To illustrate this, let's say we launch a long-running task that we don't want to run for longer than 5 seconds. If the task is not completed within 5 seconds, we'd like to stop that task, reporting back to the user that it took too long and we're stopping it. We also want a status update printed every second, to provide up-to-date information to our user, so they aren't left without information for several seconds.

Listing 2.11 Canceling a task

```
import asyncio
from asyncio import CancelledError
from util import delay


async def main():
    long_task = asyncio.create_task(delay(10))

    seconds_elapsed = 0

    while not long_task.done():
        print('Task not finished, checking again in a second
        await asyncio.sleep(1)
        seconds_elapsed = seconds_elapsed + 1
        if seconds_elapsed == 5:
            long_task.cancel()

    try:
        await long_task
    except CancelledError:
        print('Our task was cancelled')
```

```
asyncio.run(main())
```

In the preceding listing, we create a task that will take 10 seconds to run. We then create a `while` loop to check if that task is done. The `done` method on the task returns `True` if a task is finished and `False` otherwise. Every second, we check to see if the task has finished, keeping track of how many seconds we've checked so far. If our task has taken 5 seconds, we cancel the task. Then, we will go on to `await long_task`, and we'll see `Our task was cancelled` printed out, indicating we've caught a `CancelledError`.

Something important to note about cancellation is that a `CancelledError` can only be thrown from an await statement. This means that if we call cancel on a task when it is executing plain Python code, that code will run until completion until we hit the next await statement (if one exists) and a `CancelledError` can be raised. Calling cancel won't magically stop the task in its tracks; it will only stop the task if you're currently at an await point or its next await point.

etting a timeout and canceling with wait_for

Checking every second or at some other time interval, and then canceling a task, as we did in the previous example, isn't the easiest way to handle a timeout. Ideally, we'd have a helper function that would allow us to specify this timeout and handle cancellation for us.

asyncio provides this functionality through a function called `asyncio.wait_for`. This function takes in a coroutine or task object, and a timeout specified in seconds. It then returns a coroutine that we can `await`. If the task takes more time to complete

than the timeout we gave it, a `TimeoutException` will be raised. Once we have reached the timeout threshold, the task will automatically be canceled.

To illustrate how `wait_for` works, we'll look at a case where we have a task that will take 2 seconds to complete, but we'll only allow it 1 second to finish. When we get a `TimeoutError` raised, we'll catch the exception and check to see if the task was canceled.

Listing 2.12 Creating a timeout for a task with `wait_for`

```
import asyncio
from util import delay

async def main():
    delay_task = asyncio.create_task(delay(2))
    try:
        result = await asyncio.wait_for(delay_task, timeout=
        print(result)
    except asyncio.exceptions.TimeoutError:
        print('Got a timeout!')
        print(f'Was the task cancelled? {delay_task.cancelle

asyncio.run(main())
```

When we run the preceding listing, our application will take roughly 1 second to complete. After 1 second our `wait_for` statement will raise a `TimeoutError`, which we then handle. We'll then see that our original `delay` task was canceled, giving the following output:

```
sleeping for 2 second(s)
Got a timeout!
Was the task cancelled? True
```

Canceling tasks automatically if they take longer than expected is normally a good idea. Otherwise, we may have a coroutine waiting indefinitely, taking up resources that may never be released. However, in certain circumstances we may want to keep our coroutine running. For example, we may want to inform a user that something is taking longer than expected after a certain amount of time but not cancel the task when the timeout is exceeded.

To do this we can wrap our task with the `asyncio.shield` function. This function will prevent cancellation of the coroutine we pass in, giving it a "shield," which cancellation requests then ignore.

Listing 2.13 Shielding a task from cancellation

```
import asyncio
from util import delay


async def main():
    task = asyncio.create_task(delay(10))

    try:
        result = await asyncio.wait_for(asyncio.shield(task)
        print(result)
    except TimeoutError:
        print("Task took longer than five seconds, it will f
        result = await task
```

```
        print(result)


asyncio.run(main())
```

In the preceding listing, we first create a task to wrap our coroutine. This differs from our first cancellation example because we'll need to access the task in the `except` block. If we had passed in a coroutine, `wait_for` would have wrapped it in a task, but we wouldn't be able to reference it, as it is internal to the function.

Then, inside of a `try` block, we call `wait_for` and wrap the task in `shield`, which will prevent the task from being canceled. Inside our exception block, we print a useful message to the user, letting them know that the task is still running and then we `await` the task we initially created. This will let it finish in its entirety, and the program's output will be as follows:

```
sleeping for 10 second(s)
Task took longer than five seconds!
finished sleeping for 10 second(s)
finished <function delay at 0x10e8cf820> in 10 second(s)
```

Cancellation and shielding are somewhat tricky subjects with several cases that are noteworthy. We introduce the basics below, but as we get into more complicated cases, we'll explore how cancellation works in greater depth.

We've now introduced the basics of tasks and coroutines. These concepts are intertwined with one another. In the following section, we'll look at how tasks and

coroutines are related to one another and understand a bit more about how asyncio is structured.

sks, coroutines, futures, and awaitables

Coroutines and tasks can both be used in `await` expressions. So what is the common thread between them? To understand, we'll need to know about both a `future` as well as an `awaitable`. You normally won't need to use futures, but understanding them is a key to understanding the inner workings of asyncio. As some APIs return futures, we will reference them in the rest of the book.

ntroducing futures

A `future` is a Python object that contains a single value that you expect to get at some point in the future but may not yet have. Usually, when you create a `future`, it does not have any value it wraps around because it doesn't yet exist. In this state, it is considered incomplete, unresolved, or simply not done. Then, once you get a result, you can set the value of the `future`. This will complete the `future`; at that time, we can consider it finished and extract the result from the `future`. To understand the basics of futures, let's try creating one, setting its value and extracting that value back out.

Listing 2.14 The basics of futures

```
from asyncio import Future

my_future = Future()

print(f'Is my_future done? {my_future.done()}')
```

```
my_future.set_result(42)

print(f'Is my_future done? {my_future.done()}')
print(f'What is the result of my_future? {my_future.result()
```

We can create a `future` by calling its constructor. At this time, the `future` will have no result set on it, so calling its `done` method will return `False`. We then set the value of the `future` with its `set_result` method, which will mark the `future` as `done`. Alternatively, if we had an exception we wanted to set on the `future`, we could call `set_exception`.

NOTE We don't call the result method before the result is set because the `result` method will throw an invalid state exception if we do so.

Futures can also be used in `await` expressions. If we `await` a `future`, we're saying "pause until the `future` has a value set that I can work with, and once I have a value, wake up and let me process it."

To understand this, let's consider an example of making a web request that returns a `future`. Making a request that returns a `future` should complete instantly, but as the request will take some time, the `future` will not yet be defined. Then, later, once the request has finished, the result will be set, then we can access it. If you have used JavaScript in the past, this concept is analogous to *promises*. In the Java world, these are known as *completable futures*.

Listing 2.15 Awaiting a future

```
from asyncio import Future
import asyncio


def make_request() -> Future:
    future = Future()
    asyncio.create_task(set_future_value(future))     ❶
    return future


async def set_future_value(future) -> None:
    await asyncio.sleep(1)                            ❷
    future.set_result(42)


async def main():
    future = make_request()
    print(f'Is the future done? {future.done()}')
    value = await future                              ❸
    print(f'Is the future done? {future.done()}')
    print(value)


asyncio.run(main())
```

❶ Create a task to asynchronously set the value of the future.

❷ Wait 1 second before setting the value of the future.

❸ Pause main until the future's value is set.

In the preceding listing, we define a function `make_request .` In that function we create a `future` and create a `task` that will asynchronously set the result of the `future` after 1 second. Then, in the main function, we call `make_request`. When we call this, we'll instantly get a `future` with no result; it is, therefore, undone. Then, we `await` the `future`. Awaiting this `future` will pause `main` for 1 second while we wait for the value of the future to be set. Once this completes, `value` will be `42` and the `future` is `done`.

In the world of asyncio, you should rarely need to deal with futures. That said, you will run into some asyncio APIs which return futures, and you may need to work with callback-based code, which can require futures. You may also need to read or debug some asyncio API code yourself. The implementation of these asyncio APIs heavily rely on futures, so it is ideal to have a basic understanding of how they work.

The relationship between futures, tasks, and coroutines

There is a strong relationship between tasks and futures. In fact, `task` directly inherits from `future`. A `future` can be thought as representing a value that we won't have for a while. A `task` can be thought as a combination of both a coroutine and a `future`. When we create a `task`, we are creating an empty `future` and running the coroutine. Then, when the coroutine has completed with either an exception or a result, we set the result or exception of the `future`.

Given the relationship between futures and tasks, is there a similar relationship between tasks and coroutines? After all, all these types can be used in `await` expressions.

The common thread between these is the `Awaitable` abstract base class. This class defines one abstract double underscore method __`await`__. We won't go into the

specifics about how to create our own awaitables, but anything that implements the `__await__` method can be used in an `await` expression. Coroutines inherit directly from `Awaitable`, as do `futures`. Tasks then extend futures, which gives us the inheritance diagram shown in figure 2.5.

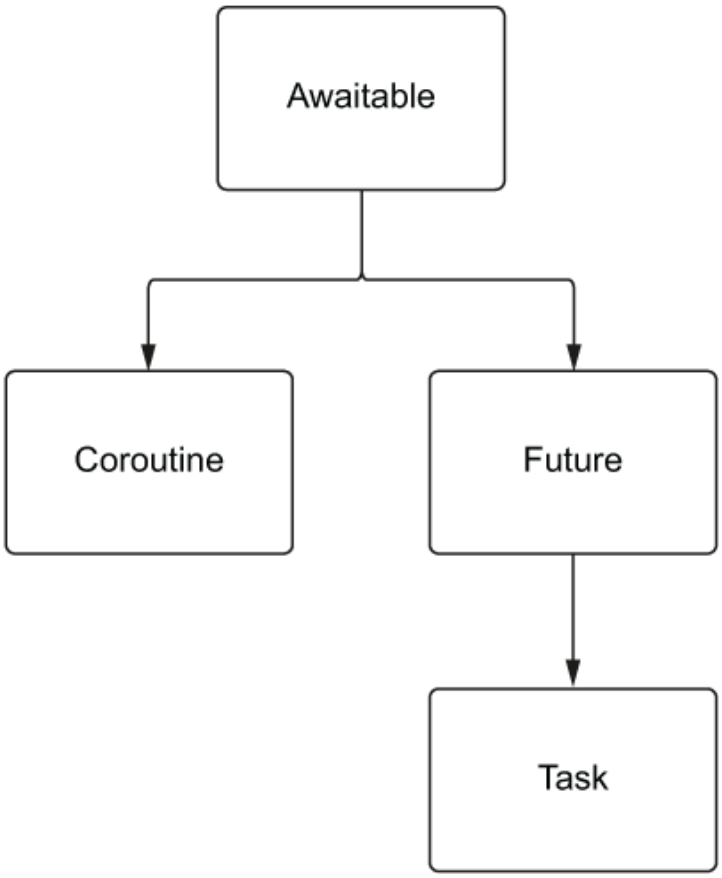

Figure 2.5 The class inheritance hierarchy of Awaitable

Going forward, we'll start to refer to objects that can be used in await expressions as *awaitables*. You'll frequently see the term *awaitable* referenced in the asyncio documentation, as many API methods don't care if you pass in coroutines, tasks, or futures.

Now that we understand the basics of coroutines, tasks, and futures, how do we assess their performance? So far, we've only theorized about how long they take. To make things more rigorous, let's add some functionality to measure execution time.

easuring coroutine execution time with decorators

So far, we've talked about roughly how long our applications take to run without timing them. To really understand and profile things we'll need to introduce some code to keep track of this for us.

As a first try we could wrap every `await` statement and keep track of the start and end time of the coroutine:

```
import asyncio
import time

async def main():
    start = time.time()
    await asyncio.sleep(1)
    end = time.time()
    print(f'Sleeping took {end - start} seconds')

asyncio.run(main())
```

However, this will get messy quickly when we have multiple `await` statements and tasks to keep track of. A better approach is to come up with a reusable way to keep track of how long any coroutine takes to finish. We can do this by creating a decorator that will run an `await` statement for us (listing 2.16). We'll call this decorator `async_timed`.

What is a decorator?

A *decorator* is a pattern in Python that allows us to add functionality to existing functions without changing that function's code. We can "intercept" a function as it is being called and apply any decorator code we'd like before or after that call. Decorators are one way to tackle cross-cutting concerns. The following listing illustrates a sample decorator.

Listing 2.16 A decorator for timing coroutines

```
import functools
import time
from typing import Callable, Any


def async_timed():
    def wrapper(func: Callable) -> Callable:
        @functools.wraps(func)
        async def wrapped(*args, **kwargs) -> Any:
            print(f'starting {func} with args {args} {kwargs
            start = time.time()
            try:
                return await func(*args, **kwargs)
            finally:
                end = time.time()
                total = end - start
                print(f'finished {func} in {total:.4f} secon

        return wrapped
```

```
    return wrapper
```

In this decorator, we create a new coroutine called *wrapped*. This is a wrapper around our original coroutine that takes its arguments, `*args` and `**kwargs`, calls an `await` statement, and then returns the result. We surround that `await` statement with one message when we start running the function and another message when we end running the function, keeping track of the start and end time in much the same way that we did in our earlier start-time and end-time example. Now, as shown in listing 2.17, we can put this annotation on any coroutine, and any time we run it, we'll see how long it took to run.

Listing 2.17 Timing two concurrent tasks with a decorator

```
import asyncio

@async_timed()
async def delay(delay_seconds: int) -> int:
    print(f'sleeping for {delay_seconds} second(s)')
    await asyncio.sleep(delay_seconds)
    print(f'finished sleeping for {delay_seconds} second(s)'
    return delay_seconds


@async_timed()
async def main():
    task_one = asyncio.create_task(delay(2))
    task_two = asyncio.create_task(delay(3))
```

```
    await task_one
    await task_two


asyncio.run(main())
```

When we run the preceding listing, we'll see console output similar to the following:

```
starting <function main at 0x109111ee0> with args () {}
starting <function delay at 0x1090dc700> with args (2,) {}
starting <function delay at 0x1090dc700> with args (3,) {}
finished <function delay at 0x1090dc700> in 2.0032 second(s)
finished <function delay at 0x1090dc700> in 3.0003 second(s)
finished <function main at 0x109111ee0> in 3.0004 second(s)
```

We can see that our two `delay` calls were both started and finished in roughly 2 and 3 seconds, respectively, for a total of 5 seconds. Notice, however, that our main coroutine only took 3 seconds to complete because we were waiting concurrently.

We'll use this decorator and the resulting output throughout the next several chapters, to illustrate how long our coroutines are taking to execute as well as when they start and complete. This will give us a clear picture of where we see performance gains by executing our operations concurrently.

To make referencing this utility decorator easier in future code listings, let's add this to our `util` module. We'll put our timer in a file called `async_timer.py`. We'll also add a line to the module's `__init__.py` file with the following line so we can nicely import the timer:

```
from util.async_timer import async_timed
```

In the rest of this book, we'll use `from util import async_timed` whenever we need to use the timer.

Now that we can use our decorator to understand the performance gains that asyncio can provide when running tasks concurrently, we may be tempted to try and use asyncio all over our existing applications. This can work, but we need to be careful that we aren't running into any of the common pitfalls with asyncio that can degrade our application's performance.

e pitfalls of coroutines and tasks

When seeing the performance improvements we can obtain from running some of our longer tasks concurrently, we can be tempted to start to use coroutines and tasks everywhere in our applications. While it depends on the application you're writing, simply marking functions `async` and wrapping them in tasks may not help application performance. In certain cases, this may degrade performance of your applications.

Two main errors occur when trying to turn your applications asynchronous. The first is attempting to run CPU-bound code in tasks or coroutines without using multiprocessing; the second is using blocking I/O-bound APIs without using multithreading.

unning CPU-bound code

We may have functions that perform computationally expensive calculations, such as looping over a large dictionary or doing a mathematical computation. Where we have

several of these functions with the potential to run concurrently, we may get the idea to run them in separate tasks. In concept, this is a good idea, but remember that asyncio has a single-threaded concurrency model. This means we are still subject to the limitations of a single thread and the global interpreter lock.

To prove this to ourselves, let's try to run some CPU-bound functions concurrently.

Listing 2.18 Attempting to run CPU-bound code concurrently

```
import asyncio
from util import delay


@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
        counter = counter + 1
    return counter


@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    await task_one
    await task_two


asyncio.run(main())
```

When we run the preceding listing, we'll see that, despite creating two tasks, our code still executes sequentially. First, we run Task 1, then we run Task 2, meaning our total runtime will be the sum of the two calls to `cpu_bound_work`:

```
starting <function main at 0x10a8f6c10> with args () {}
starting <function cpu_bound_work at 0x10a8c0430> with args
finished <function cpu_bound_work at 0x10a8c0430> in 4.6750
starting <function cpu_bound_work at 0x10a8c0430> with args
finished <function cpu_bound_work at 0x10a8c0430> in 4.6680
finished <function main at 0x10a8f6c10> in 9.3434 second(s)
```

Looking at the output above, we may be tempted to think that there are no drawbacks to making all our code use `async` and `await`. After all, it ends up taking the same amount of time as if we had run things sequentially. However, by doing this we can run into situations where our application's performance can degrade. This is especially true when we have other coroutines or tasks that have `await` expressions. Consider creating two CPU-bound tasks alongside a long-running task, such as our `delay` coroutine.

Listing 2.19 CPU-bound code with a task

```
import asyncio
from util import async_timed, delay


@async_timed()
async def cpu_bound_work() -> int:
    counter = 0
    for i in range(100000000):
```

```
            counter = counter + 1
        return counter


@async_timed()
async def main():
    task_one = asyncio.create_task(cpu_bound_work())
    task_two = asyncio.create_task(cpu_bound_work())
    delay_task = asyncio.create_task(delay(4))
    await task_one
    await task_two
    await delay_task


asyncio.run(main())
```

Running the preceding listing, we might expect to take the same amount of time as in listing 2.18. After all, won't `delay_task` run concurrently alongside the CPU-bound work? In this instance it won't because we create the two CPU-bound tasks first, which, in effect, blocks the event loop from running anything else. This means the runtime of our application will be the sum of time it took for our two `cpu_bound_work` tasks to finish plus the 4 seconds that our `delay` task took.

If we need to perform CPU-bound work and still want to use `async` / `await` syntax, we can do so. To do this we'll still need to use multiprocessing, and we need to tell asyncio to run our tasks in a *process pool*. We'll learn how to do this in chapter 6.

Running blocking APIs

We may also be tempted to use our existing libraries for I/O-bound operations by wrapping them in coroutines. However, this will generate the same issues that we saw with CPU-bound operations. These APIs block the `main` thread. Therefore, when we run a blocking API call inside a coroutine, we're blocking the event loop thread itself, meaning that we stop any other coroutines or tasks from executing. Examples of blocking API calls include libraries such as `requests`, or `time.sleep`. Generally, any function that performs I/O that is not a coroutine or performs time-consuming CPU operations can be considered blocking.

As an example, let's try getting the status code of www.example.com three times concurrently, using the `requests` library. When we run this, since we're running concurrently we'll be expecting this application to finish in about the length of time necessary to get the status code once.

Listing 2.20 Incorrectly using a blocking API in a coroutine

```
import asyncio
import requests
from util import async_timed


@async_timed()
async def get_example_status() -> int:
    return requests.get('http:/ / www .example .com').status


@async_timed()
async def main():
```

```
    task_1 = asyncio.create_task(get_example_status())
    task_2 = asyncio.create_task(get_example_status())
    task_3 = asyncio.create_task(get_example_status())
    await task_1
    await task_2
    await task_3

asyncio.run(main())
```

When running the preceding listing, we'll see output similar to the following. Note how the total runtime of the main coroutine is roughly the sum of time for all the tasks to get the status we ran, meaning that we did not have any concurrency advantage:

```
starting <function main at 0x1102e6820> with args () {}
starting <function get_example_status at 0x1102e6700> with a
finished <function get_example_status at 0x1102e6700> in 0.0
starting <function get_example_status at 0x1102e6700> with a
finished <function get_example_status at 0x1102e6700> in 0.0
starting <function get_example_status at 0x1102e6700> with a
finished <function get_example_status at 0x1102e6700> in 0.0
finished <function main at 0x1102e6820> in 0.1702 second(s)
```

This is again because the `requests` library is blocking, meaning it will block whichever thread it is run on. Since asyncio only has one thread, the `requests` library blocks the event loop from doing anything concurrently.

As a rule, most APIs you employ now are blocking and won't work out of the box with asyncio. You need to use a library that supports coroutines and utilizes non-

blocking sockets. This means that if the library you are using does not return coroutines and you aren't using `await` in your own coroutines, you're likely making a blocking call.

In the above example we can use a library such as aiohttp, which uses non-blocking sockets and returns coroutines to get proper concurrency. We'll introduce this library later in chapter 4.

If you need to use the `requests` library, you can still use `async` syntax, but you'll need to explicitly tell asyncio to use multithreading with a *thread pool executor*. We'll see how to do this in chapter 7.

We've now seen a few things to look for when using asyncio and have built a few simple applications. So far, we have not created or configured the event loop ourselves but relied on convenience methods to do it for us. Next, we'll learn to create the event loop, which will allow us to access lower-level asyncio functionality and event loop configuration properties.

cessing and manually managing the event loop

Until now, we have used the convenient `asyncio.run` to run our application and create the event loop for us behind the scenes. Given the ease of use, this is the preferred method to create the event loop. However, there may be cases in which we don't want the functionality that `asyncio.run` provides. As an example, we may want to execute custom logic to stop tasks that differ from what `asyncio.run` does, such as letting any remaining tasks finish instead of stopping them.

In addition, we may want to access methods available on the event loop itself. These methods are typically lower level and, as such, should be used sparingly. However, if

you want to perform tasks, such as working directly with sockets or scheduling a task to run at a specific time in the future, you'll need to access the event loop. While we won't, and shouldn't, be managing the event loop extensively, this will be necessary from time to time.

Creating an event loop manually

We can create an event loop by using the `asyncio.new_event_loop` method. This will return an event loop instance. With this, we have access to all the low-level methods that the event loop has to offer. With the event loop we have access to a method named `run_until_complete`, which takes a coroutine and runs it until it finishes. Once we are done with our event loop, we need to close it to free any resources it was using. This should normally be in a `finally` block so that any exceptions thrown don't stop us from closing the loop. Using these concepts, we can create a loop and run an asyncio application.

Listing 2.21 Manually creating the event loop

```
import asyncio

async def main():
    await asyncio.sleep(1)

loop = asyncio.new_event_loop()

try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

The code in this listing is similar to what happens when we call `asyncio.run` with the difference being that this does not perform canceling any remaining tasks. If we want any special cleanup logic, we would do so in our `finally` clause.

ccessing the event loop

From time to time, we may need to access the currently running event loop. asyncio exposes the `asyncio.get_running_loop` function that allows us to get the current event loop. As an example, let's look at `call_soon`, which will schedule a function to run on the next iteration of the event loop.

Listing 2.22 Accessing the event loop

```
import asyncio

def call_later():
    print("I'm being called in the future!")
async def main():
    loop = asyncio.get_running_loop()
    loop.call_soon(call_later)
    await delay(1)

asyncio.run(main())
```

In the preceding listing, our main coroutine gets the event loop with `asyncio.get_running_loop` and tells it to run `call_later`, which takes a function and will run it on the next iteration of the event loop. In addition, there is an `asyncio.get_event_loop` function that lets you access the event loop.

This function can potentially create a new event loop if it is called when one is not already running, leading to strange behavior. It is recommended to use `get_running_loop`, as this will throw an exception if an event loop isn't running, avoiding any surprises.

While we shouldn't use the event loop frequently in our applications, there are times when we will need to configure settings on the event loop or use low-level functions. We'll see an example of configuring the event loop in the next section on *debug mode*.

ing debug mode

In previous sections, we mentioned how coroutines should always be awaited at some point in the application. We also saw the drawbacks of running CPU-bound and other blocking code inside coroutines and tasks. It can, however, be hard to tell if a coroutine is taking too much time on CPU, or if we accidently forgot an `await` somewhere in our application. Luckily, asyncio gives us a debug mode to help us diagnose these situations.

When we run in `debug` mode, we'll see a few helpful log messages when a coroutine or task takes more than 100 milliseconds to run. In addition, if we don't `await` a coroutine, an exception is thrown, so we can see where to properly add an `await`. There are a few different ways to run in debug mode.

Using asyncio.run

The `asyncio.run` function we have been using to run coroutines exposes a `debug` parameter. By default, this is set to `False`, but we can set this to `True` to enable debug mode: