

### 3. Asyncio Walk-Through

#### Processes vs. Threads vs. Async

	Processes	Threads	Async
Optimize waiting periods	Yes (preemptive)	Yes (preemptive)	Yes (cooperative)
Use all CPU cores	Yes	No	No
Scalability	Low (ones/tens)	Medium (hundreds)	High (thousands+)
Use blocking std library functions	Yes	Yes	No
GIL interference	No	Some	No

## Chapter 3. Asyncio Walk-Through

*Asyncio provides another tool for concurrent programming in Python, that is more lightweight than threads or multiprocessing. In a very simple sense it does this by having an event loop execute a collection of tasks, with a key difference being that each task chooses when to yield control back to the event loop.*<sup>1</sup>

—Philip Jones, Medium

The Asyncio API in Python is complex because it aims to solve different problems for different groups of people. Unfortunately, there is very little

guidance available to help you figure out which parts of `asyncio` are important for the group *you're* in.

My goal is to help you figure that out. There are two main target audiences for the async features in Python:

#### End-user developers

These want to make applications using `asyncio`. I am going to assume that you're in this group.

#### Framework developers

These want to make frameworks and libraries that end-user developers can use in their applications.

Much of the confusion around `asyncio` in the community today is due to confusion between these two goals. For instance, the documentation for `asyncio` in the official Python documentation is more appropriate for framework developers, not end users. This means that end-user developers reading those docs quickly become shell-shocked by the apparent complexity. You're somewhat forced to take it all in before being able to do anything with it.

It is my hope that this book can help to separate, in your mind, the features

of `asyncio` that are important for end-user developers and those important for framework developers.

## TIP

If you're interested in the lower-level details around how concurrency frameworks like `asyncio` are built internally, I highly recommend a wonderful talk by Dave Beazley, [Python Concurrency From the Ground Up: LIVE!](#), in which he demonstrates putting together a simpler version of an async framework like `asyncio`.

My goal is to give you only the most basic understanding of the building blocks of Asyncio; enough that you should be able to write simple programs with it, and certainly enough that you will be able to dive into more complete references.<sup>2</sup>

First up, we have a “quickstart” section that aims to provide the most important building blocks for `asyncio` applications.

## Quickstart

*You only need to know about seven functions to use asyncio [for everyday use].*

— Yury Selivanov, author of PEP 492

It's pretty scary to open the official documentation for Asyncio. There are many sections with new, enigmatic words, and concepts that will be unfamiliar to even experienced Python programmers, as `asyncio` is a very new thing. We're going to break all that down and explain how to approach the `asyncio` documentation later, but for now you need to know that the actual surface area you have to worry about with the `asyncio` library is *much* smaller than it seems.

Yury Selivanov, the author of PEP 492 and all-round major contributor to async Python, explained in his talk *async/await in Python 3.5 And Why It Is Awesome*, presented at PyCon 2016 that many of the APIs in the `asyncio` module are really intended for *framework designers*, not end-user developers. In that talk, he emphasized the main features that end users should care about, and these are a small subset of the whole `asyncio` API.

In this section we're going to look at those core features, and see how to hit the ground looping with event-based programming in Python.

You'll need a basic knowledge of coroutines (presented in the section after this one), but except for that, if you want to be able to make use of the `asyncio` library as an end-user developer (and not a framework designer), these are the things you *need* to know, with a tiny example:

*Example 3-1. The “Hello World” of Asyncio*

---

```

# quickstart.py
import time
import asyncio

async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop() ❸
    ...

loop = asyncio.get_event_loop() ❶
loop.create_task(main()) ❷
loop.run_forever() ❸
pending = asyncio.Task.all_tasks(loop=loop)
group = asyncio.gather(*pending, return_exceptions=True)
❹
loop.run_until_complete(group) ❸
loop.close() ❺
    ...

```

Output:

```

$ python quickstart.py
Sun Sep 17 14:17:37 2017 Hello!
Sun Sep 17 14:17:38 2017 Goodbye!

```

❶ `loop = asyncio.get_event_loop()`

You need a loop instance before you can run any coroutines, and this is how you get one. In fact, anywhere you call it, `get_event_loop()` will give you the same `loop` instance each time, as long as you're using only a single thread.<sup>3</sup>

② `task = loop.create_task(coro)`

In the code above, the specific invocation is

`loop.create_task(main())`. Your coroutine function will not be executed until you do this. We say that `create_task()` *schedules* your coroutine to be run on the loop. The returned `task` object can be used to monitor the status of the task, for example whether it is still running or has completed, and can also be used to obtain a result value from your completed coroutine. You can also cancel the task with `task.cancel()`.<sup>4</sup>

③ `loop.run_until_complete(coro)` and  
`loop.run_forever()`

These are the two ways to get the loop running. Both of these will *block* the current thread, which will usually be the main thread.

Note that `run_until_complete()` will keep the loop running until the given `coro` completes—but all *other* tasks scheduled on the loop will also run while the loop is running.

④ `group = asyncio.gather(task1, task2, task3)`

The typical idiom for most programs will be to start off with `loop.run_forever()` for the “main” part of the program, and then when a process signal is received, stop the loop, gather the still-pending tasks, and then use

`loop.run_until_complete()` until those tasks are done.

This is the method for doing the gathering. More generally, it can also be used to gather multiple coroutines together and wait (using `await`!) for all of the gathered tasks to finish.

#### ⑤ `loop.stop()` and `loop.close()`

As described further above, these are used to gradually bring a program to a standstill. `stop()` is usually called as a consequence of some kind of shutdown signal being received, and `close()` is usually the final action: it must be called on a stopped loop, and it will clear all queues and shut down the *Executor*. A “stopped” loop can be restarted; a “closed” loop is gone for good.

## CAUTION

The preceding example is too simplistic to be useful in a practical setting. More information around correct shutdown handling is required. The goal of the example was merely to introduce the most important functions and methods in `asyncio`. More practical information for shutdown handling is presented later in the book.

`asyncio` in Python exposes a great deal of the underlying machinery around the event loop—and requires you to be aware of things like the event loop and its lifetime management. This is different from Node.js for example, which also contains an event loop, but keeps it somewhat hidden away. However, once you’ve worked with `asyncio` for bit, you’ll begin to notice that the pattern for starting up and shutting down the event loop doesn’t stray terribly far from the code above. And in the remainder of this book we will examine some of the nuances around loop lifetime in more detail too.

I left something out in the example above. The last item of basic functionality you’ll need to know is how to run *blocking* functions. The thing about *cooperative multitasking* is that you need all I/O-bound functions to...well, cooperate, and that means allowing a context switch back to the loop using the keyword `await`. Most of the Python code available in the wild today does not do this, and instead relies on you to run such functions in threads. Until there is more widespread support for `async def` functions, you’re going to find that using such blocking libraries is unavoidable.

For this, `asyncio` provides an API that is very similar to the API in the `concurrent.futures` package. This package provides a `ThreadPoolExecutor` and a `ProcessPoolExecutor`. The default is thread-based, but is easily replaced with a process-based one. I omitted this from the previous example because it would have obscured the description



of how the fundamental parts fit together. Now that those are covered, we can look at the executor directly.

There are a couple of quirks to be aware of. Let's have a look at a code sample:

*Example 3-2. The basic executor interface*

---

```
# quickstart_exe.py
import time
import asyncio

async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()

def blocking(): ❶
    time.sleep(0.5) ❷
    print(f"{time.ctime()} Hello from a thread!")

loop = asyncio.get_event_loop()

loop.create_task(main())
loop.run_in_executor(None, blocking) ❸

loop.run_forever()

pending = asyncio.Task.all_tasks(loop=loop) ❹
group = asyncio.gather(*pending)
loop.run_until_complete(group)
loop.close()
```

Output:

```
$ python quickstart_exe.py
Sun Sep 17 14:17:38 2017 Hello!
Sun Sep 17 14:17:38 2017 Hello from a thread!
Sun Sep 17 14:17:39 2017 Goodbye!
```

❶ `blocking()` calls the traditional `time.sleep()` internally, which *would have* blocked the main thread and prevented your event loop from running. This means that you must not make this function a coroutine, but even more severe, you cannot even call this function from *anywhere* in the main thread, which is where the `asyncio` loop is running. We solve this problem by running this function in an *executor*.

❷ Unrelated to this section, but something to keep in mind for later in the book: note that the blocking sleep time (0.5 seconds) is shorter than the non-blocking sleep time (1 second) in the `main()` coroutine. This makes the code sample neat and tidy. In a later section we'll explore what happens if executor functions outlive their async counterparts during the shutdown sequence.

❸ `await loop.run_in_executor(None, func)`  
This is the last of our list of essential, must-know features of

`asyncio`. Sometimes you need to run things in a separate thread, or even a separate process: this method is used for exactly that. Here we pass our blocking function to be run in the default executor.<sup>5</sup> Note that `loop.run_in_executor()` returns a `Future`, which means you can `await` it if called within another coroutine function.

- ④ Further to the note in item 2: the set of tasks in `pending` does *not* include an entry for the call to `blocking()` made in `run_in_executor()`. This will be true of any call that returns a `Future` rather than a `Task`. The documentation is quite good at specifying return types, so it's not hard to know; but just remember that `all_tasks()` really does return only `Tasks`, not `Futures`.

Now that you've seen the most essential parts of `asyncio` for *end-user developer* needs, it's time to expand our scope and arrange the `asyncio` API into a kind of hierarchy. This will make it easier to digest and understand how to take what you need from the documentation, but no more than that.

## The Tower of Asyncio

As you saw in “Quickstart”, there are only a handful of commands that you need to know to be able to use `asyncio` as an *end-user developer*.

Unfortunately, the documentation for `asyncio` presents a huge number of different APIs, and these are presented in a very “flat” format in which it is hard to tell which things are intended for common use, and which are facilities being provided to *framework designers*.

When framework designers look at the same documentation, they look for *hook points* to which they can connect up their new framework (or new third-party library). In this section we’ll look at `asyncio` through the eyes of a framework designer to get a sense of how they might approach building a new async-compatible library. Hopefully this will help even further to delineate the features that you need to care about in your own work.

From this perspective, it is much more useful to think about the `asyncio` module as being arranged in a *hierarchy*, rather than a flat list, in which each level is built on top of the specification of the previous level. It isn’t quite as neat as that, unfortunately, and I’ve taken liberties with the arrangement in [Table 3-1](#), but hopefully this will give an alternate view of the `asyncio` API.

## WARNING

[Table 3-1](#), and the names and numbering of “Tiers” given here, is entirely my own invention, intended to add a little structure to help explain the `asyncio` API. I made it up! The expert reader might arrange things in a different order, and that’s OK!

Level	Concept	Implementation
<b>Tier 9</b>	<b>network: streams</b>	StreamReader & StreamWriter
Tier 8	network: TCP & UDP	Protocol
Tier 7	network: transports	BaseTransport
<b>Tier 6</b>	<b>tools</b>	asyncio.Queue
<b>Tier 5</b>	<b>subprocesses &amp; threads</b>	run_in_executor(), asyncio.subprocess

		ubprocess
Tier 4	tasks	<code>asyncio.Task</code>
Tier 3	futures	<code>asyncio.Future</code>
<b>Tier 2</b>	<b>event loop</b>	BaseEventLoop
<b>Tier 1 (Base)</b>	<b>coroutines</b>	<code>async def</code> & <code>await</code>

At the most fundamental level, Tier 1, we have the coroutines that you've already seen earlier in this book. This is the lowest level at which one can begin to think about designing a third-party framework, and surprisingly, this turns out to be somewhat popular with not one, but *two* such async frameworks currently available in the wild: Curio and Trio. Both of these rely *only* on native coroutines in Python, and nothing whatsoever from the

`asyncio` library module.

The next level is the event loop. Coroutines are not useful by themselves: they won't do anything without a loop on which to run them (therefore, necessarily, Curio and Trio implement their own event loops). `asyncio` provides both a loop *specification*, `AbstractEventLoop`, as well as an *implementation*, `BaseEventLoop`.

The clear separation between specification and implementation makes it possible for third-party developers to make alternative implementations of the event loop, and this has already happened with the uvloop project, which provides a much faster loop implementation than the one in the `asyncio` standard library module. Importantly: uvloop simply “plugs into” the hierarchy, and replaces *only* the loop-part of the stack. The ability to make these kinds of choices is exactly why the `asyncio` API has been designed like this, with clear separation between the different moving parts.

Tiers 3 and 4 bring us futures and tasks, which are very closely related; they're separated only because `Task` is a subclass of `Future`, but they could easily be considered to be in the same tier. A `Future` instance represents some sort of ongoing action which will return a result via *notification* on the event loop, while a `Task` represents a *coroutine* running on the event loop. The short story is: a future is “loop-aware,” while the task is *both* “loop-aware” *and* “coroutine-aware.” As an end-user developer, you will be working with tasks much more than futures, but for a framework

designer, the proportion might be the other way round depending on the details.

Tier 5 represents the facilities for launching, and `await`-ing on work that must be run either in a separate thread, or even in a separate process.

Tier 6 represents additional async-aware tools such as `asyncio.Queue`. We could have placed this tier after the network tiers, but I think it's neater to get all of the coroutine-aware APIs out of the way first, before we look at the I/O layers. The `Queue` provided by `asyncio` has a very similar API to the thread-safe `Queue` in the `queue` module, except that the `asyncio` version requires the `await` keyword on `get()` and `put()`. You cannot use `queue.Queue` directly inside coroutines because its `get()` will block the main thread.

Finally, we have the network I/O tiers, 7 through 9. As an end-user developer, the most convenient API to work with is the “streams” API at Tier 9. I have positioned the streams API at the highest level of abstraction in the tower. The “protocols” tier, immediately below that (Tier 8), is a more fine-grained API than the “streams” API; you *can* use the “protocols” tier in all instances where you might use the “streams” tier, but “streams” will be simpler. Finally, it is unlikely you will ever have to work with the transport tier (Tier 7) directly, unless you're creating a framework for others to use and you need to customize how the transports are set up.



## Summary

In “Quickstart” we looked at the absolute bare minimum that one would need to know to get started with the `asyncio` library. Now that we’ve had a look at how the entire `asyncio` library API is put together, I’d like to revisit that short list of features and re-emphasize which parts you are likely to need to learn.

These are the tiers that are most important to focus on when learning how to use the `asyncio` library module for writing network applications:

Tier 1: understanding how to write `async def` functions, and using `await` to call and execute other coroutines is essential.

Tier 2: understanding how to start up, shut down, and interact with the event loop is essential.

Tier 5: executors are necessary to use blocking code in your `async` application, and the reality is that most third-party libraries are not yet `asyncio`-compatible. A good example of this is the SQLAlchemy database ORM library, for which no feature-comparable alternative is available right now for `asyncio`.

Tier 6: if you need to feed data to one or more long-running coroutines, the best way to do that is with `asyncio.Queue`. This is exactly the same strategy as using `queue.Queue` for distributing data between threads. The

Asyncio version of the `Queue` uses the same API as the standard library `queue` module, but uses coroutines instead of the blocking methods like `get()`.

Tier 9: The *Streams* API gives you the simplest way to handle socket communication over a network, and it is here that you should begin prototyping ideas for network applications. You may find that more fine-grained control is needed, and then you could switch to the *Protocols* API, but—in most projects—it is usually best to keep things simple until you know exactly what problem you’re trying to solve.

Of course, if you’re using an `asyncio`-compatible third-party library that handles all the socket communication for you like, say `aiohttp`, you won’t even need to directly work with the `asyncio` network tiers at all. In this case you must rely heavily on the documentation provided with such libraries.

This brings us to the end of this section. The `asyncio` library tries to provide sufficient features for both end-user developers as well as framework designers. Unfortunately, this means that the `asyncio` API can appear somewhat sprawling. I hope that this section can provide a crude “road map” to help you pick out the parts you need.

In the next sections we’re going to look at the component parts of my short list above in more detail.

## TIP

The [pysheet](#) site provides an in-depth summary (or “cheat sheet”) of large chunks of the `asyncio` API where each concept is presented with a short code snippet. The presentation is dense so I wouldn’t recommend it for beginners, but if you have experience with Python and you’re the kind of person that only “gets it” when new programming info is presented in code, this is sure to be a useful resource.

## Coroutines

Let’s begin at the very beginning: what is a coroutine?

I’m going to let you take a peek under the hood and look at some parts of the engine that you will not normally see, or even use, during day-to-day async programming. The following examples can all be reproduced in the Python interpreter in interactive mode, and I urge you to work through them on your own by typing them yourself, observing the output, and perhaps experimenting with different ways of interacting with `async` and `await`.

## CAUTION

`asyncio` was first added to Python 3.4, but the new syntax for coroutines using `async def` and `await` was only added in Python 3.5. How did people do anything with `asyncio` in 3.4? They used *generators* in very special ways to act as if they were coroutines. In some older codebases, you’ll see generator functions decorated with `@asyncio.coroutine` and containing `yield` from statements. Coroutines created with the newer `async def` are now referred to as “native” coroutines because they are built into the language as coroutines and nothing else. This book ignores the older generator-based coroutines entirely.

## The New `async def` Keyword

Let us begin with the simplest possible thing:

*Example 3-3. The first surprise*

---

```
>>> async def f(): ❶
...     return 123
...
>>> type(f) ❷
<class 'function'>
>>> import inspect ❸
>>> inspect.iscoroutinefunction(f) ❹
True
```

❶ This is the simplest possible declaration of a coroutine: it looks like a regular function, except that it begins with the keywords `async` `def`.

❷ Surprise! The precise type of `f` is *not* `coroutine`, but just an ordinary function. It is common to refer to such “`async def`” functions as “coroutines” even though—strictly speaking—they are considered by Python to be *coroutine functions*. This behavior is exactly identical to how generator functions already work in Python:

```
>>> def g():
...     yield 123
...
>>> type(g)
<class 'function'>
>>> gen = g()
>>> type(gen)
<class 'generator'>
```

Even though `g` is sometimes incorrectly referred to as a “generator,” it remains a function and it is only when this function is *evaluated*, that the generator is returned. Coroutine functions work in exactly the same way: you need to *call* the `async def` function to obtain the coroutine.

- ③ The `inspect` module in the standard library can provide much better introspective capabilities than the `type()` built-in function.
- ④ There is an `inspect.iscoroutinefunction()` function that lets you distinguish between an ordinary function and a coroutine function.

Returning to our `async def f()`, what happens when we call it?

*Example 3-4. An `async def` function returns...a coroutine!*

---

```
>>> coro = f()
>>> type(coro)
<class 'coroutine'>
>>> inspect.iscoroutine(coro)
True
```

So now the question becomes: what exactly is a “coroutine”? Coroutines are very similar to generators. Indeed, before the introduction of *native* coroutines with the `async def` and `await` keywords in Python 3.5, it was already possible to use the `Asyncio` library in Python 3.4 using normal generators with special decorators.<sup>6</sup> It isn’t surprising that the new `async def` functions (and the coroutines they return) behave in a similar way to generators.

We can play with coroutines a bit more to see how Python makes use of

them. Most importantly, we want to see how Python is able to “switch” execution between coroutines. Let’s first look at how the `return` value can be obtained.

When a coroutine *returns*, what really happens is that a `StopIteration` exception is raised. This next example, which continues in the same session as the previous examples, makes that clear:

```
>>> async def f():
...     return 123
>>> coro = f()
>>> try:
...     coro.send(None) ❶
... except StopIteration as e:
...     print('The answer was:', e.value) ❷
...
The answer was: 123
```

❶ A *coroutine* is initiated by “sending” it a `None`. Internally, this is what the *event loop* is going to be doing to your precious coroutines. You’ll never have to do this manually. All the coroutines you make will be executed either with `loop.create_task(coro)` or `await coro`. It’s the loop that does the `.send(None)` behind the scenes.

❷ When the coroutine *returns*, a special kind of exception is raised,

called `StopIteration`. Note that we can access the return value of the coroutine via the `value` attribute of the exception itself. Again, you don't need to know that it works like this: from your point of view, `async def` functions will simply return a value with the `return` statement, just like a normal function.

These two points, i.e., the `send()` and the `StopIteration`, define the start and end of the executing coroutine respectively. So far, this just seems like a really convoluted way to run a function, but that's OK: the *event loop* will be responsible for driving coroutines with these low-level internals. From your point of view, you will simply schedule coroutines for execution on the loop, and they will get executed top-down, almost like normal functions.

The next step is to see how the execution of the coroutine can be suspended.

## The New `await` Keyword

This new keyword, `await`, always takes a parameter and will *only* accept a thing called an *awaitable*, which is defined as one of these (exclusively!) <sup>7</sup> :

A coroutine (i.e., the *result* of a called `async def` function) <sup>8</sup>

Any object implementing the `__await__()` special method. That special method *must* return an iterator.



The second kind of awaitable is out of scope for this book (you'll never need it in day-to-day `asyncio` programming), but the first use case is pretty straightforward:

---

*Example 3-5. Using `await` on a coroutine*

---

```
async def f():
    await asyncio.sleep(1.0)
    return 123

async def main():
    result = await f() ❶
    return result
```

- ❶ Calling `f` produces a coroutine; this means we are allowed to `await` it. The value of the `result` variable will be 123 when `f()` completes.

Before we close out this section and move on to the event loop, it is useful to look at how coroutines may be fed exceptions, which is most commonly used for cancellation: when you call `task.cancel()`, the event loop will internally use `coro.throw()` to raise `asyncio.CancelledError` inside your coroutine:

---

*Example 3-6. Using `coro.throw()` to inject exceptions into a coroutine*

---

```
>>> coro = f() ❶
```

```

>>> coro.send(None)
>>> coro.throw(Exception, 'blah') ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
Exception: blah
blah

```

❶ As before, a new coroutine is created from coroutine function `f()`.

❷ Instead of doing another `send()`, we call `throw()` and provide an exception class and a value. This raises an exception *inside* our coroutine, at the `await` point.

The `throw()` method is used (internally in `asyncio`) for *task cancellation*, which we can also demonstrate quite easily. We're even going to go ahead and handle the cancellation inside a new coroutine that will handle it:

#### Example 3-7. Coroutine cancellation with `CancelledError`

```

>>> import asyncio
>>> async def f():
...     try:
...         while True: await asyncio.sleep(0)
...     except asyncio.CancelledError: ❶
...         print('I was cancelled!') ❷
...     else:

```

```

...         return 111
>>> coro = f()
>>> coro.send(None)
>>> coro.send(None)
>>> coro.throw(asyncio.CancelledError) ❸
I was cancelled! ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration ❺

```

❶ Our coroutine function now handles an exception: in fact, the *specific* exception type used throughout the `asyncio` library for task cancellation: `asyncio.CancelledError`. Note that the exception is being injected into coroutine from outside, i.e., by the event loop, which we're still simulating with manual `send()` and `throw()` commands. In real code, which you'll see later, when *tasks* are cancelled, `CancelledError` is raised inside the task-wrapped coroutine as shown above.

❷ A simple message to say that we got cancelled. Note that by handling the exception, it will no longer propagate and our coroutine will `return`.

❸ Here we `throw()` the `CancelledError` exception.

④ ... As expected, we see our cancellation message being printed.

⑤ ... And our coroutine exits normally. (Recall that the `StopIteration` exception is the normal way that coroutines exit.)

Just to drive the point home about how *task cancellation* is nothing more than regular exception raising (and handling), here's an example where we absorb cancellation and move on to a different coroutine:

```
>>> async def f():
...     try:
...         while True: await asyncio.sleep(0)
...     except asyncio.CancelledError:
...         print('Nope!')
...         while True: await asyncio.sleep(0) ❶
...     else:
...         return 111
>>> coro = f()
>>> coro.send(None)
>>> coro.throw(asyncio.CancelledError) ❷
Nope!
>>> coro.send(None) ❸
```

❶ ... Instead of printing a message, what happens if, after cancellation we just go right back to awaiting another awaitable?

- ② Unsurprisingly, our outer coroutine continues to live, and it immediately suspends again inside the *new* coroutine.
- ③ Everything proceeds normally, and our coroutine continues to suspend and resume as expected.

Of course, it should go without saying that you should never actually do this! If your coroutine receives a cancellation signal, that is a clear directive to do only whatever cleanup is necessary and exit. Don't just ignore it.

By this point, we are now surely getting pretty tired of *pretending* to be an event loop, by manually doing all the `.send(None)` calls, so instead we'll bring in the loop provided by `asyncio` and clean up the preceding example accordingly:

```
>>> async def f():
...     await asyncio.sleep(0)
...     return 111
>>> loop = asyncio.get_event_loop() ①
>>> coro = f()
>>> loop.run_until_complete(coro) ②
111
```

- ① Obtain a loop.

- ② ... Run the coroutine to completion. Internally, this is doing all those `.send(None)` method calls for us, and detects completion of our coroutine with the `StopIteration` exception, which also contains our return value.

## Event Loop

In the previous section, we showed how the `send()` and `throw()` methods can interact with a coroutine, but that's only for understanding how coroutines themselves are structured. The event loop in `asyncio` handles all of the switching between coroutines, as well as catching those `StopIteration` exceptions, and much more, such as listening to sockets and file descriptors for events.

You can obtain an event loop by calling `get_event_loop()`, which is an interesting function:

*Example 3-8. Always getting the same event loop*

---

```
>>> loop = asyncio.get_event_loop()
>>> loop2 = asyncio.get_event_loop()
>>> loop is loop2 ①
True
```

- ① ... Both identifiers `loop` and `loop2` refer to the same instance.

This means that if you're inside a coroutine function, and you need access to the loop instance, it's fine to call `get_event_loop()` to obtain that. You *do not* need to pass an explicit `loop` parameter through all your functions. The situation is different if you're a framework designer: if so, it would be better to design your functions to accept a `loop` parameter, just in case your users are doing something unusual with event loop policies. Policies are out of scope for this book, and we'll say no more about them.

## TIP

The `get_event_loop()` method only works within the *same thread*: In fact, `get_event_loop()` will fail if called inside a new thread unless you specifically create a new loop with `new_event_loop()`, *and* set that new instance to be “the” loop for that thread by calling `set_event_loop()`. Most of us will only ever need (and want!) a single loop instance running in a single thread. This is nearly the entire point of async programming in the first place.

Let's explore an example: consider a coroutine function inside which additional tasks are created and *not* awaited:

```
async def f():
```

```
# Create some tasks!
loop = asyncio.get_event_loop()
for i in range():
    loop.create_task(<some other coro>)
```

In this example, the intention is to launch completely new tasks inside the coroutine. By not awaiting them, they will run independently of the execution context inside coroutine function `f()`. In fact, `f()` will exit before the tasks that it launched will have completed.

For situations like this, you will sometimes see code where the loop variable is passed in as a parameter, simply to make it available so that the `create_task()` method can be called. The alternative is to use the confusingly named `asyncio.ensure_future()`, which does the same thing as `create_task()`, but doesn't require a local loop variable:

```
async def f():
    # Create some tasks!
    for i in range():
        asyncio.ensure_future(<some other coro>)
```

If you prefer, you could make your own helper function that also doesn't require a loop parameter, but with a better name:

```
def create_task(coro):
    return asyncio.get_event_loop().create_task(coro)
```



```
async def f():  
    # Create some tasks!  
    for i in range():  
        create_task(<some other coro>)
```

The difference between `loop.create_task()` and `asyncio.ensure_future()` is subtle and confusing for many newcomers. We explore these differences in the next section.

## Tasks and Futures

In the previous section we covered coroutines, and how they need to be run on a loop to be useful. Here we talk briefly about the `Task` and `Future` APIs. The one you will work with the most is `Task`, as most of your work will involve running coroutines with the `loop.create_task()` method, exactly as set out in the *Quickstart* section earlier. The `Future` class is actually a superclass of `Task`, and provides all of the functionality for interaction with the loop.

A simple way to think of it is like this: A `Future` represents a future completion state of some activity and is managed by the loop, whereas a `Task` is exactly the same *and* where the specific “activity” is a coroutine; probably one of yours that you created with an `async def` function plus `loop.create_task()`.

The `Future` class represents a *state* of something that is interacting with a

loop. That description is too fuzzy to be useful, so think of a `Future` instance like this instead: it is a toggle for completion status. When a `Future` instance is created, the toggle is “not yet completed”; but at some later time, it will be completed. In fact, a `Future` instance has a method called `done()`:

```
>>> from asyncio import Future
>>> f = Future()
>>> f.done()
False
```

A `Future` instance may also:

have a “result” value set (`.set_result(value)` and `.result()` to obtain it)

be cancelled with `.cancel()` (and check for cancellation with `.cancelled()`)

have additional callback functions added that will be run when the future completes.

Even though `Tasks` are more common, you can’t avoid `Future` entirely: for instance, running a function on an *executor* will return a `Future` instance, *not* a `Task`. Let’s take a quick look at a code sample to get a feel of what it is like to work with a `Future` instance directly:

```

>>> import asyncio
>>> async def main(f: asyncio.Future): ❶
...     await asyncio.sleep(1)
...     f.set_result('I have finished.') ❷
>>> loop = asyncio.get_event_loop()
>>> fut = asyncio.Future() ❸
>>> print(fut.done()) ❹
False
>>> loop.create_task(main(fut)) ❺
<Task pending coro=<main() running at <ast>:4>>
>>> loop.run_until_complete(fut) ❻
'I have finished.'
>>> print(fut.done())
True
>>> print(fut.result()) ❼
I have finished.

```

❶ Create a simple main function. This just gives us something to run, wait for a bit, and then set a result on this `Future`, `f`.

❷ Set the result.

❸ Manually create a future instance. Note that this instance is (by default) tied to our `loop`, but it is not, and will not be, attached to any coroutine (that's what *tasks* are for).

❹ Before doing anything, verify that the future is not done yet.

- ⑤ *Schedule* the `main()` coroutine, passing the future. Remember, all the `main()` coroutine does is sleep, then toggle the *future* instance. (Note that the `main()` coroutine will not start running yet: coroutines only run when the loop is running.)
- ⑥ This is different from what you've seen before: here we use `run_until_complete()` on a `Future` instance, rather than a `Task` instance.<sup>9</sup> Now that the loop is running, the `main()` coroutine will begin executing.
- ⑦ Eventually the future completes when its result is set. After completion, the result can be accessed.

Of course, it is unlikely that you will work with `Future` directly in the way shown above. The code sample is for education only. Most of your contact with `asyncio` will be through `Task` instances.

One final example to prove that a `Task` is really a shallow embellishment on a `Future`: we can repeat the *exact* same example as above, but using a `Task` instance instead:

```
>>> import asyncio
>>> async def main(f: asyncio.Future):
...     await asyncio.sleep(1)
```

```

...     f.set_result('I have finished.')
>>> loop = asyncio.get_event_loop()
>>> fut = asyncio.Task(asyncio.sleep(1_000_000)) ❶
>>> print(fut.done())
False
>>> loop.create_task(main(fut))
<Task pending coro=<main() running at <ast>:4>>
>>> loop.run_until_complete(fut)
'I have finished.'
>>> print(fut.done())
True
>>> print(fut.result())
I have finished.

```

- ❶ The only difference: a `Task` instance instead of `Future`. Of course, the `Task` API requires us to provide a coroutine, so we just use a `sleep()` because it's convenient, and we use a comically large duration to highlight a potential gotcha: the `Task` will complete when a result is set, *regardless* of whether the underlying coroutine has completed or not.

This example works *exactly* as it did before, and shows how even a task can be completed early by manually setting a result on it, in exactly the same way that a `Future` instance can be made complete. Again, not terribly useful in practice—you will instead usually wait for tasks to complete or cancel them explicitly—but hopefully useful for understanding that tasks and futures are very nearly synonymous: the only difference is that a task is

associated with a coroutine.

## Create a Task? Ensure a Future? Make Up Your Mind!

In “[Quickstart](#)” we said that the way to run coroutines was to use `loop.create_task(coro)`. It turns out that this can also be achieved with a different, module-level function: `asyncio.ensure_future()`.

During my research for this book, I have become convinced that the API method `asyncio.ensure_future()` is responsible for much of the widespread misunderstanding about the `asyncio` library. Much of the API is really quite clear, but there are a few bad stumbling blocks to learning, and this is one of them. When you come across `ensure_future()`, your brain works very hard to integrate it into a mental model of how `asyncio` should be used—and fails!

The problem with `ensure_future()` is best highlighted by this now-infamous explanation in the Python `asyncio` documentation:

```
asyncio.ensure_future(coro_or_future, *, loop=None)
```

*Schedule the execution of a coroutine object: wrap it in a future. Return a Task object.*

*If the argument is a Future, it is returned directly.*

—Python 3.6 Documentation

What!? Here is a, hopefully, clearer description of `ensure_future()`:

If you pass in a coroutine, it will produce a `Task` instance (and your coroutine will be scheduled to run on the event loop). This is identical to calling `loop.create_task(coro)` and returning the new `Task` instance.

If you pass in a `Future` instance (which includes `Task` instances, because `Task` is a subclass of `Future`), you get that very same thing returned, *unchanged*. Yes, really!

Let's have a closer look at how that works:

```
import asyncio

async def f(): ❶
    pass

coro = f() ❷
loop = asyncio.get_event_loop() ❸

task = loop.create_task(coro) ❹
assert isinstance(task, asyncio.Task) ❺

new_task = asyncio.ensure_future(coro) ❻
assert isinstance(new_task, asyncio.Task)

mystery_meat = asyncio.ensure_future(task) ❼
assert mystery_meat is task ❽
```

- ① A simple do-nothing coroutine function. We just need something that can make a coroutine.
- ② Here we make that coroutine object by calling the function directly. Your code will rarely do this, but I want to be explicit here (a few lines down) that we're passing a coroutine object into each of `create_task` and `ensure_future()`.
- ③ Obtain the loop.
- ④ First off, we use `loop.create_task()` to schedule our coroutine on the loop, and we get a new `Task` instance back.
- ⑤ Here we verify the type. So far, nothing interesting.
- ⑥ Here, we show that `asyncio.ensure_future()` can be used to perform the same act as `create_task`: we passed in a *coroutine* and we got back a `Task` instance (and the coroutine has been scheduled to run on the loop)! If you're passing in a coroutine, there is no difference between `loop.create_task()` and `asyncio.ensure_future()`.



But what happens if we pass a `Task` instance to `ensure_future()`...? Note that we're passing in an already-created task instance that was created by `loop.create_task()` in step 4.

- ⑧ We get back *exactly* the same `Task` instance as we passed in: it passes through unchanged.

So: what is the point of passing `Future` instances straight through? And why do two different things with the same function? The answer is that this function, `ensure_future()`, is intended to be used for *framework authors* to provide APIs to *end-user developers* that can handle both kinds of parameters. Don't believe me? Here it is from the BDFL himself:

*The point of `ensure_future()` is if you have something that could either be a coroutine or a `Future` (the latter includes a `Task` because that's a subclass of `Future`), and you want to be able to call a method on it that is only defined on `Future` (probably about the only useful example being `cancel()`). When it is already a `Future` (or `Task`) this does nothing; when it is a coroutine it wraps it in a `Task`.*

*If you know that you have a coroutine and you want it to be scheduled, the correct API to use is `create_task()`. The only time when you should be calling `ensure_future()` is when you are providing an API (like most of `asyncio`'s own APIs) that accepts either a coroutine or a `Future` and you*

*need to do something to it that requires you to have a `Future`.*<sup>10</sup>

—Guido van Rossum, [github.com/python/asyncio/issues/477](https://github.com/python/asyncio/issues/477)

In sum: the `asyncio.ensure_future()` API is a helper function intended for framework designers. It is easiest to explain in analogy to a much more common kind of function: if you have a few years' programming experience behind you, you may have seen functions similar to the `listify()` function below:

```
def listify(x: Any) -> List:
    """ Try hard to convert x into a list """
    if isinstance(x, (str, bytes)):
        return [x]

    try:
        return [_ for _ in x]
    except TypeError:
        return [x]
```

This function tries to convert the argument into a list, no matter what comes in. These kinds of functions are often used in APIs and frameworks to *coerce* inputs into a known type, which simplifies subsequent code because you know that the parameter (output from `listify()`) will always be a list.

If I renamed my `listify()` function to `ensure_list()`, then you should begin to see the parallel with `asyncio.ensure_future()`? It

tries to always coerce the argument into a `Future` (or subclass) type. This is a utility function to make life easier for *framework developers*, and not end-user developers like you and I.

And indeed, even the `asyncio` standard library module itself uses `ensure_future()` for exactly this reason. When next you look over the API, everywhere you see a function parameter described as `coro_or_future`, it is likely that internally, `ensure_future()` is being used to coerce the parameter. For example, the `asyncio.gather()` function has the following signature:

```
asyncio.gather(*coros_or_futures, loop=None, ...)
```

Internally, `gather()` is using `ensure_future()` for type coercion, and this allows you to pass it either coroutines, tasks, or futures.

The key point here is: as an end-user application developer, you should never need to use `asyncio.ensure_future`. It's more a tool for framework designers. If you need to schedule a coroutine on the event loop, just do that directly with `loop.create_task()`.

Unfortunately, that is not quite the end of the story. Right now, some end-user developers *prefer* using `asyncio.ensure_future()` over `loop.create_task()` for one very simple, pragmatic reason: it's less work! To call `create_task()`, you either need a `loop` instance available

in the local namespace where you're writing code, or you need an additional call to `asyncio.get_event_loop()` to get the loop reference; by contrast, `ensure_future()` can be called as is.

In the example below, we have three coroutine functions inside which a *background* coroutine will be launched. The goal of the exercise is to compare the aesthetic benefits of using `create_task()` compared to `ensure_future()` for scheduling coroutines on the loop.

*Example 3-9. Comparing how to use `create_task()` versus `ensure_future`*

---

```
import asyncio

async def background_job():
    pass

async def option_A(loop): ❶
    loop.create_task(background_job())

async def option_B(): ❷
    asyncio.ensure_future(background_job())

async def option_C(): ❸
    loop = asyncio.get_event_loop()
    loop.create_task(background_job())

loop = asyncio.get_event_loop()

loop.create_task(option_A(loop)) ❶
loop.create_task(option_B) ❷
loop.create_task(option_C) ❸
```

- ① In the first option, after creating the loop as normal, we schedule the coroutine returned from `option_A()`. Internally, a *new task* is created for the background job. Because we don't *await* the background task, the coroutine `option_A()` will exit; but that is not of interest here. To schedule the background task using the `create_task()` call, it was necessary to have the `loop` object available. In this case we passed it in when the `option_A()` coroutine was made.
- ② In the second option, we also successfully schedule a background task, but it was not necessary to pass in the `loop` instance, because the `ensure_future()` function is available directly in the `asyncio` module. And this is the point: some people are using `ensure_future()` *not* for its ability to coerce parameters to `Future` instances, but instead to avoid having to pass around `loop` identifiers.
- ③ In the third and final option, it is also not necessary to pass in a `loop` instance. Here, we obtain the current<sup>11</sup> event loop by calling `get_event_loop()`, and then we are once again able to do “the right thing” and call `loop.create_task()`. This is similar to how `ensure_future()` does it internally.

What really *should have* been available is an `asyncio.create_task(coro)` helper function that does exactly the same thing as our “Option C” above, but is predefined in the standard library. This will negate the convenience of `ensure_future()` and preserve the clarity of `loop.create_task()`. This need has not gone unnoticed by the Python development team, and I’m happy to say that `asyncio.create_task()` will indeed be available in Python 3.7!

In the next few sections we’ll go back to language-level features, starting with asynchronous context managers.

## Async Context Managers: `async with`

Support for coroutines in *context managers* turns out to be exceptionally convenient. This makes sense, because many situations require network resources—say, connections—to be opened and closed around a well-defined scope.

The key to understanding `async with` is to realize that the operation of a context manager is driven by *method calls*; and then consider: what if those methods were coroutine functions? Indeed, this is exactly how it works. Here is pseudocode to demonstrate:

*Example 3-10. Async context manager*

---

```
class Connection:
```

```

def __init__(self, host, port):
    self.host = host
    self.port = port
    async def __aenter__(self): ❶
        self.conn = await get_conn(self.host, self.port)
        return conn
    async def __aexit__(self, exc_type, exc, tb): ❷
        await self.conn.close()

async with Connection('localhost', 9001) as conn:
    <do stuff with conn>

```

❶ Instead of the `__enter__()` special method for synchronous context managers, the new `__aenter__()` special method is used.

❷ Likewise, instead of `__exit__()`, use `__aexit__()`. The other parameters are identical to those for `__exit__()` and are populated if an exception was raised in the body of the context manager.

## CAUTION

Just because you might be using `asyncio` in your program, it doesn't mean that all your context managers must be async ones like this! They're only useful if you need to `await` something

inside the *enter* and *exit* methods. If there is no blocking I/O code, just use regular context managers.

Now—between you and me—I don’t much like this explicit style of context manager when the wonderful `@contextmanager` decorator exists in the `contextlib` module of the standard library! As you might guess, an asynchronous version, `@asynccontextmanager` exists, but unfortunately it will only be available in Python 3.7, which is not yet available at the time of writing this book. Nevertheless, it’s fairly easy to compile Python from source and in the next section we’ll show how `@asynccontextmanager` will work in Python 3.7.

## The contextlib Way

This is analogous to the `@contextmanager` decorator in the `contextlib` standard library. To recap, let’s have a look at the blocking way first:

*Example 3-11. The blocking way*

---

```
from contextlib import contextmanager

@contextmanager ❶
def web_page(url):
    data = download_webpage(url) ❷
    yield data
```



```

        update_stats(url) ❸
    """

    with web_page('google.com') as data: ❹
        process(data) ❺
    """

```

- ❶ The `contextmanager` decorator transforms a generator function into a context manager.
- ❷ This function call (which I made up for this example) looks suspiciously like the sort of thing that will want to use a network interface, which is many orders of magnitude slower than “normal,” CPU-bound code. This context manager *must* be used in a dedicated thread, otherwise the whole program will be paused while waiting for data.
- ❸ Imagine that we update some statistics every time we process data from a URL, such as the number of times a URL has been downloaded. Again, from a concurrency perspective, we would need to know whether this function involves I/O internally, such as writing to a database over a network. If so, `update_stats()` is also a blocking call.
- ❹ Here our context manager is being used. Note specifically how the network call (to `download_webpage()`) is hidden inside the

construction of the context manager.

- ⑤ This function call, `process ( )`, might also be blocking. We'd have to look at what the function does. To give you an overview of the various considerations that matter for deciding whether a function call is “blocking” or “non-blocking,” it might be:
- innocuous, non-blocking (fast and CPU-bound)
  - mildly-blocking (fast and I/O-bound, perhaps something like fast disk access instead of network I/O)
  - blocking (slow, I/O-bound)
  - diabolical (slow, CPU-bound)

For the sake of simplicity in this example, let's presume that the call to `process ( )` is a fast, CPU-bound operation and therefore non-blocking.

Now let's compare exactly the same example, but using the new async-aware helper coming in Python 3.7:

*Example 3-12. The non-blocking way*

---

```
from contextlib import asynccontextmanager

@asynccontextmanager ①
async def web_page(url): ②
    data = await download_webpage(url) ③
    yield data ④
```

```

    await update_stats(url) ❸
    ...

async with web_page('google.com') as data: ❹
    process(data)

```

- ❶ The new `asynccontextmanager` is used in exactly the same way.
- ❷ It does, however, require that the decorated generator function be declared with `async def`.
- ❸ As before, we fetch the data from the URL before making it available to the body of the context manager. Here, I have added the `await` keyword, which tells us that this coroutine will allow the event loop to run other tasks while we wait for the network call to complete.

Note that we *cannot* simply tack on the `await` keyword to anything. This change presupposes that we were also able to *modify* the `download_webpage()` function itself, and convert it into a coroutine that is compatible with the `await` keyword. For the times when this is not possible to modify the function, a different approach is needed and we'll discuss that in the next example.

- ④ ... As before, the data is made available to the body of the context manager. I'm trying to keep the code simple, and so I've omitted the usual `try/finally` handler that you should normally write to deal with exceptions raised in the body of caller.

Note: The presence of `yield` is what changes a function into a *generator function*; the additional presence of the `async def` keywords in point 1 makes this an *asynchronous generator function*. When called, it will return an *asynchronous generator*. The `inspect` module has two functions that can test for these: `isasyncgenfunction()` and `isasyncgen()` respectively.

- ⑤ ... Here, assume that we've also converted the code inside function `update_stats()` to allow it to produce coroutines. We can then use the `await` keyword which allows a context switch to the event loop while we wait for the I/O-bound work to complete.

- ⑥ ... Another change was required in the usage of the context manager itself: we needed to use `async with` instead of a plain `with`.

Hopefully this example shows that the new `@asynccontextmanager` is perfectly analogous to the `@contextmanager` decorator.

In callout 3 for the preceding example, I said that it was necessary to modify some functions to return coroutines; these were `download_webpage()` and `update_stats()`. This is usually not that easy to do, since async support needs to be added down at the socket level.

The focus of the preceding examples was simply to show off the new `asynccontextmanager`, not show how to convert blocking functions into non-blocking ones. The more common situation is that you might want to use a blocking function in your program, but it will not be possible to modify the code in those functions.

This situation will usually happen with third-party libraries, and a great example here might be the *requests* library, which uses blocking calls throughout.<sup>12</sup> Well, if you can't change the code being called, there is another way, and this is a convenient place to show you how an *executor* can be used to do exactly that:

*Example 3-13. The non-blocking-with-a-little-help-from-my-friends way*

---

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def web_page(url): ❶
    loop = asyncio.get_event_loop()
    data = await loop.run_in_executor(
        None, download_webpage, url) ❷
    yield data
    await loop.run_in_executor(None, update_stats, url) ❸
```

```
async with web_page('google.com') as data:
    process(data)
```

- ❶ For this example, assume that we are *unable* to modify the code for our two blocking calls `download_webpage()` and `update_stats()`, i.e., we can't alter them to be coroutine functions. That's bad, because the most grave sin of event-based programming is breaking the rule that you must never, under any circumstances, prevent the event loop from processing events.

To get around the problem, we will use an *executor* to run the blocking calls in a separate thread. The *executor* is made available to us as an attribute of the event loop itself.

- ❷ Here we call the executor. The signature is `AbstractEventLoop.run_in_executor(executor, func, *args)` and if you want to use the default executor (which is a `ThreadPoolExecutor`) then you must pass `None` as the value for the “executor” argument.<sup>13</sup>

- ❸ As with the call to `download_webpage()`, we also run the other blocking call to `update_stats()` in an executor. Note that you *must* use the `await` keyword out in front. If you forget, the execution of the asynchronous generator (i.e., your `async` context

manager) will not wait for the call to complete before proceeding.

It's likely that async context managers are going to be heavily used in many `asyncio`-based codebases, so it's pretty important to have a good understanding of them. You can read more about the new async context manager decorator in the [Python 3.7 documentation](#).

## IMPORTANT

`asyncio` is still under very active development by the Python development team, and there are several other small, but significant improvements besides `asynccontextmanager` that will be available in Python 3.7 including this small sample:

`asyncio.run()`: a new helper function intended to be used as a main entry point for `asyncio` programs.

`asyncio.create_task()`: create a task without requiring an explicit loop identifier.

`AbstractEventLoop.sock_sendfile()`: send a file over a TCP socket using the high-performance `os.sendfile()` API.

`AbstractEventLoop.start_tls()`: upgrade an existing connection to transport-layer security (TLS).

`asyncio.Server.serve_forever()`: a cleaner API to start `asyncio` network servers.

## Async Iterators: `async for`

Along with `async def` and `await`, there are a few other extensions to the Python language syntax. First up is the `async` version of the “for-loop.” It is easiest to understand how this works if you first recognize that ordinary iteration—just like so many other language features—is implemented through the use of *special methods*, recognizable by the double-underscores in their names.

For example, this is how a standard (non-`async`) iterator is defined through the use of the `__iter__()` and `__next__()` methods:

```
>>> class A:
...     def __iter__(self): ❶
...         self.x = 0 ❷
...         return self ❸
...     def __next__(self): ❹
...         if self.x > 2:
...             raise StopIteration ❺
...         else:
...             self.x += 1
...             return self.x ❻
>>> for i in A():
...     print(i)
```



1  
2  
3

- ① An *iterator* must implement the `__iter__()` special method.
- ② Initialize some state to the “starting” state.
- ③ The `__iter__()` special method must return an *iterable*, i.e., an object that implements the `__next__()` special method. In this case, the same instance, because A itself also implements the `__next__()` special method.
- ④ The `__next__()` method is defined. This will be called for every step in the iteration sequence until...
- ⑤ `...StopIteration` is raised.
- ⑥ Here the *returned values* for each iteration are generated.

Now you ask the question: what happens if you declare the `__next__()` special method as an `async def` coroutine function? That will allow it to `await` some kind of I/O-bound operation; and this is pretty much exactly

how `async for` works, except for some small details around naming. The specification (in PEP 492) shows that to use `async for` on an async iterator, several things are required in the async iterator itself:

You must implement `def __aiter__()` (Note: *not* with `async def`!)

`__aiter__()` must return an object that implements `def __anext__()`

`__anext__()` must return a value for each iteration, and must raise `StopAsyncIteration` when finished

Let's take a quick look at how that might work. Imagine that we have a bunch of keys in a Redis database, and we want to iterate over their data, but we only fetch the data on-demand. An asynchronous iterator for that might look something like the following:

```
import asyncio
from aioredis import create_redis

async def main(): ❶
    redis = await create_redis(('localhost', 6379)) ❷
    keys = ['Americas', 'Africa', 'Europe', 'Asia'] ❸

    async for value in OneAtATime(redis, keys): ❹
        await do_something_with(value) ❺

class OneAtATime:
    def __init__(self, redis, keys): ❻
```

```

        self.redis = redis
        self.keys = keys
    def __aiter__(self): ❶
        self.ikeys = iter(self.keys)
        return self
    async def __anext__(self): ❷
        try:
            k = next(self.ikeys) ❸
        except StopIteration: ❹
            raise StopAsyncIteration

        value = await redis.get(k) ❺
        return value

asyncio.get_event_loop().run_until_complete(main())

```

- ❶ The main function: we run it using `run_until_complete()` towards the bottom of the code sample.
- ❷ Use the high-level interface in `aioredis` to get a connection.
- ❸ Imagine that each of the values associated with these keys is quite large, and stored in the Redis instance.
- ❹ Here we're using `async for`: the point is that *iteration itself* is able to suspend itself while waiting for the next datum to arrive.

- 5 For completeness, imagine that we also perform some I/O-bound activity on the fetched value; perhaps a simple data transformation and then it gets sent on to another destination.
- 6 The initializer of this class is quite ordinary: we store the Redis connection instance and the list of keys to iterate over.
- 7 Just as in the previous code example with `__iter__()`, we use `__aiter__()` to set things up for iteration. We create a normal iterator over the keys, `self.ikeys`, and return `self` because `OneAtATime` also implements the `__anext__()` coroutine method.
- 8 Note that the `__anext__()` method is declared with `async def`, while the `__aiter__()` method is declared only with `def`.
- 9 For each key, fetch the value from Redis: `self.ikeys` is a regular iterator over the keys, so we use `next()` to move over them.
- 10 When `self.ikeys` is exhausted, handle the `StopIteration` and simply turn it into a `StopAsyncIteration`! This is how

you signal stop from inside an async iterator.

- ⑪ Finally—the entire point of this example—we can get the data from Redis associated with this key. We can `await` the data, which means that other code can run on the event loop while we wait on network I/O.

Hopefully this example is clear: `async for` provides the ability to retain the convenience of a simple for-loop, even when iterating over data where the iteration itself is performing I/O. The benefit is that you could process enormous amounts of data, all with a single loop, because you only have to deal with each chunk in tiny batches.

## Async Generators: `yield` Inside `async def` Functions

Async generators answer the question, “What happens if you use `yield` inside a native `async def` coroutine function?” This concept might be confusing if you have some experience with using generators *as if* they were coroutines, such as with the *Twisted* framework, or the *Tornado* framework, or even with `yield from` in Python 3.4’s *asyncio*.

Therefore, before we continue in this section, it is best if you can convince yourself that:

Coroutines and generators are completely different concepts.

Async generators behave much like ordinary generators.

For iteration, you use `async for` for async generators, instead of ordinary `for` for ordinary generators.

The example used in the previous section to demonstrate an async iterator for interaction with Redis turns out to be much simpler if we set it up as an async generator:

*Example 3-14. Easier with an async generator*

---

```
import asyncio
from aioredis import create_redis

async def main(): ❶
    redis = await create_redis(('localhost', 6379))
    keys = ['Americas', 'Africa', 'Europe', 'Asia']

    async for value in one_at_a_time(redis, keys): ❷
        await do_something_with(value)

async def one_at_a_time(redis, keys): ❸
    for k in keys:
        value = await redis.get(k) ❹
        yield value ❺

asyncio.get_event_loop().run_until_complete(main())
```

❶ The `main()` function is identical to how it was in the code in “Async Iterators: async for”.

- ② Well, almost identical: I had to change the name from camel case to snake case.
- ③ Our function is now declared with `async def`, making it a *coroutine function*, and since this function also contains the `yield` keyword, we refer to it as an *asynchronous generator function*.
- ④ We don't have to do the convoluted things necessary in the previous example with `self.ikeys`: here, just loop over the keys directly and obtain the value...
- ⑤ ...and then yield it to the caller, just like a normal generator.

It might seem very complex if this is new to you, but I urge you to play around with this yourself on a few toy examples. It starts to feel natural pretty quickly. Async generators are likely to become very popular in `asyncio`-based codebases because they bring all the same benefits as normal generators: making code shorter and simpler.

## Async Comprehensions

Now that we've seen how Python supports asynchronous iteration, the next natural question to ask is whether it also works for list comprehensions—and

the answer is yes! This support was introduced in [PEP 530](#) and I recommend you check out the PEP yourself if possible. It is quite short and readable.

### *Example 3-15. Async list, dict, and set comprehensions*

```
>>> import asyncio
>>> async def doubler(n):
...     for i in range(n):
...         yield i, i * 2 ❶
...         await asyncio.sleep(0.1) ❷
>>> async def main():
...     result = [x async for x in doubler(3)] ❸
...     print(result)
...
...     result = {x: y async for x, y in doubler(3)} ❹
...     print(result)
...
...     result = {x async for x in doubler(3)} ❺
...     print(result)
>>> asyncio.get_event_loop().run_until_complete(main())
[(0, 0), (1, 2), (2, 4)]
{0: 0, 1: 2, 2: 4}
{(1, 2), (0, 0), (2, 4)}
```

❶ `doubler()` is a very simple async generator: given an upper value, it'll iterate over a simple range, yielding a tuple of the value, and its double.

❷ Sleep a little, just to emphasize that this is really an async function.



- ③ An async list comprehension: note how `async for` is used instead of the usual `for`. This difference is the same as that shown earlier with “Async Iterators: `async for`”.
- ④ Async dict comprehension; all the usual tricks work, such as unpacking the tuple into `x` and `y` so that they can feed the dict comprehension syntax.
- ⑤ The async set comprehension works exactly as you would expect.

The other side of the coin, as the PEP 530 outlines, is using `await` inside comprehensions. But this is not really that special: `await <coro>` is a normal expression and can be used in most places you would expect.

It is the `async for` that makes a comprehension an *async comprehension*, not the presence of `await`. All you need for `await` to be legal (inside a comprehension) is that you’re inside the body of a coroutine function, i.e., a function declared with `async def`. So even though using `await` and `async for` inside the same list comprehension is really combining two separate concepts, let’s do it anyway to continue the desensitization process of becoming comfortable with async language syntax:

*Example 3-16. Putting it all together*

---

```

>>> import asyncio
>>> async def f(x): ❶
...     await asyncio.sleep(0.1)
...     return x + 100
>>> async def factory(n): ❷
...     for x in range(n):
...         await asyncio.sleep(0.1)
...         yield f, x ❸
>>> async def main():
...     results = [await f(x) async for f, x in factory(3)]
...     ❹
...     print('results = ', results)
>>> asyncio.get_event_loop().run_until_complete(main())
results = [100, 101, 102]

```

- ❶ Very simple coroutine function: sleep for a bit, then return the parameter plus 100.
- ❷ This is an *async generator*, which we will call inside an async list comprehension a bit further down, using `async for` to drive the iteration.
- ❸ The async generator will yield a tuple of `f` and the iteration var `x`. The `f` return value is a *coroutine function*, not yet a coroutine.
- ❹ Finally, the async comprehension. This example has been contrived

to demonstrate a comprehension that includes *both* `async for` as well as `await`. Let's break down what is happening inside the comprehension: the `factory(3)` call returns an async generator, which must be driven by iteration. Because it's an *async* generator, you can't just use `for`; you must use `async for`.

Then, the values produced by the async generator are a tuple of a *coroutine function* `f`, and an `int`. Calling the coroutine function `f()` produces a *coroutine*, which must be evaluated with `await`.

Note that inside the comprehension, the use of `await` has nothing at all to do with the use of `async for`: they are doing completely different things and acting on different objects entirely.

## Starting Up and Shutting Down (Gracefully!)

Most async-based programs are going to be long-running, network-based applications. This domain holds a surprising amount of complexity in dealing with how to start up and shut down correctly.

Of the two, startup is simpler. The standard way of starting up an `asyncio` application is to create a task, and then call `loop.run_forever()`, as shown in the Hello World example in the *Quickstart* section.

The one exception might be when you need to start a listening server. Startup is then typically a two-stage process:

First, create a coroutine for *only* the “starting up” phase of the server, and then use `run_until_complete()` on that initialization coroutine to start the server itself.

Second, continue with the usual “main” part of the application by calling `loop.run_forever()`.

Generally, startup will be fairly straightforward; and for the server case described above, you can read more about [it in the docs](#). We’ll also briefly look at a demonstration of such server startup in an upcoming code example.

Shutdown is much more intricate.

For shutdown, we previously covered the dance that follows when something stops the event loop. When a running loop is stopped, the `run_forever()` call is unblocked, and code that appears after it is executed. At this point you have to:

- collect all the still-pending task objects (if any)

- cancel these tasks (this raises `CancelledError` inside each running coroutine, which you may choose to handle in a `try/except` within the body of the coroutine function)

- gather all these tasks into a “group” task

use `run_until_complete()` on the “group” task to wait for them to finish, i.e., let the `CancelledError` exception be raised and dealt with

Only then is shutdown complete.

A rite of passage in building your first few `asyncio` apps is going to be trying to get rid of error messages like *Task was destroyed but it is pending!* during shutdown. This happens because one or more of the above steps was missed. Here’s an example of the error:

#### *Example 3-17. Destroyer of pending tasks*

---

```
# taskwarning.py
import asyncio

async def f(delay):
    await asyncio.sleep(delay)

loop = asyncio.get_event_loop()
t1 = loop.create_task(f(1)) ❶
t2 = loop.create_task(f(2)) ❷
loop.run_until_complete(t1) ❸
loop.close()
```

❶ Task 1 will run for 1 second.

❷ Task 2 will run for 2 seconds.

③ Only run until task 1 is complete.

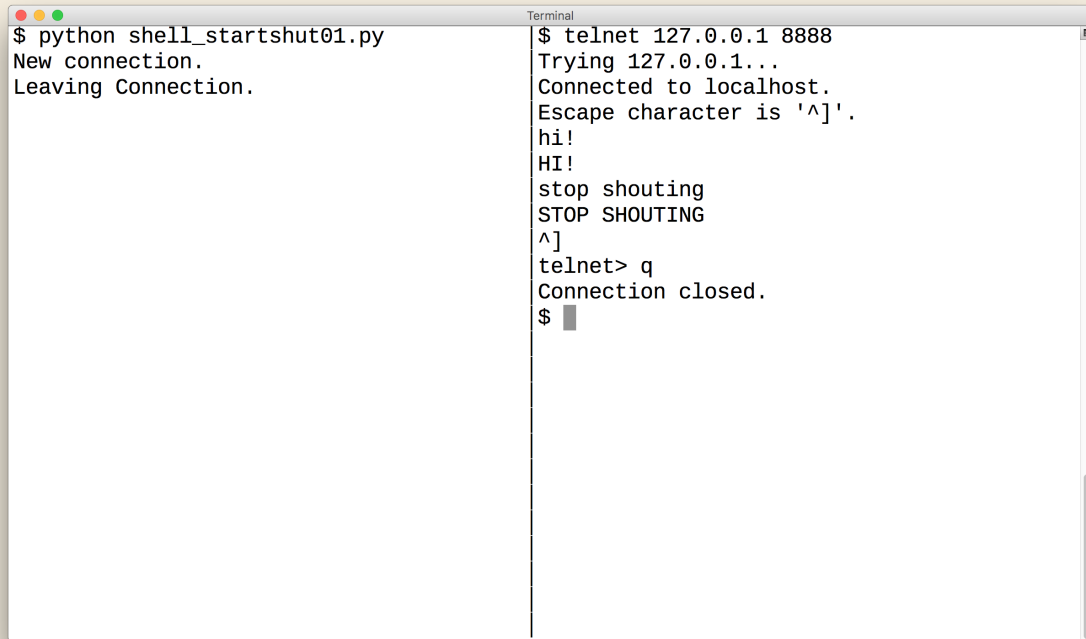
Output:

```
$ python taskwarning.py
Task was destroyed but it is pending!
task: <Task pending coro=<f() done, defined at [...snip...]
>>
```

This error is telling you that some tasks had not yet been completed when the loop was closed. We want to avoid this, and that is why the idiomatic shutdown procedure is to collect all unfinished tasks, cancel them, and then let them all finish up *before* closing the loop.

Let's look at a more detailed code sample than the *Quickstart* example, and look at these phases again. We'll do so via a mini-case-study with a telnet-based echo server.

In [Figure 3-1](#), the server is started on the left-hand pane. Then, on the right, a telnet session connects to the server. We type a few commands that are fed back to us (in ALL-CAPS) and then we disconnect. Now we take a look at the code that drives this program.

A terminal window with two panes. The left pane shows a Python script execution: '\$ python shell\_startshut01.py', 'New connection.', and 'Leaving Connection.'. The right pane shows a telnet session: '\$ telnet 127.0.0.1 8888', 'Trying 127.0.0.1...', 'Connected to localhost.', 'Escape character is '^]'.', 'hi!', 'HI!', 'stop shouting', 'STOP SHOUTING', '^]', 'telnet> q', 'Connection closed.', '\$'.

```
$ python shell_startshut01.py
New connection.
Leaving Connection.

$ telnet 127.0.0.1 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hi!
HI!
stop shouting
STOP SHOUTING
^]
telnet> q
Connection closed.
$
```

Figure 3-1. Telnet interaction with an echo server

Example 3-18. Asyncio application life cycle (based on the TCP echo server in the Python Documentation)

```
from asyncio import ( ❶
    get_event_loop, start_server, CancelledError,
    StreamReader, StreamWriter, Task, gather)

async def echo(reader: StreamReader, writer: StreamWriter)
: ❷
    print('New connection.')
    try:
        while True: ❸
            data: bytes = await reader.readline() ❹
            if data in [b'', b'quit']:
                break
```

```

        writer.write(data.upper()) ❸
        await writer.drain()
        print('Leaving Connection.')
    # except CancelledError: ❹
    #     writer.write_eof()
    #     print('Cancelled')
    finally:
        writer.close()

loop = get_event_loop()
coro = start_server(echo, '127.0.0.1', 8888, loop=loop) ❺
server = loop.run_until_complete(coro) ❻

try:
    loop.run_forever() ❼
except KeyboardInterrupt:
    print('Shutting down!')

server.close() ❽
loop.run_until_complete(server.wait_closed()) ❾

tasks = Task.all_tasks() ❿
for t in tasks:
    t.cancel()
group = gather(*tasks, return_exceptions=True) ⓫
loop.run_until_complete(group) ⓬
loop.close()

```

❶ Import a bunch of things from the `asyncio` namespace—I need to control line-length for this book format!

❷ This `echo()` coroutine function will be used (by the server) to



create a coroutine for every connection made. This is using the *streams* API for networking with `asyncio`.

③ To keep the connection alive, we'll have an infinite loop to wait for messages.

④ Wait for a line of data from the other side.

⑤ Return the data back to the sender, but in ALL-CAPS.

⑥ I intentionally commented this block out to draw attention to what happens if your coroutines do not properly handle cancellation. If you *do* handle `CancelledError`, you get a chance to do any cleanup specific to a shutdown scenario.

⑦ This is the *startup* phase of the program: the server requires a separate step from the “main” `run_forever()` stage. The `start_server()` function will return a coroutine, and that must be `run_until_complete()`. Note how our `echo` coroutine function is passed in: it will be used like a factory that produces a new coroutine for every new connection.

⑧ Run the coroutine to start the TCP server.

- ⑨ Only now do we start the main, “listening” part of the program. From this point on, every TCP connection that gets made to our server will spawn a coroutine from our `echo` coroutine function. The only thing that can stop our event loop is a `KeyboardInterrupt`, which is the same as a `SIGINT` on Unix systems. (In a production system, you would use specific signal handlers rather than `KeyboardInterrupt`; this is presented later in “Signals”.)
- ⑩ To reach here, we know that shutdown has been initiated, e.g., with Ctrl-C. The first thing that must be done is to prevent our server from accepting any more new connections. This takes two steps: first call `server.close()`...
- ⑪ ...and then run `wait_closed()` on the loop to close any sockets that are still waiting for connections, but which do not yet have connections. Note that any active coroutines (e.g., spawned from `echo()`) with active connections will *not* be affected: those connections remain up.
- ⑫ Now we can start shutting down tasks. The idiomatic sequence is to collect all the currently pending tasks on the event loop, cancel them, and then gather them into a single group task.

13 Take note of this parameter `return_exceptions`. It is pretty important and discussed separately in the next section.

14 As before, run the group task to completion.

Hopefully you're starting to see a familiar pattern emerge.

## TIP

One last point before we move on: If you catch `CancelledError` inside a coroutine, you want to be careful to not create any new tasks inside the exception handler.

Here's why: the `all_tasks()` call at callout 12 is not going to be aware of any new tasks created during the `run_until_complete()` phases at callout 14—which is when the code inside your cancellation handlers will actually execute. So the rule is: no new tasks inside `CancelledError` exception handlers, unless you also `await` them within a `CancelledError` exception handler.

And remember: if you're using a *library* or *framework*, make sure to follow

their documentation on how you should perform startup and shutdown. Third-party frameworks usually provide their own functions for startup and shutdown, and they'll provide event hooks for customization. You can see an example of these hooks with the *Sanic* framework in “Case Study: Cache Invalidation”, later in the book.

## What Is the `return_exceptions=True` for?

You may have noticed that I set the keyword argument `return_exceptions=True` in the call to `gather()` at callout 13 in the previous code sample. I also did it earlier in “Quickstart”, and I very sneakily said nothing at the time. Its moment has arrived.

The default is `gather(..., return_exceptions=False)`. This default is problematic for our *shutdown* process. It's a little complicated to explain directly; instead, let's step through a sequence of facts that'll make it much easier to understand:

`run_until_complete()` operates on a future; during shutdown, it's the future returned by `gather`.

If that future raises an exception, the exception will *also* be raised out of `run_until_complete()`, which means that the loop will stop.

If `run_until_complete()` is being used on a “group” future, any exception raised inside *any of the subtasks* will also be raised in the “group”

future if it isn't handled in the subtask. Note this includes `CancelledError`.

If only some tasks handle `CancelledError`, and others don't, the ones that don't will cause the loop to stop, as above. This means that the loop will be stopped *before* all the tasks are done.

For shutdown mode, we really don't want this behavior. We want `run_until_complete()` to finish only when all the tasks in the group really do finish, regardless of whether some of the tasks raise exceptions or not.

Hence we have `gather(*, return_exceptions=True)`: that setting makes the “group” future treat exceptions from the subtasks as *returned values*, so that they don't bubble out and interfere with `run_until_complete()`.

And there you have it: the relationship between `return_exceptions=True` and `run_until_complete()`!

An undesirable consequence of capturing exceptions in this way is that some errors may escape your attention because they're now (effectively) being handled inside the “group” task. If this is a concern, you can obtain the output from `run_until_complete()` and scan it for any subclasses of `Exception`; and then write log messages appropriate for your situation.

Here's a quick look at what you get:

*Example 3-19. All the tasks will complete*

---

```
import asyncio

async def f(delay):
    await asyncio.sleep(1 / delay) ❶
    return delay

loop = asyncio.get_event_loop()
for i in range(10):
    loop.create_task(f(i))
pending = asyncio.Task.all_tasks()
group = asyncio.gather(*pending, return_exceptions=True)
results = loop.run_until_complete(group)
print(f'Results: {results}')
loop.close()
```

❶ It would be awful if someone were to pass in a zero...

Output:

```
$ python alltaskscomplete.py
Results: [6, 9, 3, 7, ...
         ZeroDivisionError('division by zero',), 4, ...
         8, 1, 5, 2]
```

Without `return_exceptions=True`, the `ZeroDivisionError`

would be raised from `run_until_complete()`, stopping the loop and thus preventing the other tasks from finishing.

In the next section we look at handling signals (beyond `KeyboardInterrupt`), but before we get there, it's worth keeping in mind that graceful shutdown is one of the more difficult aspects of network programming, and this remains true for `asyncio`. The information in this section is merely a start. I encourage you to have specific tests for clean shutdown in your own automated test suites. Different applications often require different strategies.

I've published a tiny package on the Python package index, [`aiorun`](#), primarily for my own experiments and education in dealing with shutdown, and it incorporates many ideas from this section. It may also be useful for you to tinker with the code and experiment with your own ideas around `asyncio` shutdown scenarios.

## Signals

In the previous example we showed how the event loop is stopped with a `KeyboardInterrupt`, i.e., pressing Ctrl-C. The raised `KeyboardInterrupt` effectively unblocks the `run_forever()` call and allows the subsequent shutdown sequence to happen.

`KeyboardInterrupt` corresponds to the `SIGINT` signal. In network

services, the more common signal for process termination is actually `SIGTERM`, and this is also the default signal when you use the `kill()` command in a UNIX shell.

## TIP

The `kill()` command on UNIX systems is deceptively named: all it does is send signals to a process! Without arguments, `$ kill <PID>` will send a `TERM` signal: your process can receive the signal and do a graceful shutdown, or simply ignore it! That's a bad idea though, because if your process doesn't stop eventually, the next thing the killer usually does is `$ kill -s KILL <PID>` which sends the `KILL` signal. This will shut you down and there's nothing your program can do about it.

`asyncio` has built-in support for handling process signals, but there's a surprising degree of complexity around signal handling in general (not specific to `asyncio`). We cannot cover everything, but we can have a look at some of the more basic considerations that need to be made. The next code sample will produce the following output:

```
$ python shell_signal01.py
<Your app is running>
<Your app is running>
```



```
<Your app is running>
<Your app is running>
^CGot signal: SIGINT, shutting down.
```

I pressed Ctrl-C to stop the program, visible on the last line. Here is the code:

*Example 3-20. Refresher for using KeyboardInterrupt as a SIGINT handler*

```
# shell_signal01.py
import asyncio

async def main(): ❶
    while True:
        print('<Your app is running>')
        await asyncio.sleep(1)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.create_task(main()) ❷
    try:
        loop.run_forever()
    except KeyboardInterrupt: ❸
        print('Got signal: SIGINT, shutting down.')
    tasks = asyncio.Task.all_tasks()
    for t in tasks:
        t.cancel()
    group = asyncio.gather(*tasks, return_exceptions=True)
    loop.run_until_complete(group)
    loop.close()
```

This is the main part of our application. To keep things simple we're just going to sleep in an infinite loop.

- ② This startup and shutdown sequence will be familiar to you from the previous section. We schedule `main()`, call `run_forever()`, and wait for something to stop the loop.
- ③ In this case, only Ctrl-C will stop the loop, and then we handle `KeyboardInterrupt` and do all the necessary cleanup bits as we've covered in the previous sections.

So far that's pretty straightforward. Now we're going to complicate things:

One of your colleagues asks that you please handle `SIGTERM` in addition to `SIGINT` as a shutdown signal.

In your real application, you need to *handle* `CancelledError` inside your `main()` coroutine, and the cleanup code inside the exception handler is going to take several seconds to finish (imagine that you have to communicate with network peers and close a bunch of socket connections).

Your app must not do weird things if you're sent signals multiple times; after you receive the first shutdown signal, you want to simply ignore any new signals until exit.

`asyncio` provides enough granularity in the API to handle all these situations. I've modified our simple code example above to introduce these points:

*Example 3-21. Handling both `SIGINT` and `SIGTERM`, but stop the loop only once*

```
# shell_signal02.py
import asyncio
from signal import SIGINT, SIGTERM ❶

async def main():
    try:
        while True:
            print('<Your app is running>')
            await asyncio.sleep(1)
    except asyncio.CancelledError: ❷
        for i in range(3):
            print('<Your app is shutting down...>')
            await asyncio.sleep(1)

def handler(sig): ❸
    loop.stop() ❹
    print(f'Got signal: {sig!s}, shutting down.')
    loop.remove_signal_handler(SIGTERM) ❺
    loop.add_signal_handler(SIGINT, lambda: None) ❻

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    for sig in (SIGTERM, SIGINT): ❼
        loop.add_signal_handler(sig, handler, sig)
    loop.create_task(main())
    loop.run_forever() ❽
```

```
tasks = asyncio.Task.all_tasks()
for t in tasks:
    t.cancel()
group = asyncio.gather(*tasks, return_exceptions=True)
loop.run_until_complete(group)
loop.close()
```

- ① Import the signal values from the standard library `signal` module.
- ② This time, our `main()` coroutine is going to do some cleanup internally. When the cancellation signal is received (initiated by cancelling each of the tasks), there will be a period of three seconds while `main()` will continue running during the “`run_until_complete()`” phase of the shutdown process. It’ll print, “Your app is shutting down...”.
- ③ This is a callback handler for when we receive a signal. It is configured on the loop via the call to `add_signal_handler()` a bit further down.
- ④ The primary purpose of the handler is to stop the loop: this will unblock the `loop.run_forever()` call and allow pending-task collection, cancellation, and the `run_complete()` for shutdown.

- 5 Since we are now in a shutdown mode, we *don't want* another SIGINT or SIGTERM to trigger this handler again: that would call `loop.stop` during the `run_until_complete()` phase which would interfere with our shutdown process. Therefore, we *remove* the signal handler for SIGTERM from the loop.
- 6 Gotcha time! We can't simply remove the handler for SIGINT, because in that case `KeyboardInterrupt` will *again* become the handler for SIGINT, the same as it was before we added our own handlers! Instead, we set an empty `lambda` function as the handler. This means that `KeyboardInterrupt` stays away, and SIGINT (and Ctrl-C) has no effect.<sup>14</sup>
- 7 Here the signal handlers are attached to the loop. Note that, as discussed above, by setting a handler on SIGINT, a `KeyboardInterrupt` will no longer be raised on SIGINT. The raising of a `KeyboardInterrupt` is the “default” handler for SIGINT and is preconfigured in Python until you do something to change the handler, as we're doing here.
- 8 As usual, execution blocks on `run_forever()` until something stops the loop. In this case, the loop will be stopped inside `handler()` if either SIGINT or SIGTERM is sent to our process.

The remainder of the code is the same as before.

Output:

```
$ python shell_signal02.py
<Your app is running>
<Your app is running>
<Your app is running>
<Your app is running>
<Your app is running>
^CGot signal: Signals.SIGINT, shutting down.
<Your app is shutting down...>
^C<Your app is shutting down...> ❶
^C<Your app is shutting down...>
```

- ❶ I hit Ctrl-C a bunch of times during the shutdown phase, but as expected, nothing happened until the `main()` coroutine eventually completed.

## Waiting for the Executor During Shutdown

Recall in “Quickstart”, we introduced the basic executor interface with [Example 3-2](#). In [Example 3-2](#), we pointed out that the blocking `time.sleep()` call was shorter than the `asyncio.sleep()` call—luckily for us—because it means the executor task completes sooner than the `main()` coroutine, and as a result the program shuts down correctly.

In this section, we examine what happens during shutdown when executor jobs take longer to finish than all the pending `Task` instances. The short answer is: without intervention, you’re going to get these kinds of errors:

*Example 3-22. The executor takes too long to finish*

```
# quickstart.py
import time
import asyncio

async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()

def blocking():
    time.sleep(1.5) ❶
    print(f"{time.ctime()} Hello from a thread!")

loop = asyncio.get_event_loop()

loop.create_task(main())
loop.run_in_executor(None, blocking)
loop.run_forever()
tasks = asyncio.Task.all_tasks(loop=loop)
group = asyncio.gather(*tasks, return_exceptions=True)
loop.run_until_complete(group)
loop.close()
```

❶ This code sample is exactly the same as the one in “Quickstart”,

*except* that the sleep time in the blocking function is now longer than the async one.

### *Example 3-23. Output*

---

```
Fri Sep 15 16:25:08 2017 Hello!
Fri Sep 15 16:25:09 2017 Goodbye!
exception calling callback for <Future at [...snip...]>
Traceback (most recent call last):

<big nasty traceback>

RuntimeError: Event loop is closed
Fri Sep 15 16:25:09 2017 Hello from a thread!
```

What’s happening here is that behind the scenes, `run_in_executor()` does *not* create a `Task` instance: it returns a `Future`. That means it isn’t included in the set of “active tasks” returned from `asyncio.Task.all_tasks()`, and therefore `run_until_complete()` does *not* wait for the executor task to finish!

There are three ideas for fixing this that spring to mind, all with different trade-offs, and we’re going to look at each of them. The real goal of this exercise is to help you think about the event loop life cycle from different points of view, and think about lifetime management of all your coroutines, threads, and sub-processes that might all be interoperating in a non-trivial program.



The first idea, and the easiest to implement is to always `await` an executor task from inside a coroutine:

*Example 3-24. Option A: wrap the executor call inside a coroutine*

---

```
# quickstart.py
import time
import asyncio

async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()

def blocking():
    time.sleep(2.0)
    print(f"{time.ctime()} Hello from a thread!")

async def run_blocking(): ❶
    await loop.run_in_executor(None, blocking)

loop = asyncio.get_event_loop()
loop.create_task(main())
loop.create_task(run_blocking()) ❷
loop.run_forever()
tasks = asyncio.Task.all_tasks(loop=loop)
group = asyncio.gather(*tasks, return_exceptions=False)
loop.run_until_complete(group)
loop.close()
```

❶ The idea aims at fixing the shortcoming that

`run_in_executor()` returns only a `Future` instance and not a task. We can't capture the job in `all_tasks()`, but we *can* use `await` on the future. To do this we create this new coroutine function `run_blocking()`, and inside we do nothing more than `await` the result of the `run_in_executor()` call. This new coroutine function `run_blocking()` *will* be scheduled in a `Task` and so our shutdown process will include it in the group.

- ② To run the new `run_blocking()` coroutine function, we use `create_task` *exactly the same* as how the `main()` coroutine was scheduled to run.

The code above looks great, except for one thing: it can't handle cancellation! If you look carefully, you'll see I've omitted the task cancellation loop that appeared in previous examples. If you put it back, we get the "Event loop is closed" errors as before. We can't even handle `CancelledError` inside `run_blocking()` to try to re-await the future. No matter what we try, the *task* that wraps `run_blocking()` would get cancelled, but the *executor* job will run until its internal `time.sleep()` completes. Let's move on to the next idea.

The next idea, below, is a little more cunning. The strategy is to collect the pending tasks, then cancel *only them*, but before we call `run_until_complete()`, we add in the future from the

```
run_in_executor().
```

*Example 3-25. Option B: add the executor Future to the gathered tasks*

---

```
# quickstart.py
import time
import asyncio

async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()

def blocking():
    time.sleep(2.0)
    print(f"{time.ctime()} Hello from a thread!")

loop = asyncio.get_event_loop()
loop.create_task(main())
future = loop.run_in_executor(None, blocking) ❶
loop.run_forever()
tasks = asyncio.Task.all_tasks(loop=loop) ❷
for t in tasks:
    t.cancel() ❸
group_tasks = asyncio.gather(*tasks, return_exceptions=True)
group = asyncio.gather(group_tasks, future) ❹
loop.run_until_complete(group)
loop.close()
```

❶ Just like the original blocking example, we call

`run_in_executor()` directly, but we make sure to assign the result to identifier `future`. This identifier will be used shortly.

- ② The loop has stopped, and now we're in the shutdown phase. First, get all the tasks. Note that this does not include the executor job, since `run_in_executor()` doesn't create a task.
- ③ Cancel the tasks.
- ④ This is the trick: we create a *new* group that combines the tasks and the executor future. This way, the executor will finish normally, while the tasks still undergo the usual cancellation process.

This solution is more well-behaved during shutdown, but it still has flaws. In general, it'll be quite inconvenient to somehow collect all the executor futures throughout your entire program into a collection that you can use during the shutdown sequence; then create a tasks-plus-futures group; and then run that till completion. It works, but there's a better way.

*Example 3-26. Option C: use your own executor and wait*

---

```
# quickstart.py
import time
import asyncio
from concurrent.futures import ThreadPoolExecutor as Execu
```

tor

```
async def main():
    print(f'{time.ctime()} Hello!')
    await asyncio.sleep(1.0)
    print(f'{time.ctime()} Goodbye!')
    loop.stop()

def blocking():
    time.sleep(2.0)
    print(f'{time.ctime()} Hello from a thread!')

loop = asyncio.get_event_loop()
executor = Executor() ❶
loop.set_default_executor(executor) ❷
loop.create_task(main())
future = loop.run_in_executor(None, blocking) ❸
loop.run_forever()
tasks = asyncio.Task.all_tasks(loop=loop)
for t in tasks:
    t.cancel()
group = asyncio.gather(*tasks, return_exceptions=True)
loop.run_until_complete(group)
executor.shutdown(wait=True) ❹
loop.close()
```

❶ This time, we create our own executor instance.

❷ You have to set your custom executor as the default one for the loop. This means that anywhere the code calls `run_in_executor()`, it'll be using your custom instance.

- ③ As before, run the blocking function.
- ④ Finally, we can explicitly wait for all the executor jobs to finish before closing the loop. This will avoid the “Event loop is closed” messages that we saw before. We can do this because we have access to the executor object; the default executor is not exposed in the `asyncio` API, which is why we cannot call `shutdown` on it.

Finally we have a strategy with general applicability: you can call `run_in_executor( )` *anywhere*, and your program will still shut down cleanly.

I strongly urge you to experiment with the code examples shown here and try different strategies to create tasks and executor jobs, stagger them in time, and try to shut down cleanly.

## Testing with asyncio

In an earlier section, I said that you don’t have to pass a loop variable around your program (if you need access to the loop), and you can simply use `asyncio.get_event_loop( )` to obtain the loop wherever it is needed.

However, when writing *unit tests*, you’re going to have to work with multiple events loops because each unit test will typically require a new loop

instance. The question will again arise: should you write code that requires event loops to be passed around in function parameters, in order to support unit testing?

Let's look at an example using the wonderful `pytest` testing framework. First, let's look at the function we want to test:

*Example 3-27. Sometimes you need to use `call_soon()`*

```
import asyncio
from typing import Callable

async def f(notify: Callable[[str], None]): ❶
    # <...some code...>
    loop = asyncio.get_event_loop() ❷
    loop.call_soon(notify, 'Alert!') ❸
    # <...some code...>
```

- ❶ Imagine a coroutine function `f` inside which it is necessary to use `loop.call_soon()` to fire off another alerting function. (It might do logging, write messages into Slack, short stocks, or anything else you can think of!)
- ❷ In this function, we did not receive the loop via the function parameters, so it is obtained via `get_event_loop()`; remember, this call always returns the loop that is associated with the current

thread.

③ A basic alert call.

The best way to throw pytest at your asyncio code is to provide an event loop via a fixture. Pytest injects these into each of your tests as a function parameter. It sounds more complicated than it is, so here is a simple fixture for providing an event loop:

*Example 3-28. Pytest fixture for providing an event loop—with a bug!*

---

```
# conftest.py ①
import pytest

@pytest.fixture(scope='function') ②
def loop():
    loop = asyncio.new_event_loop() ③
    try:
        yield loop
    finally:
        loop.close() ④
```

① Pytest, by convention, will automatically import and make available to all your test modules everything defined in a file called *conftest.py*.



- ② This creates a fixture. The “scope” argument tells Pytest when the fixture should be finalized, and a new one made. For “function” scope, as above, every single test will have a new loop made.
- ③ Create a completely new loop instance. Note that we don’t start the loop running. We leave that to each test in which this fixture will be used.
- ④ When the fixture is being finalized (in our case, after each test), close the loop.

The code above causes a problem, so don’t use it. The problem is subtle, but it is also the entire point of this section, so we’ll discuss it in detail shortly. First let’s look at the tests for coroutine function `f`:

*Example 3-29. The tests*

---

```
from somewhere import f ①
...

def test_f(loop): ②
    ...

    collected_msgs = [] ③

    def dummy_notify(msg): ④
        ...
        collected_msgs.append(msg)

    loop.create_task(f(dummy_notify)) ⑤
    ...
```

```

loop.call_later(1, loop.stop) ❸
loop.run_forever()

assert collected_msgs[0] == 'Alert!' ❹

```

- ❶ Treat this code sample as pseudocode—there is no `somewhere` module! Here, `f` refers to the coroutine function defined further up.
- ❷ Pytest will recognize the “loop” argument here, find that name in the list of fixtures, and then call our `test_f()` above *with* that new loop instance yielded out of the fixture.
- ❸ We’re going to pass a “dummy” `notify()` callback to `f`. This list will collect all of the alerts received (and our test checks that we got the right messages back).
- ❹ This is the fake `notify()` function required by `f`.
- ❺ Schedule a coroutine from `f`, passing our fake `notify()` callback.
- ❻ We’re using `run_forever()` to run the loop, so this makes sure the loop will actually stop. Alternatively, we might have used `run_until_complete(f(...))` without a `call_later()`;

but which you choose depends on what you're testing. For testing side effects, as is the case here, I find it simpler to use `run_forever()`. On the other hand, when you're testing *return values* from coroutines, it's better to use `run_until_complete()`.<sup>15</sup>

## 7 Finally, the test!

Remember when I said we had a problem? Here it is: inside coroutine function `f`, the loop instance returned from `get_event_loop()` is *different* from the event loop that was provided to us by the fixture. The test will fail because of this issue; in fact the loop from `get_event_loop()` will never even have started running.

One solution to this dilemma is to add an explicit `loop` parameter to the coroutine function `f`. Then you can always be sure that the correct loop instance is being used. In practice, however, this can be very painful, because large codebases will require a huge number of functions to pass the loop around.

There is a better way: when a new loop is in play, set *that loop* to be “the” loop associated with the current thread, which means that every call to `get_event_loop()` will return the new one. This works very well for tests. All we need to do is to modify our pytest fixture slightly:

### Example 3-30. A more useful loop fixture for pytesting

```
# conftest.py
import pytest

@pytest.fixture(scope='function')
def loop():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop) ❶
    try:
        yield loop
    finally:
        loop.close()
```

- ❶ The magic line: once `set_event_loop()` is called, every subsequent call to `asyncio.get_event_loop()` will return the loop instance created in this fixture, and you don't have to explicitly pass the loop instance through all your function calls.

We used quite a few pages to end up with an important one-line fix, but it was necessary to explain *why* the line was important. As in previous sections, my goal is not to tell you the “one true way,” but rather to guide your thinking as you make sense of how to incorporate `asyncio` into your toolbox.

1

<http://bit.ly/2EPys9Q>

2

When they become available! At the time of writing, the only available references for Asyncio were the API specification in the official Python documentation and a collection of blog posts, most of which have been linked in this book.

3

The `asyncio` API lets you do lots of wild things with multiple loop instances and threads, but this is not the right book to get into that. 99% of the time you're only going to use a single, main thread for your app, as shown here.

4

One other point: you'll notice in the code line above that the parameter in the function call to `create_task()` is `coro`. This is a convention in most of the API documentation you'll find, and it refers to a *coroutine*; i.e., strictly speaking, the *result* of calling an `async def` function, and *not* the function itself.

5

Unfortunately, the first parameter of `run_in_executor()` is the `Executor` instance to use, and you *must* pass `None` in order to use the default. Every time I use this it feels like the “executor” parameter is crying out to be a kwarg with a default value of `None`.

6

And furthermore, this is how other open source libraries such as Twisted and Tornado have exposed async support in the past.

7

<https://www.python.org/dev/peps/pep-0492/#id56>

8

Also acceptable is a *legacy*, generator-based coroutine, which is a generator function that is decorated with `@types.coroutine`, and uses the `yield` from keyword internally to suspend. We are going to completely ignore legacy coroutines in this book. Erase it from your mind!

9

The documentation is inconsistent here: The signature is given as `AbstractEventLoop.run_until_complete(future)` but it really should be `AbstractEventLoop.run_until_complete(coro_or_future)` as the same rules apply.

10

<https://github.com/python/asyncio/issues/477#issuecomment-268709555>

11

Here “current” means the event loop instance that is associated with the active thread.

12

Async support can be quite difficult to add to an existing framework after the fact since large structural changes to the codebase might be needed. This was discussed in a [github issue for Requests](#).

13

Yes, this is super annoying. Every time I use this call I can’t help wondering why the more common idiom of using `executor=None` as a keyword argument was not preferred.

14

`add_signal_handler()` should probably be named `set_signal_handler()`, since you can have only one handler per signal type, and calling `add_signal_handler()` a second time for the same signal will *replace* a previous handler for that signal if one exists.

15

And yet another option that can be very useful to speed up tests: I could also have called `loop.stop()` *inside* `fake_notify()`, immediately after collecting the `msg` in the list. This saves time because the `run_forever()` call is immediately unblocked, and we don't wait for `f()` to do any further processing. Of course, this may mean that you get warning output about “pending tasks” when the fixture calls `loop.close()` during finalization. Testing is a tricky art, and each situation is different.