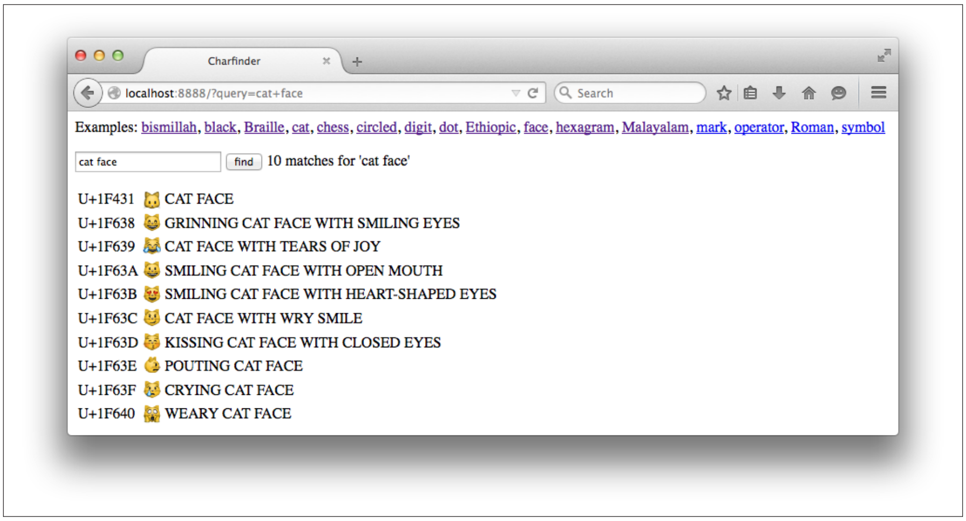
Note how `main` almost immediately displays the `Serving on...` message and blocks in the `loop.run_forever()` call. At that point, control flows into the event loop and stays there, occasionally coming back to the `handle_queries` coroutine, which yields control back to the event loop whenever it needs to wait for the network as it sends or receives data. While the event loop is alive, a new instance of the `handle_queries` coroutine will be started for each client that connects to the server. In this way, multiple clients can be handled concurrently by this simple server. This continues until a `KeyboardInterrupt` occurs or the process is killed by the OS.

The *tcp_charfinder.py* code leverages the high-level `asyncio` Streams API that provides a ready-to-use server so you only need to implement a handler function, which can be a plain callback or a coroutine. There is also a lower-level Transports and Protocols API, inspired by the transport and protocols abstractions in the Twisted framework. Refer to the `asyncio` Transports and Protocols documentation for more information, including a TCP echo server implemented with that lower-level API.

The next section presents an HTTP character finder server.

An aiohttp Web Server

The `aiohttp` library we used for the `asyncio` flags examples also supports server-side HTTP, so that's what I used to implement the *http_charfinder.py* script. Figure 18-3 shows the simple web interface of the server, displaying the result of a search for a "cat face" emoji.



Examples: bismillah, black, Braille, cat, chess, circled, digit, dot, Ethiopic, face, hexagram, Malayalam, mark, operator, Roman, symbol

cat face 10 matches for 'cat face'

```
U+1F431 🐱 CAT FACE
U+1F638 😸 GRINNING CAT FACE WITH SMILING EYES
U+1F639 😹 CAT FACE WITH TEARS OF JOY
U+1F63A 😺 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😻 SMILING CAT FACE WITH HEART-SHAPED EYES
U+1F63C 😼 CAT FACE WITH WRY SMILE
U+1F63D 😽 KISSING CAT FACE WITH CLOSED EYES
U+1F63E 😾 POUTING CAT FACE
U+1F63F 😿 CRYING CAT FACE
U+1F640 🙀 WEARY CAT FACE
```

Figure 18-3. Browser window displaying search results for "cat face" on the http_charfinder.py server

Some browsers are better than others at displaying Unicode. The screenshot in Figure 18-3 was captured with Firefox on OS X, and I got the same result with Safari. But up-to-date Chrome and Opera browsers on the same machine did not display emoji characters like the cat faces. Other search results (e.g., "chess") looked fine, so it's likely a font issue on Chrome and Opera on OSX.

We'll start by analyzing the most interesting part of *http_charfinder.py*: the bottom half where the event loop and the HTTP server is set up and torn down. See Example 18-17.

Example 18-17. http_charfinder.py: the main and init functions

```
@asyncio.coroutine
def init(loop, address, port):  ❶
    app = web.Application(loop=loop)  ❷
    app.router.add_route('GET', '/', home)  ❸
    handler = app.make_handler()  ❹
    server = yield from loop.create_server(handler,
                                           address, port)  ❺
    return server.sockets[0].getsockname()  ❻

def main(address="127.0.0.1", port=8888):
    port = int(port)
    loop = asyncio.get_event_loop()
    host = loop.run_until_complete(init(loop, address, port))  ❼
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))
    try:
        loop.run_forever()  ❽
    except KeyboardInterrupt:  # CTRL+C pressed
        pass
    print('Server shutting down.')
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:])
```

❶ The `init` coroutine yields a server for the event loop to drive.

❷ The `aiohttp.web.Application` class represents a web application…

❸ …with routes mapping URL patterns to handler functions; here `GET /` is routed to the `home` function (see Example 18-18).

❹ The `app.make_handler` method returns an `aiohttp.web.RequestHandler` instance to handle HTTP requests according to the routes set up in the `app` object.

❺ `create_server` brings up the server, using `handler` as the protocol handler and binding it to `address` and `port`.

❻ Return the address and port of the first server socket.

❼ Run `init` to start the server and get its address and port.

❽ Run the event loop; `main` will block here while the event loop is in control.

❾ Close the event loop.

As you get acquainted with the `asyncio` API, it's interesting to contrast how the servers are set up in Example 18-17 and in the TCP example (Example 18-15) shown earlier.

In the earlier TCP example, the server was created and scheduled to run in the `main` function with these two lines:

```
server_coro = asyncio.start_server(handle_queries, address, port,
                                   loop=loop)
server = loop.run_until_complete(server_coro)
```

In the HTTP example, the `init` function creates the server like this:

```
server = yield from loop.create_server(handler,
                                        address, port)
```

But `init` itself is a coroutine, and what makes it run is the `main` function, with this line:

```
host = loop.run_until_complete(init(loop, address, port))
```

Both `asyncio.start_server` and `loop.create_server` are coroutines that return `asyncio.Server` objects. In order to start up a server and return a reference to it, each of these coroutines must be driven to completion. In the TCP example, that was done by calling `loop.run_until_complete(server_coro)`, where `server_coro` was the result of `asyncio.start_server`. In the HTTP example, `create_server` is invoked on a `yield_from` expression inside the `init` coroutine, which is in turn driven by the `main` function when it calls `loop.run_until_complete(init(...))`.

I mention this to emphasize this essential fact we've discussed before: a coroutine only does anything when driven, and to drive an `asyncio.coroutine` you either use `yield from` or pass it to one of several `asyncio` functions that take coroutine or future arguments, such as `run_until_complete`.

Example 18-18 shows the `home` function, which is configured to handle the / (root) URL in our HTTP server.

Example 18-18. http_charfinder.py: the home function

```
def home(request):  ❶
    query = request.GET.get('query', '').strip()  ❷
    print('Query: {!r}'.format(query))  ❸
    if query:  ❹
        descriptions = list(index.find_descriptions(query))
        res = '\n'.join(ROW_TPL.format(**vars(descr))
                        for descr in descriptions)
        msg = index.status(query, len(descriptions))
    else:
        descriptions = []
        res = ''
        msg = 'Enter words describing characters.'

    html = template.format(query=query, result=res,    ❺
                           message=msg)
    print('Sending {} results'.format(len(descriptions)))   ❻
    return web.Response(content_type=CONTENT_TYPE, text=html)  ❼
```

❶ A route handler receives an `aiohttp.web.Request` instance.

❷ Get the query string stripped of leading and trailing blanks.

❸ Log query to server console.

❹ If there was a query, bind `res` to HTML table rows rendered from result of the query to the `index`, and `msg` to a status message.

❺ Render the HTML page.

❻ Log response to server console.

❼ Build `Response` and return it.

Note that `home` is not a coroutine, and does not need to be if there are no `yield from` expressions in it. The `aiohttp` documentation for the `add_route` method states that the handler "is converted to coroutine internally when it is a regular function."

There is a downside to the simplicity of the `home` function in Example 18-18. The fact that it's a plain function and not a coroutine is a symptom of a larger issue: the need to rethink how we code web applications to achieve high concurrency. Let's consider this matter.

Smarter Clients for Better Concurrency

The `home` function in Example 18-18 looks very much like a view function in Django or Flask. There is nothing asynchronous about its implementation: it gets a request, fetches data from a database, and builds a response by rendering a full HTML page. In this example, the "database" is the `UnicodeNameIndex` object, which is in memory. But accessing a real database should be done asynchronously, otherwise you're blocking the event loop while waiting for database results. For example, the `aiopg` package provides an asynchronous PostgreSQL driver compatible with `asyncio`; it lets you use `yield from` to send queries and fetch results, so your view function can behave as a proper coroutine.

Besides avoiding blocking calls, highly concurrent systems must split large chunks of work into smaller pieces to stay responsive. The *http_charfinder.py* server illustrates this point: if you search for "cjk" you'll get back 75,821 Chinese, Japanese, and Korean ideographs.[9] In this case, the `home` function will return a 5.3 MB HTML document, featuring a table with 75,821 rows.

On my machine, it takes 2s to fetch the response to the "cjk" query, using the `curl` command-line HTTP client from a local *http_charfinder.py* server. A browser takes even longer to actually layout the page with such a huge table. Of course, most queries return much smaller responses: a query for "braille" returns 256 rows in a 19 KB page and takes 0.017s on my machine. But if the server spends 2s serving a single "cjk" query, all the other clients will be waiting for at least 2s, and that is not acceptable.

The way to avoid the long response problem is to implement pagination: return results with at most, say, 200 rows, and have the user click or scroll the page to fetch more. If you look up the *charfinder.py* module in the *Fluent Python* code repository, you'll see that the `UnicodeNameIndex.find_descriptions` method takes optional `start` and `stop` arguments: they are offsets to support pagination. So you could return the first 200 results, then use AJAX or even WebSockets to send the next batch when—and if—the user wants to see it.

Most of the necessary coding for sending results in batches would be on the browser. This explains why Google and all large-scale Internet properties rely on lots of client-side coding to build their services: smart asynchronous clients make better use of server resources.

Although smart clients can help even old-style Django applications, to really serve them well we need frameworks that support asynchronous programming all the way: from the handling of HTTP requests and responses, to the database access. This is especially true if you want to implement real-time services such as games and media streaming with WebSockets.[10]

Enhancing *http_charfinder.py* to support progressive download is left as an exercise to the reader. Bonus points if you implement "infinite scroll," like Twitter does. With this challenge, I wrap up our coverage of concurrent programming with `asyncio`.

Chapter Summary

This chapter introduced a whole new way of coding concurrency in Python, leveraging `yield from`, coroutines, futures, and the `asyncio` event loop. The first simple examples, the spinner scripts, were designed to demonstrate a side-by-side comparison of the `threading` and the `asyncio` approaches to concurrency.

9. That's what CJK stands for: the ever-expanding set of Chinese, Japanese, and Korean characters. Future versions of Python may support more CJK ideographs than Python 3.4 does.

10. I have more to say about this trend in "Soapbox" on page 580.

We then discussed the specifics of `asyncio.Future`, focusing on its support for `yield from`, and its relationship with coroutines and `asyncio.Task`. Next, we analyzed the `asyncio`-based flag download script.

We then reflected on Ryan Dahl's numbers for I/O latency and the effect of blocking calls. To keep a program alive despite the inevitable blocking functions, there are two solutions: using threads or asynchronous calls—the latter being implemented as callbacks or coroutines.

In practice, asynchronous libraries depend on lower-level threads to work—down to kernel-level threads—but the user of the library doesn't create threads and doesn't need to be aware of their use in the infrastructure. At the application level, we just make sure none of our code is blocking, and the event loop takes care of the concurrency under the hood. Avoiding the overhead of user-level threads is the main reason why asynchronous systems can manage more concurrent connections than multithreaded systems.

Resuming the flag downloading examples, adding a progress bar and proper error handling required significant refactoring, particularly with the switch from `asyncio.wait` to `asyncio.as_completed`, which forced us to move most of the functionality of `download_many` to a new `downloader_coro` coroutine, so we could use `yield from` to get the results from the futures produced by `asyncio.as_completed`, one by one.

We then saw how to delegate blocking jobs—such as saving a file—to a thread pool using the `loop.run_in_executor` method.

This was followed by a discussion of how coroutines solve the main problems of callbacks: loss of context when carrying out multistep asynchronous tasks, and lack of a proper context for error handling.

The next example—fetching the country names along with the flag images—demonstrated how the combination of coroutines and `yield from` avoids the so-called callback hell. A multistep procedure making asynchronous calls with `yield from` looks like simple sequential code, if you pay no attention to the `yield from` keywords.

The final examples in the chapter were `asyncio` TCP and HTTP servers that allow searching for Unicode characters by name. Analysis of the HTTP server ended with a discussion on the importance of client-side JavaScript to support higher concurrency on the server side, by enabling the client to make smaller requests on demand, instead of downloading large HTML pages.

Further Reading

Nick Coghlan, a Python core developer, made the following comment on the draft of PEP-3156 — Asynchronous IO Support Rebooted: the "asyncio" Module in January 2013:

> Somewhere early in the PEP, there may need to be a concise description of the two APIs for waiting for an asynchronous `Future`:
>
> 1. `f.add_done_callback(…)`
> 2. `yield from f` in a coroutine (resumes the coroutine when the future completes, with either the result or exception as appropriate)
>
> At the moment, these are buried in amongst much larger APIs, yet they're key to understanding the way everything above the core event loop layer interacts.[11]

Guido van Rossum, the author of PEP-3156, did not heed Coghlan's advice. Starting with PEP-3156, the `asyncio` documentation is very detailed but not user friendly. The nine *.rst* files that make up the `asyncio` package docs total 128 KB—that's roughly 71 pages. In the standard library, only the "Built-in Types" chapter is bigger, and it covers the API for the numeric types, sequence types, generators, mappings, sets, `bool`, context managers, etc.

Most pages in the `asyncio` manual focus on concepts and the API. There are useful diagrams and examples scattered all over it, but one section that is very practical is "18.5.11. Develop with asyncio," which presents essential usage patterns. The `asyncio` docs need more content explaining how `asyncio` should be used.

Because it's very new, `asyncio` lacks coverage in print. Jan Palach's *Parallel Programming with Python* (Packt, 2014) is the only book I found that has a chapter about `asyncio`, but it's a short chapter.

There are, however, excellent presentations about `asyncio`. The best I found is Brett Slatkin's "Fan-In and Fan-Out: The Crucial Components of Concurrency," subtitled "Why do we need Tulip? (a.k.a., PEP 3156—`asyncio`)," which he presented at PyCon 2014 in Montréal (video). In 30 minutes, Slatkin shows a simple web crawler example, highlighting how `asyncio` is intended to be used. Guido van Rossum is in the audience and mentions that he also wrote a web crawler as a motivating example for `asyncio`; Guido's code does not depend on `aiohttp`—it uses only the standard library. Slatkin also wrote the insightful post "Python's asyncio Is for Composition, Not Raw Performance."

11. Comment on PEP-3156 in a Jan. 20, 2013 message to the python-ideas list.

Other must-see asyncio talks are by Guido van Rossum himself: the PyCon US 2013 keynote, and talks he gave at LinkedIn and Twitter University. Also recommended are Saúl Ibarra Corretgé's "A Deep Dive into PEP-3156 and the New asyncio Module" (slides, video).

Dino Viehland showed how asyncio can be integrated with the Tkinter event loop in his "Using futures for async GUI programming in Python 3.3" talk at PyCon US 2013. Viehland shows how easy it is to implement the essential parts of the asyncio.AbstractEventLoop interface on top of another event loop. His code was written with Tulip, prior to the addition of asyncio to the standard library; I adapted it to work with the Python 3.4 release of asyncio. My updated refactoring is on GitHub.

Victor Stinner—an asyncio core contributor and author of the Trollius backport—regularly updates a list of relevant links: The new Python asyncio module aka "tulip". Other collections of asyncio resources are Asyncio.org and aio-libs on Github, where you'll find asynchronous drivers for PostgreSQL, MySQL, and several NoSQL databases. I haven't tested these drivers, but the projects seem very active as I write this.

Web services are going to be an important use case for asyncio. Your code will likely depend on the aiohttp library led by Andrew Svetlov. You'll also want to set up an environment to test your error handling code, and the Vaurien "chaos TCP proxy" designed by Alexis Métaireau and Tarek Ziadé is invaluable for that. Vaurien was created for the Mozilla Services project and lets you introduce delays and random errors into the TCP traffic between your program and backend servers such as databases and web services providers.

Soapbox

The One Loop

For a long time, asynchronous programming has been the approach favored by most Pythonistas for network applications, but there was always the dilemma of picking one of the mutually incompatible libraries. Ryan Dahl cites Twisted as a source of inspiration for Node.js, and Tornado championed the use of coroutines for event-oriented programming in Python.

In the JavaScript world, there is some debate between advocates of simple callbacks and proponents of various competing higher-level abstractions. Early versions the Node.js API used Promises—similar to our Futures—but Ryan Dahl decided to standardize on callbacks only. James Coglan argues this was Node's biggest missed opportunity.

In Python, the debate is over: the addition of asyncio to the standard library establishes coroutines and futures as the Pythonic way of writing asynchronous code. Furthermore, the asyncio package defines standard interfaces for asynchronous futures and the event loop, providing reference implementations for them.

The *Zen of Python* applies perfectly:

> There should be one—and preferably only one—obvious way to do it.
>
> Although that way may not be obvious at first unless you're Dutch.

Maybe it takes a Dutch passport to find `yield from` obvious. It was not obvious at first for this Brazilian, but after a while I got the hang of it.

More importantly, `asyncio` was designed so that its event loop can be replaced by an external package. That's why the `asyncio.get_event_loop` and `set_event_loop` functions exist; they are part of an abstract Event Loop Policy API.

Tornado already has an AsyncIOMainLoop class that implements the `asyncio.AbstractEventLoop` interface, so you can run asynchronous code using both libraries on the same event loop. There is also the intriguing Quamash project that integrates `asyncio` to the Qt event loop for developing GUI applications with PyQt or PySide. These are just two of a growing number of interoperable event-oriented packages made possible by `asyncio`.

Smarter HTTP clients such as single-page web applications (like Gmail) or smartphone apps demand quick, lightweight responses and push updates. These needs are better served by asynchronous frameworks instead of traditional web frameworks like Django, which are designed to serve fully rendered HTML pages and lack support for asynchronous database access.

The WebSockets protocol was designed to enable real-time updates for clients that are always connected, from games to streaming applications. This requires highly concurrent asynchronous servers able to keep ongoing interactions with hundreds or thousands of clients. WebSockets is very well supported by the `asyncio` architecture and at least two libraries already implement it on top of `asyncio`: Autobahn|Python and WebSockets.

This overall trend—dubbed "the real-time Web"—is a key factor in the demand for Node.js, and the reason why rallying around `asyncio` is so important for the Python ecosystem. There's still a lot of work to do. For starters, we need an asynchronous HTTP server and client API in the standard library, an asynchronous DBAPI 3.0, and new database drivers built on `asyncio`.

The biggest advantage Python 3.4 with `asyncio` has over Node.js is Python itself: a better designed language, with coroutines and `yield from` to make asynchronous code more maintainable than the primitive callbacks of JavaScript. Our biggest disadvantage is the libraries: Python comes with "batteries included," but our batteries are not designed for asynchronous programming. The rich ecosystem of libraries for Node.js is entirely built around async calls. But Python and Node.js both have a problem that Go and Erlang have solved from the start: we have no transparent way to write code that leverages all available CPU cores.

Standardizing the event loop interface and an asynchronous library was a major coup, and only our BDFL could have pulled it off, given that there were well-entrenched, high-quality alternatives available. He did it in consultation with the authors of the major Python asynchronous frameworks. The influence of Glyph Lefkowitz, the leader of Twisted, is most evident. Guido's "Deconstructing Deferred" post to the Python-tulip group is a must-read if you want to understand why `asyncio.Future` is not like the Twisted `Deferred` class. Making clear his respect for the oldest and largest Python asynchronous framework, Guido also started the meme WWTD—What Would Twisted Do?—when discussing design options in the python-twisted group.[12]

Fortunately, Guido van Rossum led the charge so Python is better positioned to face the concurrency challenges of the present. Mastering `asyncio` takes effort. But if you plan to write concurrent network applications in Python, seek the One Loop:

> *One Loop to rule them all, One Loop to find them,*
>
> *One Loop to bring them all and in liveness bind them.*

12. See Guido's January 29, 2015, message, immediately followed by an answer from Glyph.