# 2 asyncio basics

This chapter covers

- The basics of `async` `await` syntax and coroutines
- Running coroutines concurrently with tasks
- Canceling tasks
- Manually creating the event loop
- Measuring a coroutine's execution time
- Keeping eyes open for problems when running coroutines

Chapter 1 dived into concurrency, looking at how we can achieve it with both processes and threads. We also explored how we could utilize non-blocking I/O and an event loop to achieve concurrency with only one thread. In this chapter, we'll cover the basics of how to write programs using this single-threaded concurrency model with asyncio. Using the techniques in this chapter, you'll be able to take long-running operations, such as web requests, database queries, and network connections and execute them in tandem.

We'll learn more about the *coroutine* construct and how to use `async` `await` syntax to define and run coroutines. We'll also examine how to run coroutines concurrently by using tasks and examine the time savings we get from running concurrently by creating a reusable timer. Finally, we'll look at common errors software engineers may make when using asyncio and how to use debug mode to spot these problems.

## roducing coroutines

Think of a coroutine like a regular Python function but with the superpower that it can pause its execution when it encounters an operation that could take a while to complete. When that long-running operation is complete, we can “wake up” our paused coroutine and finish executing any other code in that coroutine. While a paused coroutine is waiting for the operation it paused for to finish, we can run other code. This running of other code while waiting is what gives our application concurrency. We can also run several time-consuming operations concurrently, which can give our applications big performance improvements.

To both create and pause a coroutine, we’ll need to learn to use Python’s `async` and `await` keywords. The `async` keyword will let us define a coroutine; the `await` keyword will let us pause our coroutine when we have a long-running operation.

Which Python version should I use?

The code in this book assumes you are using the latest version of Python at the time of writing, which is Python 3.10. Running code with versions earlier than this may have certain API methods missing, may function differently, or may have bugs.

### reating coroutines with the async keyword

Creating a coroutine is straightforward and not much different from creating a normal Python function. The only difference is that, instead of using the `def` keyword, we use `async` `def`. The `async` keyword marks a function as a coroutine instead of a normal Python function.

Listing 2.1 Using the `async` keyword

```
async def my_coroutine() -> None
    print(‘Hello world!’)
```

The coroutine in the preceding listing does nothing yet other than print “Hello world!” It’s also worth noting that this coroutine does not perform any long-running operations; it just prints our message and returns. This means that, when we put the coroutine on the event loop, it will execute immediately because we don’t have any blocking I/O, and nothing is pausing execution yet.

This syntax is simple, but we’re creating something very different from a plain Python function. To illustrate this, let’s create a function that adds one to an integer as well as a coroutine that does the same and compare the results of calling each. We’ll also use the `type` convenience function to look at the type returned by calling a coroutine as compared to calling our normal function.

Listing 2.2 Comparing coroutines to normal functions

```
async def coroutine_add_one(number: int) -> int:
    return number + 1


def add_one(number: int) -> int:
    return number + 1


function_result = add_one(1)
coroutine_result = coroutine_add_one(1)
```

```
print(f'Function result is {function_result} and the type is
print(f'Coroutine result is {coroutine_result} and the type
```

When we run this code, we’ll see output like the following:

```
Method result is 2 and the type is <class 'int'>
Coroutine result is <coroutine object coroutine_add_one at 0
```

Notice how when we call our normal `add_one` function it executes immediately and returns what we would expect, another integer. However, when we call `coroutine_add_one` we don’t get our code in the coroutine executed at all. We get a *coroutine object* instead.

This is an important point, as coroutines aren’t executed when we call them directly. Instead, we create a coroutine object that can be run later. To run a coroutine, we need to explicitly run it on an event loop. So how can we create an event loop and run our coroutine?

In versions of Python older than 3.7, we had to create an event loop if one did not already exist. However, the asyncio library has added several functions that abstract the event loop management. There is a convenience function, `asyncio.run`, we can use to run our coroutine. This is illustrated in the following listing.

Listing 2.3 Running a coroutine

```
import asyncio
```

```
async def coroutine_add_one(number: int) -> int:
    return number + 1


result = asyncio.run(coroutine_add_one(1))

print(result)
```

Running listing 2.3 will print “2,” as we would expect for returning the next integer. We’ve properly put our coroutine on the event loop, and we have executed it!

`asyncio.run` is doing a few important things in this scenario. First, it creates a brand-new event. Once it successfully does so, it takes whichever coroutine we pass into it and runs it until it completes, returning the result. This function will also do some cleanup of anything that might be left running after the main coroutine finishes. Once everything has finished, it shuts down and closes the event loop.

Possibly the most important thing about `asyncio.run` is that it is intended to be the main entry point into the asyncio application we have created. It only executes one coroutine, and that coroutine should launch all other aspects of our application. As we progress further, we will use this function as the entry point into nearly all our applications. The coroutine that `asyncio.run` executes will create and run other coroutines that will allow us to utilize the concurrent nature of asyncio.

## ’ausing execution with the await keyword

The example we saw in listing 2.3 did not need to be a coroutine, as it executed only non-blocking Python code. The real benefit of asyncio is being able to pause execution to let the event loop run other tasks during a long-running operation. To pause execution, we use the `await` keyword. The `await` keyword is usually followed by a call to a coroutine (more specifically, an object known as an *awaitable*, which is not always a coroutine; we'll learn more about awaitables later in the chapter).

Using the `await` keyword will cause the coroutine following it to be run, unlike calling a coroutine directly, which produces a coroutine object. The `await` expression will also pause the coroutine where it is contained in until the coroutine we awaited finishes and returns a result. When the coroutine we awaited finishes, we'll have access to the result it returned, and the containing coroutine will "wake up" to handle the result.

We can use the `await` keyword by putting it in front of a coroutine call. Expanding on our earlier program, we can write a program where we call the `add_one` function inside of a "main" async function and get the result.

**Listing 2.4 Using `await` to wait for the result of coroutine**

```
import asyncio

async def add_one(number: int) -> int:
    return number + 1



async def main() -> None:
    one_plus_one = await add_one(1)     ❶
    two_plus_one = await add_one(2)     ❷
    print(one_plus_one)
    print(two_plus_one)
```

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

❶ Pause, and wait for the result of add_one(1).

❷ Pause, and wait for the result of add_one(2).

In listing 2.4, we pause execution twice. We first `await` the call to `add_one(1)`. Once we have the result, the main function will be "unpaused," and we will assign the return value from `add_one(1)` to the variable `one_plus_one`, which in this case will be two. We then do the same for `add_one(2)` and then print the results. We can visualize the execution flow of our application, as shown in figure 2.1. Each block of the figure represents what is happening at any given moment for one or more lines of code.

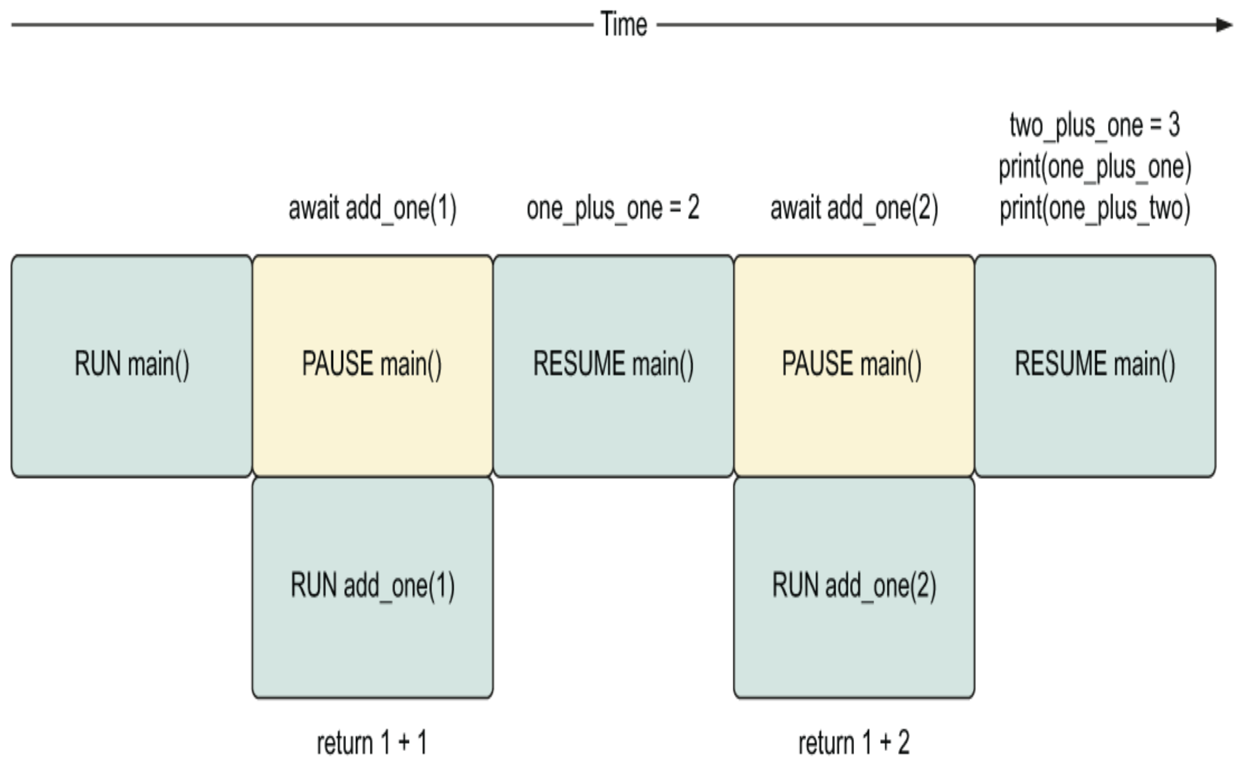

Figure 2.1 When we hit an `await` expression, we pause our parent coroutine and run the coroutine in the await expression. Once it is finished, we resume the parent coroutine and assign the return value.

As it stands now, this code does not operate differently from normal, sequential code. We are, in effect, mimicking a normal call stack. Next, let's look at a simple example of how to run other code by introducing a dummy sleep operation while we're waiting.

## roducing long-running coroutines with sleep

Our previous examples did not use any slow operations and were used to help us learn the basic syntax of coroutines. To fully see the benefits and show how we can run multiple events simultaneously, we'll need to introduce some long-running operations. Instead of making web API or database queries right away, which are nondeterministic as to how much time they will take, we'll simulate long-running operations by specifying how long we want to wait. We'll do this with the `asyncio.sleep` function.

We can use `asyncio.sleep` to make a coroutine "sleep" for a given number of seconds. This will pause our coroutine for the time we give it, simulating what would happen if we had a long-running call to a database or web API.

`asyncio.sleep` is itself a coroutine, so we must use it with the `await` keyword. If we call it just by itself, we'll get a coroutine object. Since `asyncio.sleep` is a coroutine, this means that when a coroutine awaits it, other code will be able to run.

Let's examine a simple example, shown in the following listing, that sleeps for 1 second and then prints a `'Hello World!'` message.

**Listing 2.5 A first application with `sleep`**

```
import asyncio

async def hello_world_message() -> str:
    await asyncio.sleep(1)                    ❶
    return ‘Hello World!’

async def main() -> None:
hello_world_message()   ❷
    print(message)

asyncio.run(main())
```

❶ Pause hello_world_message for 1 second.

❷ Pause main until hello_world_message finishes.

When we run this application, our program will wait 1 second before printing our `'Hello World!'` message. Since `hello_world_message` is a coroutine and we pause it for 1 second with `asyncio.sleep`, we now have 1 second where we could be running other code concurrently.

We'll be using `sleep` a lot in the next few examples, so let's invest the time to create a reusable coroutine that sleeps for us and prints out some useful information. We'll call this coroutine `delay`. This is shown in the following listing.

Listing 2.6 A reusable `delay` function

```
import asyncio


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
```

`delay` will take in an integer of the duration in seconds that we'd like the function to sleep and will return that integer to the caller once it has finished sleeping. We'll also print when sleep begins and ends. This will help us see what other code, if any, is running concurrently while our coroutines are paused.

To make referencing this utility function easier in future code listings, we'll create a module that we'll import in the remainder of this book when needed. We'll also add to this module as we create additional reusable functions. We'll call this module `util`, and we'll put our delay function in a file called `delay_functions.py`. We'll also add an `__init__.py` file with the following line, so we can nicely import the timer:

```
from util.delay_functions import delay
```

From now on in this book, we'll use `from` `util` `import` `delay` whenever we need to use the `delay` function. Now that we have a reusable delay coroutine, let's combine it with the earlier coroutine `add_one` to see if we can get our simple addition to run concurrently while `hello_world_message` is paused.

Listing 2.7 Running two coroutines

```
import asyncio
from util import delay

async def add_one(number: int) -> int:
    return number + 1

async def hello_world_message() -> str:
    await delay(1)
    return ‘Hello World!’

async def main() -> None:
    message = await hello_world_message()    (1)
    one_plus_one = await add_one(1)          (2)
    print(one_plus_one)
    print(message)
```

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

❶ Pause main until hello_world_message returns.

❷ Pause main until add_one returns.

When we run this, 1 second passes before the results of both function calls are printed. What we really want is the value of `add_one(1)` to be printed immediately while `hello_world_message()` runs concurrently. So why isn't this happening with this code? The answer is that `await` pauses our current coroutine and won't execute any other code inside that coroutine until the `await` expression gives us a value. Since it will take 1 second for our `hello_world_message` function to give us a value, the main coroutine will be paused for 1 second. Our code behaves as if it were sequential in this case. This behavior is illustrated in figure 2.2.

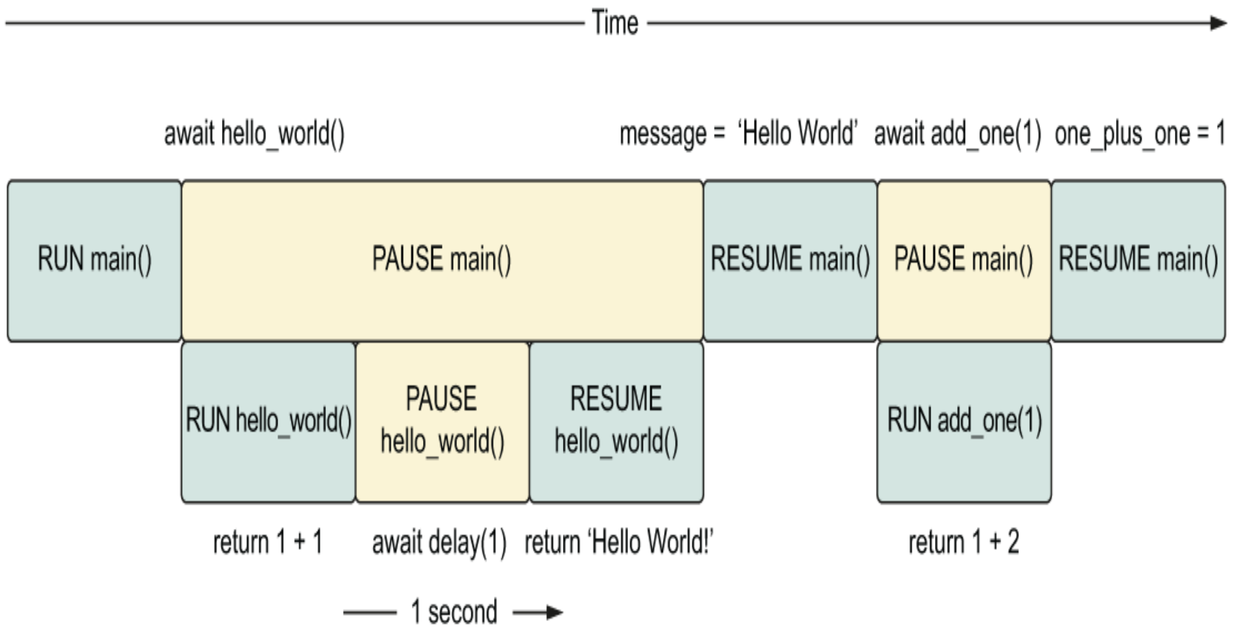

Figure 2.2 Execution flow of listing 2.7

Both `main` and `hello_world` paused while we wait for `delay(1)` to finish. After it has finished, `main` resumes and can execute `add_one`.

We'd like to move away from this sequential model and run `add_one` concurrently with `hello_world`. To achieve this, we'll need to introduce a concept called *tasks*.

## nning concurrently with tasks

Earlier we saw that, when we call a coroutine directly, we don't put it on the event loop to run. Instead, we get a coroutine object that we then need to either use the `await` keyword on it or pass it in to `asyncio.run` to run and get a value. With only these tools we can write async code, but we can't run anything concurrently. To run coroutines concurrently, we'll need to introduce *tasks*.

Tasks are wrappers around a coroutine that schedule a coroutine to run on the event loop as soon as possible. This scheduling and execution happen in a non-blocking fashion, meaning that, once we create a task, we can execute other code instantly while the task is running. This contrasts with using the `await` keyword that acts in a blocking manner, meaning that we pause the entire coroutine until the result of the `await` expression comes back.

The fact that we can create tasks and schedule them to run instantly on the event loop means that we can execute multiple tasks at roughly the same time. When these tasks wrap a long-running operation, any waiting they do will happen concurrently. To illustrate this, let's create two tasks and try to run them at the same time.

## he basics of creating tasks

Creating a task is achieved by using the `asyncio.create_task` function. When we call this function, we give it a coroutine to run, and it returns a task object instantly. Once we have a task object, we can put it in an `await` expression that will extract the return value once it is complete.

Listing 2.8 Creating a task

```
import asyncio
from util import delay


async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    print(type(sleep_for_three))
    result = await sleep_for_three
    print(result)

asyncio.run(main())
```

In the preceding listing, we create a task that requires 3 seconds to complete. We also print out the type of the task, in this case, `<class '_asyncio.Task'>`, to show that it is different from a coroutine.

One other thing to note here is that our print statement is executed immediately after we run the task. If we had simply used await on the delay coroutine we would have waited 3 seconds before outputting the message.

Once we’ve printed our message, we apply an `await` expression to the task `sleep_ for_three`. This will suspend our `main` coroutine until we have a result from our task.

It is important to know that we should usually use an `await` keyword on our tasks at some point in our application. In listing 2.8, if we did not use `await`, our task would be scheduled to run, but it would almost immediately be stopped and “cleaned up” when `asyncio.run` shut down the event loop. Using `await` on our tasks in our application also has implications for how exceptions are handled, which we’ll look at in chapter 3. Now that we’ve seen how to create a task and allow other code to run concurrently, we can learn how to run multiple long-running operations at the same time.

## unning multiple tasks concurrently

Given that tasks are created instantly and are scheduled to run as soon as possible, this allows us to run many long-running tasks concurrently. We can do this by sequentially starting multiple tasks with our long-running coroutine.

Listing 2.9 Running multiple tasks concurrently

```
import asyncio
from util import delay

async def main():
    sleep_for_three = asyncio.create_task(delay(3))
    sleep_again = asyncio.create_task(delay(3))
    sleep_once_more = asyncio.create_task(delay(3))

    await sleep_for_three
```

```
        await sleep_again
        await sleep_once_more


    asyncio.run(main())
```

In the preceding listing we start three tasks, each taking 3 seconds to complete. Each call to `create_task` returns instantly, so we reach the `await sleep_for_three` statement right away. Previously, we mentioned that tasks are scheduled to run “as soon as possible.” Generally, this means the first time we hit an `await` statement after creating a task, any tasks that are pending will run as `await` triggers an iteration of the event loop.

Since we’ve hit `await` `sleep_for_three`, all three tasks start running and will carry out any sleep operations concurrently. This means that the program in listing 2.9 will complete in about 3 seconds. We can visualize the concurrency as shown in figure 2.3, noting that all three tasks are running their sleep coroutines at the same time.

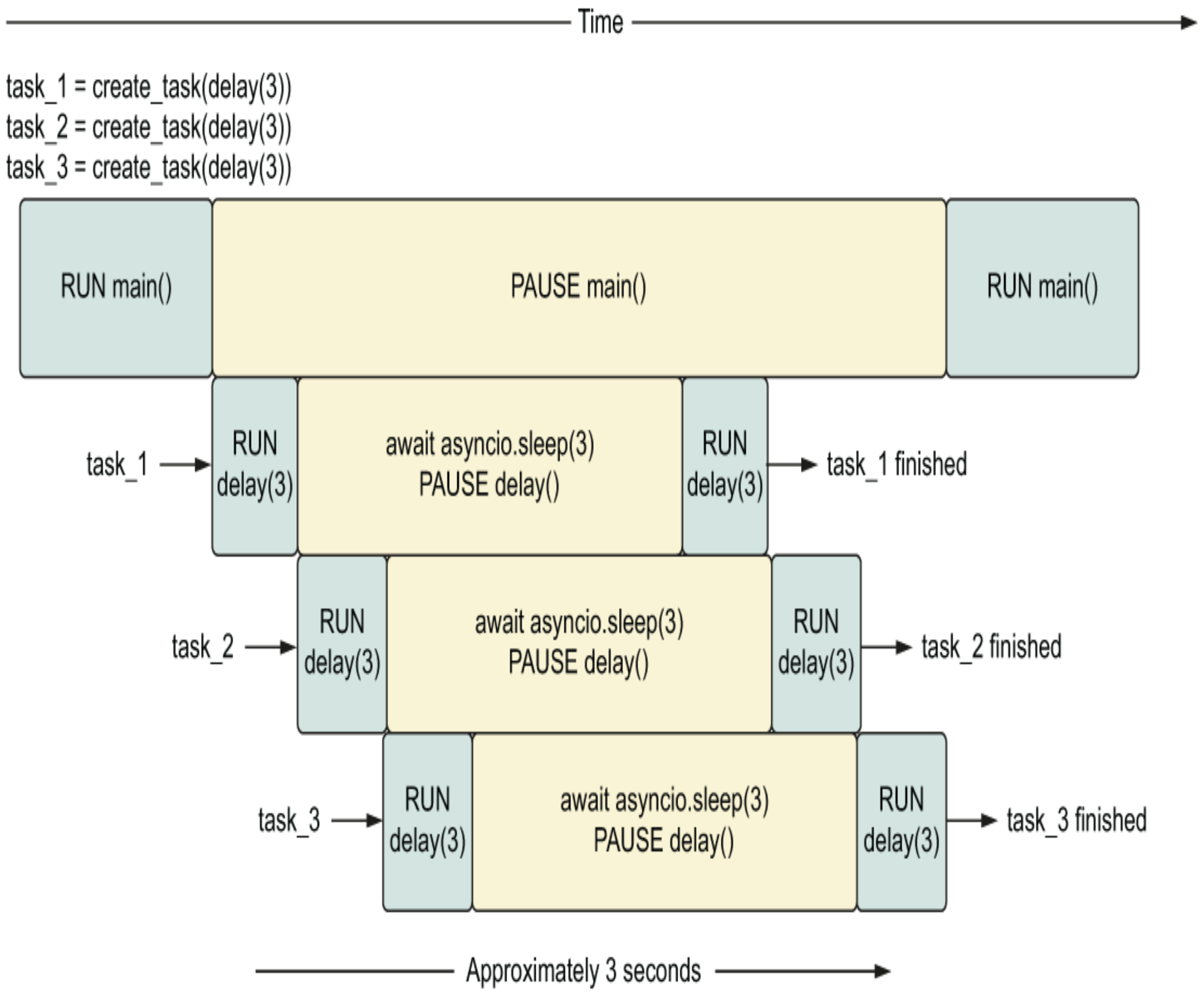
Time
task_1 = create_task(delay(3))
task_2 = create_task(delay(3))
task_3 = create_task(delay(3))
RUN main()
PAUSE main()
RUN main()
task_1
RUN delay(3)
await asyncio.sleep(3) PAUSE delay()
RUN delay(3)
task_1 finished
task_2
RUN delay(3)
await asyncio.sleep(3) PAUSE delay()
RUN delay(3)
task_2 finished
task_3
RUN delay(3)
await asyncio.sleep(3) PAUSE delay()
RUN delay(3)
task_3 finished
Approximately 3 seconds

Figure 2.3 Execution flow of listing 2.9

Note that in figure 2.3 the code in the tasks labeled RUN delay(3) (in this case, some print statements) does not run concurrently with other tasks; only the sleep coroutines run concurrently. If we were to run these delay operations sequentially, we'd have an application runtime of just over 9 seconds. By doing this concurrently, we've decreased the total runtime of this application three-fold!

**NOTE** This benefit compounds as we add more tasks; if we had launched 10 of these tasks, we would still take roughly 3 seconds, giving us a 10-fold speedup.

Executing these long-running operations concurrently is where asyncio really shines and delivers drastic improvements in our application's performance, but the benefits don't stop there. In listing 2.9, our application was actively doing nothing, while it was waiting for 3 seconds for our delay coroutines to complete. While our code is waiting, we can execute other code. As an example, let's say we wanted to print out a status message every second while we were running some long tasks.

Listing 2.10 Running code while other operations complete

```
import asyncio
from util import delay


async def hello_every_second():
    for i in range(2):
        await asyncio.sleep(1)
        print("I'm running other code while I'm waiting!")


async def main():
    first_delay = asyncio.create_task(delay(3))
    second_delay = asyncio.create_task(delay(3))
    await hello_every_second()
    await first_delay
    await second_delay
```

In the preceding listing, we create two tasks, each of which take 3 seconds to complete. While these tasks are waiting, our application is idle, which gives us the opportunity to run other code. In this instance, we run a coroutine `hello_every_second`, which prints a message every second 2 times. While our two tasks are running, we'll see messages being output, giving us the following:

```
sleeping for 3 second(s)
sleeping for 3 second(s)
I'm running other code while I'm waiting!
I'm running other code while I'm waiting!
finished sleeping for 3 second(s)
finished sleeping for 3 second(s)
```

We can imagine the execution flow as shown in figure 2.4.

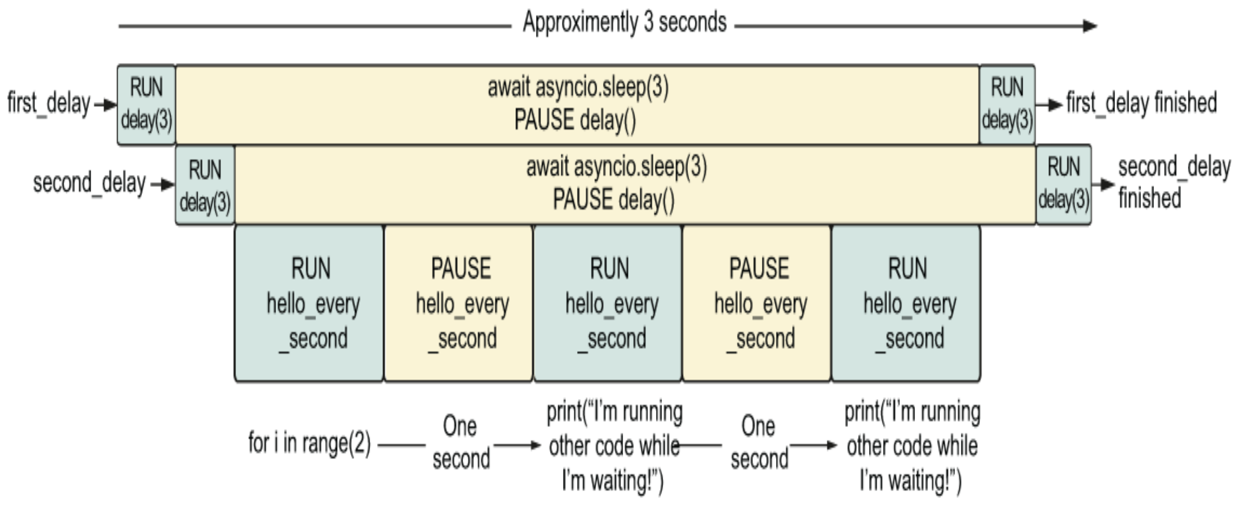

Figure 2.4 Execution flow of listing 2.10

First, we start two tasks that sleep for 3 seconds; then, while our two tasks are idle, we start to see `I'm running other code while I'm waiting!` being printed every second. This means that even when we're running time-intensive operations, our application can still be performing other tasks.

One potential issue with tasks is that they can take an indefinite amount of time to complete. We could find ourselves wanting to stop a task if it takes too long to finish. Tasks support this use case by allowing cancellation.

## nceling tasks and setting timeouts

Network connections can be unreliable. A user's connection may drop because of a network slowdown, or a web server may crash and leave existing requests in limbo. When making one of these requests, we need to be especially careful that we don't wait indefinitely. Doing so could lead to our application hanging, waiting forever for a result that may never come. It could also lead to a poor user experience; if we allow a user to make a request that takes too long, they are unlikely to wait forever for a response. Additionally, we may want to allow our users a choice if a task continues to run. A user may proactively decide things are taking too long, or they may want to stop a task they made in error.

In our previous examples, if our tasks took forever, we would be stuck waiting for the `await` statement to finish with no feedback. We also had no way to stop things if we wanted to. asyncio supports both these situations by allowing tasks to be canceled as well as allowing them to specify a timeout.

### anceling tasks

Canceling a task is straightforward. Each task object has a method named `cancel`, which we can call whenever we'd like to stop a task. Canceling a task will cause that task to raise a `CancelledError` when we `await` it, which we can then handle as needed.