you’ll need to add a tie-breaker key that gives you the ordering you want. One simple way to do this and preserve insertion order is to add an item count to the work item, though there are many ways you could do this.

Listing 12.9 Breaking ties in a priority queue

```python
import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field


@dataclass(order=True)
class WorkItem:
    priority: int
    order: int
    data: str = field(compare=False)


async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Processing work item {work_item}')
        queue.task_done()


async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 1, 'Lowest priority'),
                  WorkItem(3, 2, 'Lowest priority second'),
                  WorkItem(3, 3, 'Lowest priority third'),
                     WorkItem(2, 4, 'Medium priority'),
                     WorkItem(1, 5, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue)

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)


asyncio.run(main())
```

In the previous listing, we add an `order` field to our `WorkItem` class. Then, when we insert work items, we add an integer representing the order we insert it into the queue. When there is a tie in priority, this will be the field that we order on. In our case, this gives us the desired ordering of insertion for the low priority items:

```
Processing work item WorkItem(priority=1, order=5, data='Hig
Processing work item WorkItem(priority=2, order=4, data='Med
Processing work item WorkItem(priority=3, order=1, data='Low
Processing work item WorkItem(priority=3, order=2, data='Low
Processing work item WorkItem(priority=3, order=3, data='Low
```

We've now seen how to process work items in a FIFO queue order and in a priority queue order. What if we want to process the most recently added work items first? Next, let's see how to do this with a LIFO queue.

IFO queues

LIFO queues are more commonly referred to *stacks* in the computer science world. We can imagine these like a stack of poker chips: As you place bets, you take chips from the top of your stack (or “pop” them), and as you hopefully win hands, you put chips back on the top of the stack (or “push” them). These are useful for when we want our workers to process the most recently added items first.

We won’t build much more than a simple example to demonstrate the order that workers process elements. As for when to use a LIFO queue, it depends on the order your application needs to process items in the queue. Do you need to process the most recently inserted item in the queue first? In this case, you’ll want to use a LIFO queue.

Listing 12.10 A LIFO queue

```
import asyncio
from asyncio import Queue, LifoQueue
from dataclasses import dataclass, field


@dataclass(order=True)
class WorkItem:
    priority: int
    order: int
    data: str = field(compare=False)


async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()          ❶
        print(f'Processing work item {work_item}')
        queue.task_done()


async def main():
    lifo_queue = LifoQueue()

    work_items = [WorkItem(3, 1, 'Lowest priority first'),
                  WorkItem(3, 2, 'Lowest priority second'),
                  WorkItem(3, 3, 'Lowest priority third'),
                  WorkItem(2, 4, 'Medium priority'),
                  WorkItem(1, 5, 'High priority')]

    worker_task = asyncio.create_task(worker(lifo_queue))

    for work in work_items:
        lifo_queue.put_nowait(work)                        ❷

    await asyncio.gather(lifo_queue.join(), worker_task)


asyncio.run(main())
```

❶ Get an item from the queue, or “pop” it, from the stack.

❷ Put an item into the queue, or “push” it, onto the stack.

In the preceding listing, we create a LIFO queue and a set of work items. We then insert them one after another into the queue, pulling them out and processing them. Running this, you’ll see the following output:

```
Processing work item WorkItem(priority=1, order=5, data='Hig
Processing work item WorkItem(priority=2, order=4, data='Med
Processing work item WorkItem(priority=3, order=3, data='Low
Processing work item WorkItem(priority=3, order=2, data='Low
Processing work item WorkItem(priority=3, order=1, data='Low
```

Notice that we process items in the queue in the reverse order that we inserted them into the queue. As this is a stack, this makes sense, since we're processing the most recently added work item to our queue first.

We've now seen all the flavors of queue that the asyncio queue library has to offer. Are there any pitfalls to using these queues? Can we just use them whenever we need a queue in our application? We'll address this in chapter 13.

ary

- asyncio queues are task queues that are useful in workflows in which we have coroutines that produce data and coroutines responsible for processing that data.
- Queues decouple data generation from data processing, as we can have a producer put items into a queue that multiple workers can then process independently and concurrently.
- We can use priority queues to give certain tasks priority over one another. This is useful for instances in which certain work is of higher importance than others and should always be handled first.
- asyncio queues are not distributed, not persistent, and not durable. If you need any of these qualities, you'll need to look towards a separate architectural component, such as Celery or RabbitMQ.

13 Managing subprocesses

This chapter covers

- Running multiple subprocesses asynchronously
- Handling standard output from subprocesses
- Communicating with subprocesses using standard input
- Avoiding deadlocks and other pitfalls with subprocesses

Many applications will never need to leave the world of Python. We’ll call code from other Python libraries and modules or use multiprocessing or multithreading to run Python code concurrently. However, not everything we’ll want to interact with is written in Python. We may have an already built application that is written in C++, Go, Rust, or some other language that provides better runtime characteristics or is simply already there for us to use without reimplementing. We may also want to use OS provided command-line utilities, such as GREP for searching through large files, cURL for making HTTP requests, or any of the numbers of applications we have at our disposal.

In standard Python, we can use the subprocess module to run different applications in separate processes. Like most other Python modules, the standard subprocess API is blocking, making it incompatible with asyncio without multithreading or multiprocessing. asyncio provides a module modeled on the subprocess module to create and manage subprocesses asynchronously with coroutines.

In this chapter, we’ll learn the basics of creating and managing subprocesses with asyncio by running an application written in a different language. We’ll also learn how to handle input and output, reading standard output, and sending input from our application to our subprocesses.

reating a subprocess

Suppose you'd like to extend the functionality of an existing Python web API. Another team within your organization has already built the functionality you'd like in a command-line application for a batch processing mechanism they have, but there is a major problem in that the application is written in Rust. Given the application already exists, you don't want to reinvent the wheel by reimplementing it in Python. Is there a way we can still use this application's functionality within our existing Python API?

Since this application has a command-line interface, we can use subprocesses to reuse this application. We'll call the application via its command-line interface and run it in a separate subprocess. We can then read the results of the application and use it within our existing API as needed, saving us the trouble of having to reimplement the application.

So how do we create a subprocess and execute it? asyncio provides two coroutine functions out of the box to create subprocesses: `asyncio.create_subprocess_shell` and `asyncio.create_subprocess_exec`. Each of these coroutine functions returns an instance of a `Process`, which has methods to let us wait for the process to finish and terminate the process as well as a few others. Why are there two coroutines to accomplish seemingly the same task? When would we want to use one over the other? The `create_subprocess_shell` coroutine function creates a subprocess within a shell installed on the system it runs on such as `zsh` or `bash`. Generally speaking, you'll want to use `create_subprocess_exec` unless you need to use functionality from the shell. Using the shell can have pitfalls, such as different machines having different shells or the same shell configured differently. This can make it hard to guarantee your application will behave the same on different machines.

To learn the basics of how to create a subprocess, let's write an asyncio application to run a simple command-line program. We'll start with the `ls` program, which lists the contents of the current directory to test things out, although we wouldn't likely do this in the real world. If you're running on a Windows machine, replace `ls` `-l` with `cmd` `/c` `dir`.

Listing 13.1 Running a simple command in a subprocess

```
import asyncio
from asyncio.subprocess import Process


async def main():
    process: Process = await asyncio.create_subprocess_exec(
    print(f'Process pid is: {process.pid}')
    status_code = await process.wait()
    print(f'Status code: {status_code}')


asyncio.run(main())
```

In the preceding listing, we create a `Process` instance to run the `ls` command with `create_subprocess_exec`. We can also specify other arguments to pass to the program by adding them after. Here we pass in `-l`, which adds some extra information around who created the files in the directory. Once we've created the process, we print out the process ID and then call the `wait` coroutine. This coroutine will wait until the process finishes, and once it does it will return the subprocesses status code; in this case it should be zero. By default, standard output from our subprocess will be piped to standard output of our own application, so when you run this you should see something like the following, differing based in what you have in your directory:

```
Process pid is: 54438
total 8
drwxr-xr-x   4 matthewfowler  staff  128 Dec 23 15:20 .
drwxr-xr-x  25 matthewfowler  staff  800 Dec 23 14:52 ..
-rw-r--r--   1 matthewfowler  staff    0 Dec 23 14:52 __init
-rw-r--r--   1 matthewfowler  staff  293 Dec 23 15:20 basics
Status code: 0
```

Note that the `wait` coroutine will block until the application terminates, and there are no guarantees as to how long a process will take to terminate, let alone if it will terminate at all. If you're concerned about a runaway process, you'll need to introduce a timeout with `asyncio.wait_for`. There is a caveat to this, however. Recall that `wait_ for` will terminate the coroutine that it is running if it times out. You may assume that this will terminate the process, but it does not. It only terminates the task that is waiting for the process to finish, and not the underlying process.

We'll need a better way to shut down the process when it times out. Luckily, `Process` has two methods that can help us out in this situation: `terminate` and `kill`. The `terminate` method will send the `SIGTERM` signal to the subprocess, and kill will send the `SIGKILL` signal. Note that both these methods are not coroutines and are also non-blocking. They just send the signal. If you want to try and get the return code once you've terminated the subprocess or you want to wait for any cleanup, you'll need to call `wait` again.

Let's test out terminating a long-running application with the `sleep` command line application (for Windows users, replace `'sleep', '3'` with the more complicated `'cmd', 'start', '/wait', 'timeout', '3'`). We'll create a subprocess that sleeps for a few seconds and try to terminate it before it has a chance to finish.

Listing 13.2 Terminating a subprocess

```
import asyncio
from asyncio.subprocess import Process

async def main():
    process: Process = await asyncio.create_subprocess_exec(
    print(f'Process pid is: {process.pid}')
    try:
        status_code = await asyncio.wait_for(process.wait(),
        print(status_code)
    except asyncio.TimeoutError:
        print('Timed out waiting to finish, terminating...')
        process.terminate()
        status_code = await process.wait()
        print(status_code)
```

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

In the preceding listing, we create a subprocess that will take 3 seconds to complete but wrap it in a `wait_for` with a 1-second timeout. After 1 second, `wait_for` will throw a `TimeoutError`, and in the `except` block we terminate the process and wait for it to finish, printing out its status code. This should give us output similar to the following:

```
Process pid is: 54709
Timed out waiting to finish, terminating...
-15
```

One thing to watch out for when writing your own code is the `wait` inside of the `except` block still has a chance of taking a long time, and you may want to wrap this in a `wait_for` if this is a concern.

Controlling standard output

In the previous examples, the standard output of our subprocess went directly to our application's standard output. What if we don't want this behavior? Perhaps we want to do additional processing on the output, or maybe the output is inconsequential, and we can safely ignore it. The `create_subprocess_exec` coroutine has a `stdout` parameter that let us specify where we want standard output to go. This parameter takes in an `enum` that lets us specify if we want to redirect the subprocess's output to our own standard output, pipe it to a `StreamReader`, or ignore it entirely by redirecting it to `/dev/null`.

Let’s say we’re planning to run multiple subprocesses concurrently and echo their output. We’d like to know which subprocess generated the output to avoid confusion. To make this output easier to read, we’ll add some extra data about which subprocess generated the output before writing it to our application’s standard output. We’ll prepend the command that generated the output before printing it out.

To do this, the first thing we’ll need to do is set the `stdout` parameter to `asyncio.subprocess.PIPE`. This tells the subprocess to create a new `StreamReader` instance we can use to read output from the process. We can then access this stream reader with the `Proccess.stdout` field. Let’s try this with our `ls` `-la` command.

Listing 13.3 Using the standard output stream reader

```
import asyncio
from asyncio import StreamReader
from asyncio.subprocess import Process


async def write_output(prefix: str, stdout: StreamReader):
    while line := await stdout.readline():
        print(f'[{prefix}]: {line.rstrip().decode()}')


async def main():
    program = ['ls', '-la']
    process: Process = await asyncio.create_subprocess_exec(


    print(f'Process pid is: {process.pid}')
    stdout_task = asyncio.create_task(write_output(' '.join(

    return_code, _ = await asyncio.gather(process.wait(), st
    print(f'Process returned: {return_code}')


asyncio.run(main())
```

In the preceding listing, we first create a coroutine `write_output` to prepend a prefix to output from a stream reader line by line. Then, in our main coroutine, we create a subprocess specifying we want to pipe `stdout`. We also create a task to run `write_output`, passing in the process's standard output stream reader, and run this concurrently with `wait`. When running this, you'll see the output prepended with the command:

```
Process pid is: 56925
[ls -la]: total 32
[ls -la]: drwxr-xr-x   7 matthewfowler  staff  224 Dec 23 09
[ls -la]: drwxr-xr-x  25 matthewfowler  staff  800 Dec 23 14
[ls -la]: -rw-r--r--   1 matthewfowler  staff    0 Dec 23 14
Process returned: 0
```

One crucial aspect of using pipes, and dealing with subprocesses input and output in general, is that they are susceptible to deadlocks. The `wait` coroutine is especially susceptible to this if our subprocess generates a lot of output, and we don't properly consume it. To demonstrate this, let's look at a simple example by generating a Python application that writes a lot of data to standard output and flushes it all at once.

Listing 13.4 Generating a lot of output

```
import sys

[sys.stdout.buffer.write(b'Hello there!!\n') for _ in range(

sys.stdout.flush()
```

The preceding listing writes `Hello there!!` to the standard output buffer 1,000,000 times and flushes it all at once. Let's see what happens if we use a pipe with this application but don't consume the data.

Listing 13.5 A deadlock with pipes

```
import asyncio
from asyncio.subprocess import Process


async def main():
    program = ['python3', 'listing_13_4.py']
    process: Process = await asyncio.create_subprocess_exec(


    print(f'Process pid is: {process.pid}')

    return_code = await process.wait()
    print(f'Process returned: {return_code}')
```

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

If you run the preceding listing, you'll see the process `pid` printed out and then nothing more. The application will hang forever, and you'll need to forcefully terminate it. If this does not happen on your system, simply increase the number of times we output data in the output application, and you'll eventually run into the problem.

Our application seems simple enough, so why are we running into this deadlock? The issue lies in how the stream reader's buffer works. When the stream reader's buffer fills up, any more calls to write into it block until more space in the buffer becomes available. While our stream reader buffer is blocked because its buffer is full, our process is still trying to finish writing its large output to the stream reader. This makes our process dependent on the stream reader becoming unblocked, but the stream reader will never become unblocked because we never free up any space in the buffer. This is a circular dependency and therefore a deadlock.

Previously, we avoided this issue entirely by reading from the standard output stream reader concurrently as we were waiting for the process to finish. This meant that even if the buffer filled, we would drain it such that the process wouldn't block indefinitely waiting to write additional data. When dealing with pipes, you'll need to be careful about consuming stream data, so you don't run into deadlocks.

You can also address this issue by avoiding use of the `wait` coroutine. In addition, the `Process` class has another coroutine method called `communicate` that avoids deadlocks entirely. This coroutine blocks until the subprocess completes and concurrently consumes standard output and standard error, returning the output complete once the application finishes. Let’s adapt our previous example to use `communicate` to fix the issue.

Listing 13.6 Using `communicate`

```
import asyncio
from asyncio.subprocess import Process


async def main():
    program = ['python3', 'listing_13_4.py']
    process: Process = await asyncio.create_subprocess_exec(


    print(f'Process pid is: {process.pid}')

    stdout, stderr = await process.communicate()
    print(stdout)
    print(stderr)
    print(f'Process returned: {process.returncode}')


asyncio.run(main())
```

When you run the preceding listing, you’ll see all the application’s output printed to the console all at once (and `None` printed once, since we didn’t write anything to standard output). Internally, `communicate` creates a few tasks that constantly read output from standard output and standard error into an internal buffer, thus, avoiding any deadlock issues. While we avoid potential deadlocks, we have a serious drawback in that we can't interactively process output from standard output. If you're in a situation in which you need to react to output from an application (perhaps you need to terminate when you encounter a certain message or spawn another task), you'll need to use `wait`, but be careful to read output from your stream reader appropriately to avoid deadlocks.

An additional drawback is that communicate buffers *all* the data from standard output and standard input in memory. If you're working with a subprocess that could produce a large amount of data, you run the risk of running out of memory. We'll see how to address these shortcomings in the next section.

Running subprocesses concurrently

Now that we know the basics of creating, terminating, and reading output from subprocesses, it is added with our existing knowledge to run multiple applications concurrently. Let's imagine we need to encrypt multiple pieces of text that we have in memory, and for security purposes we'd like to use the Twofish cipher algorithm. This algorithm isn't supported by the `hashlib` module, so we'll need an alternative. We can use the `gpg` (short for GNU Privacy Guard, which is a free software replacement of PGP [pretty good privacy]) command-line application. You can download gpg at https://gnupg.org/download/.

First, let's define the command we'll want to use for our encryption. We can use gpg by defining a passcode and setting an algorithm with command line parameters. Then, it is a matter of echoing text to the application. For example, to encrypt the text "encrypt this!", we can run the following:

```
echo 'encrypt this!' | gpg -c --batch --passphrase 3ncryptm3
```

This should produce encrypted output to standard output similar to the following:

```
?
Q+??/??*??C??H`??`)R??u??7þ_{f{R;n?FE .?b5??(?i??????o\k?b<?
```

This will work on our command line, but it won't work if we're using `create_subprocess_exec`, since we won't have the `|` pipe operator available (`create_subprocess_shell` will work if you truly need a pipe). So how can we pass in the text we want to encrypt? In addition to allowing us to pipe standard output and standard error, `communicate` and `wait` let us pipe in standard input as well. The `communicate` coroutine also lets us specify input bytes when we start the application. If we've piped standard input when we created our process, these bytes will get sent to the application. This will work nicely for us; we'll simply pass the string we want to encrypt with the `communicate` coroutine.

Let's try this out by generating random pieces of text and encrypting them concurrently. We'll create a list of 100 random text strings with 1,000 characters each and run `gpg` on each of them concurrently.

Listing 13.7 Encrypting text concurrently

```
import asyncio
import random
import string
import time
from asyncio.subprocess import Process


async def encrypt(text: str) -> bytes:
    program = ['gpg', '-c', '--batch', '--passphrase', '3ncr
        '--cipher-algo', 'TWOFISH']

    process: Process = await asyncio.create_subprocess_exec(





    stdout, stderr = await process.communicate(text.encode()
    return stdout


async def main():
    text_list = [''.join(random.choice(string.ascii_letters)

    s = time.time()
    tasks = [asyncio.create_task(encrypt(text)) for text in
    encrypted_text = await asyncio.gather(*tasks)
    e = time.time()

    print(f'Total time: {e - s}')
    print(encrypted_text)


asyncio.run(main())
```

In the preceding listing, we define a coroutine called `encrypt` that creates a gpg process and sends in the text we want to encrypt with `communicate`. For simplicity, we just return the standard output result and don't do any error handling; in a real-world application you'd likely want to be more robust here. Then, in our main coroutine we create a list of random text and create an `encrypt` task for each piece of text. We then run them all concurrently with `gather` and print out the total runtime and encrypted bits of text. You can compare the concurrent runtime with the synchronous runtime by putting `await` in front of `asyncio.create_task` and removing the `gather`, and you should see a reasonable speedup.

In this listing, we only had 100 pieces of text. What if we had thousands or more? Our current code takes 100 pieces of text and tries to encrypt them all concurrently; this means that we create 100 processes at the same time. This poses a challenge because our machines are resource constrained, and one process could eat up a lot of memory. In addition, spinning up hundreds or thousands of processes creates nontrivial context-switching overhead.

In our case we have another wrinkle caused by gpg, since it relies on shared state to encrypt data. If you take the code in listing 13.7 and increase the number of pieces of text into the thousands, you'll likely start to see the following printed to standard error:

```
gpg: waiting for lock on `/Users/matthewfowler/.gnupg/random
```

So not only have we created a lot of processes and all the overhead associated with that, but we've also created processes that are actually blocked on shared state that gpg needs. So how can we limit the number of processes running to circumvent this issue? This is a perfect example of when a semaphore comes in handy. Since our work is CPU-bound, adding a semaphore to limit the number of processes to the number of CPU cores we have available makes sense. Let's try this out by using a semaphore that is limited to the number of CPU cores on our system and encrypting 1,000 pieces of text to see if this can improve our performance.

Listing 13.8 Subprocesses with a semaphore

```
import asyncio
import random
import string
import time
import os
from asyncio import Semaphore
from asyncio.subprocess import Process


async def encrypt(sem: Semaphore, text: str) -> bytes:
    program = ['gpg', '-c', '--batch', '--passphrase', '3ncr

    async with sem:
        process: Process = await asyncio.create_subprocess_e





        stdout, stderr = await process.communicate(text.enco
        return stdout


async def main():
    text_list = [''.join(random.choice(string.ascii_letters)
    semaphore = Semaphore(os.cpu_count())
    s = time.time()
```

```
        tasks = [asyncio.create_task(encrypt(semaphore, text)) f
        encrypted_text = await asyncio.gather(*tasks)
        e = time.time()

        print(f'Total time: {e - s}')


    asyncio.run(main())
```

Comparing this with the runtime of 1,000 pieces of text with an unbounded set of subprocesses you should see some performance improvement, alongside a reduction in memory usage. You might think this is similar to what we saw in chapter 6 with a `ProcessPoolExecutor`'s concept of maximum workers, and you'd be correct. Internally, a `ProcessPoolExecutor` uses a semaphore to manage how many processes run concurrently.

We've now seen the basics around creating, terminating, and running multiple subprocesses concurrently. Next, we'll take a look at how to communicate with subprocesses in a more interactive manner.

ommunicating with subprocesses

Up to this point, we've been using one-way, noninteractive communication with processes. But what if we're working with an application that may require user input? For example, we may be asked for a passphrase, username, or any other number of inputs.

In the case in which we know we only have one piece of input to deal with, using `communicate` is ideal. We saw this previously using gpg to send in text to encrypt, but let’s try it when the subprocess explicitly asks for input. We’ll first create a simple Python program to ask for a username and echo it to standard output.

Listing 13.9 Echoing user input

```
username = input('Please enter a username: ')
print(f'Your username is {username}')
```

Now, we can use `communicate` to input the username.

Listing 13.10 Using `communicate` with standard input

```
import asyncio
from asyncio.subprocess import Process


async def main():
    program = ['python3', 'listing_13_9.py']
    process: Process = await asyncio.create_subprocess_exec(






    stdout, stderr = await process.communicate(b'Zoot')
    print(stdout)
    print(stderr)


asyncio.run(main())
```

When we run this code, we'll see `b'Please enter a username: Your username is Zoot\n'` printed to the console, as our application terminates right after our first user input. This won't work if we have a more interactive application. For example, take this application, which repeatedly asks for user input and echoes it until the user types `quit`.

Listing 13.11 An echo application

```
user_input = ''

while user_input != 'quit':
    user_input = input('Enter text to echo: ')
    print(user_input)
```

Since `communicate` waits until the process terminates, we'll need to use `wait` and process standard output and standard input concurrently. The `Process` class exposes `StreamWriter` in a `stdin` field we can use when we've set standard input to `PIPE`. We can use this concurrently with the standard output `StreamReader` to handle these types of applications. Let's see how to do this with the following listing, where we'll create an application to write a few pieces of text to our subprocess.

Listing 13.12 Using the echo application with subprocesses

```
import asyncio
from asyncio import StreamWriter, StreamReader
from asyncio.subprocess import Process
```

```
async def consume_and_send(text_list, stdout: StreamReader,
    for text in text_list:
        line = await stdout.read(2048)
        print(line)
        stdin.write(text.encode())
        await stdin.drain()


async def main():
    program = ['python3', 'listing_13_11.py']
    process: Process = await asyncio.create_subprocess_exec(





    text_input = ['one\n', 'two\n', 'three\n', 'four\n', 'qu

    await asyncio.gather(consume_and_send(text_input, proces


asyncio.run(main())
```

In the preceding listing, we define a `consume_and_send` coroutine that reads standard output until we receive the expected message for a user to specify input. Once we've received this message, we dump the data to our own application's standard output and write the strings in `'text_list'` to standard input. We repeat this until we've sent all data into our subprocess. When we run this, we should see all of our output was sent to our subprocess and properly echoed:

```
b'Enter text to echo: '
b'one\nEnter text to echo: '
b'two\nEnter text to echo: '
b'three\nEnter text to echo: '
b'four\nEnter text to echo: '
```

The application we're currently running has the luxury of producing deterministic output and stopping at deterministic points to ask for input. This makes managing standard output and standard input relatively straightforward. What if the application we're running in a subprocess only asks for input sometimes or could write a lot of data before asking for input? Let's adapt our sample echo program to be a bit more complicated. We'll have it echo user input between 1 and 10 times randomly, and we'll `sleep` for a half second between each echo.

Listing 13.13 A more complex echo application

```
from random import randrange
import time

user_input = ''

while user_input != 'quit':
    user_input = input('Enter text to echo: ')
    for i in range(randrange(10)):
        time.sleep(.5)
        print(user_input)
```

If we run this application as a subprocess with a similar approach to listing 13.12, it will work because we're still deterministic in that we eventually ask for input with a known piece of text. However, the drawback of using this approach is that our code to read from standard output and write to standard input is strongly coupled. This combined with increasing complexity of our input/output logic can make the code hard to follow and maintain.

We can address this by decoupling reading standard output from writing data to standard input, thus, separating the concerns of reading standard output and writing to standard input. We'll create one coroutine to read standard output and one coroutine to write text to standard input. Our coroutine that reads standard output will set an event once it has received the input prompt we expect. Our coroutine that writes to standard input will wait for that event to be set, then once it is, it will write the specified text. We'll then take these two coroutines and run them concurrently with `gather`.

Listing 13.14 Decoupling output reading from input writing

```
import asyncio
from asyncio import StreamWriter, StreamReader, Event
from asyncio.subprocess import Process



async def output_consumer(input_ready_event: Event, stdout:
    while (data := await stdout.read(1024)) != b'':
        print(data)
        if data.decode().endswith("Enter text to echo: "):
            input_ready_event.set()
async def input_writer(text_data, input_ready_event: Event,
    for text in text_data:
        await input_ready_event.wait()
        stdin.write(text.encode())
```

```
            await stdin.drain()
            input_ready_event.clear()

async def main():
    program = ['python3', 'interactive_echo_random.py']
    process: Process = await asyncio.create_subprocess_exec(







    input_ready_event = asyncio.Event()

    text_input = ['one\n', 'two\n', 'three\n', 'four\n', 'qu

    await asyncio.gather(output_consumer(input_ready_event,
                         input_writer(text_input, input_read
                         process.stdin),
                         process.wait())


asyncio.run(main())
```

In the preceding listing, we first define an `output_consumer` coroutine function. This function takes in an `input_ready` event as well as a `StreamReader` that will reference standard output and reads from standard output until we encounter the text `Enter` `text` `to` `echo:`. Once we see this text, we know that the standard input of our subprocess is ready to accept input, so we set the `input_ready` event.

Our `input_writer` coroutine function iterates over our input list and waits on our event for standard input to become ready. Once standard input is ready, we write out our input and clear the event so that on the next iteration of our `for` loop we'll block until standard input becomes ready again. With this implementation we now have two coroutine functions, each with one clear responsibility: one to write to standard input and one to read to standard output, increasing the readability and maintainability of our code.

ary

- We can use asyncio's subprocess module to launch subprocesses asynchronously with `create_subprocess_shell` and `create_subprocess_exec`. Whenever possible, prefer `create_subprocess_exec`, as it ensures consistent behavior across machines.
- By default, output from subprocesses will go to our own application's standard output. If we need to read and interact with standard input and standard output, we'll need to configure them to pipe to `StreamReader` and `StreamWriter` instances.
- When we pipe standard output or standard error, we need to be careful to consume output. If we don't, we could deadlock our application.
- When we have a large amount of subprocesses to run concurrently, semaphores can make sense to avoid abusing system resources and creating unneeded contention.
- We can use the `communicate` coroutine method to send input to standard input on a subprocess.

14 Advanced asyncio

This chapter covers

- Designing APIs for both coroutines and functions
- Coroutine context locals
- Yielding to the event loop
- Using different event loop implementations
- The relationship between coroutines and generators
- Creating your own event loop with custom awaitables

We’ve learned the vast majority of what asyncio has to offer. Using the modules of asyncio covered in previous chapters, you should be able to complete almost any task you need. That said, there are still a few lesser-known techniques that you may need to use, especially if you’re designing your own asyncio APIs.

In this chapter, we’ll learn a smorgasbord of more advanced techniques available in asyncio. We’ll learn how to design APIs that can handle both coroutines and regular Python functions, how to force iterations of the event loop, and how to pass state between tasks without ever passing arguments. We’ll also dig into more details on how exactly asyncio uses generators to fully understand what is happening under the hood. We’ll do this by implementing our own custom awaitables and using them to build our own simple implementation of an event loop that can run multiple coroutines concurrently.

You’re not likely to need a lot of what is covered in this chapter in your day-to-day development tasks unless you’re building new APIs or frameworks that rely on the internals of asynchronous programming. These techniques are primarily for these applications and for the curious who'd like a deeper understanding of the internals of asynchronous Python.

PIs with coroutines and functions

If we're building an API on our own, we may not want to assume that our users are using our library in their own asynchronous application. They may not have migrated yet, or they may not get any benefits from an async stack and will never migrate. How can we design an API that accepts both coroutines and plain old Python functions to accommodate these types of users?

asyncio provides a couple of convenience functions to help us do this: `asyncio.iscoroutine` and `asyncio.iscoroutinefunction`. These functions let us test if a callable object is a coroutine or not, letting us apply different logic based on this. These functions are the basis of how Django can handle both synchronous and asynchronous views seamlessly, as we saw in chapter 9.

To see this, let's build a sample task runner class that accepts both functions and coroutines. This class will let users add functions to an internal list that we'll then run concurrently (if they are coroutines) or serially (if they are normal functions) when the user calls a start method on our task runner.

Listing 14.1 A task runner class

```
import asyncio


class TaskRunner:
    def __init__(self):
        self.loop = asyncio.new_event_loop()
        self.tasks = []

    def add_task(self, func):
        self.tasks.append(func)

    async def _run_all(self):
        awaitable_tasks = []

        for task in self.tasks:
            if asyncio.iscoroutinefunction(task):
                awaitable_tasks.append(asyncio.create_task(t
            elif asyncio.iscoroutine(task):
                awaitable_tasks.append(asyncio.create_task(t
            else:
                self.loop.call_soon(task)

        await asyncio.gather(*awaitable_tasks)

    def run(self):
        self.loop.run_until_complete(self._run_all())


if __name__ == "__main__":

    def regular_function():
        print('Hello from a regular function!')


    async def coroutine_function():
        print('Running coroutine, sleeping!')
        await asyncio.sleep(1)
        print('Finished sleeping!')


    runner = TaskRunner()
    runner.add_task(coroutine_function)
    runner.add_task(coroutine_function())
    runner.add_task(regular_function)

    runner.run()
```

In the preceding listing, our task runner creates a new event loop and an empty task list. We then define an `add` method which just adds a function (or coroutine) to the pending task list. Then, once a user calls `run()`, we kick off the `_run_all` method in the event loop. Our `_run_all` method iterates through our task list and checks to see if the function in question is a coroutine. If it is a coroutine, we create a task; otherwise, we use the event loops `call_soon` method to schedule our plain function to run on the next iteration of the event loop. Then once we've created all the tasks we need to, we call gather on them and wait for them all to finish.

We then define two functions: a normal Python function aptly named `regular_function` and a coroutine named `coroutine_function`. We create an instance of `TaskRunner` and add three tasks, calling `coroutine_function` twice to demonstrate the two different ways we can reference a coroutine in our API. This gives us the following output:

```
Running coroutine, sleeping!
Running coroutine, sleeping!
Hello from a regular function!
```

```
Finished sleeping!
Finished sleeping!
```

This demonstrates that we've successfully run both coroutines and normal Python functions. We've now built an API which can handle both coroutines as well as normal Python functions, increasing the ways available for our end users to consume our API. Next, we'll look at context variables, which let us store a state that is local to a task without having to explicitly pass it around as a function argument.

ontext variables

Imagine we're using a REST API built with thread-per-request web server. When a request to the web server comes in, we may have common data about the user making the request we need to keep track of, such as a user ID, access token, or other information. We may be tempted to store this data globally across all threads in the web servers, but this has drawbacks. Chief among the drawbacks is that we need to handle the mapping from a thread to its data as well as any locking to prevent race conditions. We can resolve this by using a concept called *thread locals*. Thread locals are global state that are specific to one thread. This data we set in a thread local will be seen by the thread that set it and only by that thread, avoiding any thread to data mapping as well as race conditions. While we won't go into to details of thread locals, you can read more about them in the documentation for the threading module available at https://docs.python.org/3/library/threading.html#thread-local-data.

Of course, in asyncio applications we usually have only one thread, so anything we store as a thread local is available anywhere in our application. In PEP-567 (https://www.python.org/dev/peps/pep-0567) the concept of *context variables* was introduced to handle the concept of a thread local within a single-threaded concurrency model.

Context variables are similar to thread locals with the difference being that they are local to a particular task instead of to a thread. This means that if a task creates a context variable, any inner coroutine or task within that initial task will have access to that variable. No other tasks outside of that chain will be able to see or modify the variable. This lets us keep track of a state specific to one task without ever having to pass it as an explicit argument.

To see an example of this, we'll create a simple server that listens for data from connected clients. We'll create a context variable to keep track of a connected user's address, and when a user sends a message, we'll print out their address along with the message they sent.

Listing 14.2 A server with a context variables

```
import asyncio
from asyncio import StreamReader, StreamWriter
from contextvars import ContextVar


class Server:
    user_address = ContextVar('user_address')

    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port

    async def start_server(self):
        server = await asyncio.start_server(self._client_con
                                            self.host, self.
        await server.serve_forever()
    def _client_connected(self, reader: StreamReader, writer
        self.user_address.set(writer.get_extra_info('peernam
        asyncio.create_task(self.listen_for_messages(reader)

    async def listen_for_messages(self, reader: StreamReader
        while data := await reader.readline():
            print(f'Got message {data} from {self.user_addre


async def main():
    server = Server('127.0.0.1', 9000)
    await server.start_server()


asyncio.run(main())
```

❶ Create a context variable with the name 'user_address'.

❷ When a client connects, store the client's address in the context variable.

❸ Display the user's message alongside their address from the context variable.

In the preceding listing, we first create an instance of a `ContextVar` to hold our user's address information. Context variables require us to provide a string name, so here we give it a descriptive name of `user_address`, primarily for debugging purposes. Then in our `_client_connected` callback, we set the data of the context variable to the client's address. This will allow any tasks spawned from this parent task to have access to the information we set; in this instance, this will be tasks that listen for messages from the clients.

In our `listen_for_messages` coroutine method we listen for data from our clients, and when we get it, we print it out alongside the address that we stored in our context variable. When you run this application and connect with multiple clients and send some messages, you should see output like the following:

```
Got message b'Hello!\r\n' from ('127.0.0.1', 50036)
Got message b'Okay!\r\n' from ('127.0.0.1', 50038)
```

Note that the port number of the address is different, indicating we got the message from two separate clients on localhost. Even though we created only one context variable, we're still able to access unique data specific to each individual client. This gives us a clean way to pass data among tasks without having to explicitly pass data into tasks or other method calls within that task.

orcing an event loop iteration

How the event loop operates internally is mostly outside of our control. It decides when and how to execute coroutines and tasks. That said, there is a way to trigger an event loop iteration if we need to do so. This can come in handy for long-running tasks to avoid blocking the event loop (though you should also consider threads in this case) or ensuring that a task starts instantly.

Recall that if we're creating several tasks, none of them will start to run until we hit an `await`, which will trigger the event loop to schedule and start to run them. What if we wanted each task to start running right away?

asyncio provides an optimized idiom to suspend the current coroutine and force an iteration of the event loop by passing in zero to `asyncio.sleep`. Let's see how to use this to start running tasks as soon as we create them. We’ll create two functions: one that does not use `sleep` and one which does to compare the order in which things run.

Listing 14.3 Forcing an event loop iteration

```
import asyncio
from util import delay


async def create_tasks_no_sleep():
    task1 = asyncio.create_task(delay(1))
    task2 = asyncio.create_task(delay(2))
    print('Gathering tasks:')
    await asyncio.gather(task1, task2)


async def create_tasks_sleep():
    task1 = asyncio.create_task(delay(1))
    await asyncio.sleep(0)
    task2 = asyncio.create_task(delay(2))
    await asyncio.sleep(0)
    print('Gathering tasks:')
    await asyncio.gather(task1, task2)


async def main():
    print('--- Testing without asyncio.sleep(0) ---')
    await create_tasks_no_sleep()
    print('--- Testing with asyncio.sleep(0) ---')
    await create_tasks_sleep()
```

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

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

```
--- Testing without asyncio.sleep(0) ---
Gathering tasks:
sleeping for 1 second(s)
sleeping for 2 second(s)
finished sleeping for 1 second(s)
finished sleeping for 2 second(s)
--- Testing with asyncio.sleep(0) ---
sleeping for 1 second(s)
sleeping for 2 second(s)
Gathering tasks:
finished sleeping for 1 second(s)
finished sleeping for 2 second(s)
```

We first create two tasks and then `gather` them without using `asyncio.sleep(0)`, and this runs as we would normally expect with our two `delay` coroutines not running until our `gather` statement. Next, we insert an `asyncio.sleep(0)` after we create each task. In the output, you'll notice that the messages from the `delay` coroutine print immediately before we call `gather` on the tasks. Using the `sleep` forces an event loop iteration, which causes the code in our tasks to execute immediately.

We've been almost exclusively using the asyncio implementation of an event loop. However, other implementations exist that we can swap in if we have a need. Next, let's look at how to use different event loops.

sing different event loop implementations

asyncio provides a default implementation of an event loop that we have been using up until this point, but it is entirely possible to use a different implementation that may have different characteristics. There are a few ways to use a different implementation. One is to subclass the `AbstractEventLoop` class and implement its methods, create an instance of this, then set it as the event loop with the `asyncio.set_event_loop` function. If we're building our own custom implementation this makes sense, but there are off-the-shelf event loops we can use. Let's look at one such implementation called *uvloop*.

So, what is uvloop, and why would you want to use it? uvloop is an implementation of an event loop that heavily relies on the `libuv` library (https://libuv.org), which is the backbone of the `node.js` runtime. Since `libuv` is implemented in C, it has better performance than pure interpreted Python code. As a result, uvloop can be faster than the default asyncio event loop. It tends to perform particularly well when writing socket and stream-based applications. You can read more about benchmarks on the project's github site at https://github.com/magicstack/uvloop. Note that at the time of writing, uvloop is only available on *Nix platforms.

To get started, let's first install the latest version of uvloop with the following command:

```
pip -Iv uvloop==0.16.0
```

Once we've installed `libuv`, we're ready to use it. We'll make a simple echo server and will use the uvloop implementation of the event loop.

Listing 14.4 Using uvloop as an event loop

```
import asyncio
from asyncio import StreamReader, StreamWriter
import uvloop


async def connected(reader: StreamReader, writer: StreamWrit
    line = await reader.readline()
    writer.write(line)
    await writer.drain()
    writer.close()
    await writer.wait_closed()


async def main():
    server = await asyncio.start_server(connected, port=9000
    await server.serve_forever()


uvloop.install()      ❶
asyncio.run(main())
```

❶ Install the uvloop event loop.

In the preceding listing, we call `uvloop.install()`, which will switch out the event loop for us. We can do this manually with the following code instead of calling install if we'd like:

```
loop = uvloop.new_event_loop()
asyncio.set_event_loop(loop)
```

The important part is to call this *before* calling `asyncio.run(main()).` Internally, `asyncio.run` calls `get_event_loop` that will create an event loop if one does not exist. If we do this before properly installing uvloop, we'll get a typical asyncio event loop, and installation after the fact will have no effect.

You may want to complete an exercise to benchmark if an event loop such as uvloop helps performance characteristics of your application. The uvloop project on Github has code that will run benchmarking both in terms of throughput and requests per second.

We've now seen how to use an existing event loop implementation instead of the default event loop implementation. Next, we'll see how to create our own event loop completely outside of the confines of asyncio. This will let us gain a deeper understanding of how the asyncio event loop, as well as coroutines, tasks, and futures, work under the hood.

reating a custom event loop

One potentially non-obvious aspect of asyncio is that it is conceptually distinct from `async/await` syntax and coroutines. The coroutine class definition isn't even in the asyncio library module!

Coroutines and `async/await` syntax are concepts that are independent of the ability to execute them. Python comes with a default event loop implementation, asyncio, which is what we have been using to run them until now, but we could use any event loop implementation, even our own. In the previous section, we saw how we could swap out the asyncio event loop with different implementations with potentially better (or at least, different) runtime performance. Now, let's see how to build our own simple event loop implementation that can handle non-blocking sockets.

Coroutines and generators

Before `async` and `await` syntax were introduced in Python 3.5, the relationship between coroutines and generators was obvious. Let's build a simple coroutine which sleeps for 1 second with the old syntax using decorators and generators to understand this.

Listing 14.5 Generator-based coroutines

```
import asyncio


@asyncio.coroutine
def coroutine():
    print('Sleeping!')
    yield from asyncio.sleep(1)
    print('Finished!')


asyncio.run(coroutine())
```

Instead of the `async` keyword, we apply the `@asyncio.coroutine` decorator to specify the function is a coroutine function and instead of the `await` keyword we use the `yield from` syntax we're familiar with in generators. Currently, the `async` and `await` keywords are just syntactic sugar around this construct.

Generator-based coroutines are deprecated

Note that generator based coroutines are currently scheduled to be removed entirely in Python version 3.10. You may run into them in legacy codebases, but you should not write new async code in this style anymore.

So why do generators make sense at all for a single-threaded concurrency model? Recall that a coroutine needs to suspend execution when it runs into a blocking operation to allow other coroutines to run. Generators suspend execution when they hit a yield point, effectively pausing them midstream. This means if we have two generators, we can interleave their execution. We let the first generator run until it hits a yield point (or, in coroutine language, an `await` point), then we let the second generator run until it hits its yield point, and we repeat this until both generators are exhausted. To see this in action, let's build a very simple example that interleaves two generators, using some background methods we'll need to use to build our event loop.

Listing 14.6 Interleaving generator execution

```
from typing import Generator


def generator(start: int, end: int):
    for i in range(start, end):
        yield i


one_to_five = generator(1, 5)
```

```
five_to_ten = generator(5, 10)


def run_generator_step(gen: Generator[int, None, None]):
    try:
        return gen.send(None)
    except StopIteration as si:
        return si.value


while True:
    one_to_five_result = run_generator_step(one_to_five)
    five_to_ten_result = run_generator_step(five_to_ten)
    print(one_to_five_result)
    print(five_to_ten_result)

    if one_to_five_result is None and five_to_ten_result is
        break
```

1 Run one step of the generator.

2 Interleave execution of both generators.

In the preceding listing, we create a simple generator that counts from a start integer to an end integer, yielding values along the way. We then create two instances of that generator: one that counts from one to four and one that counts from five to nine.

We also create a convenience method, `run_generator_step`, to handle running one step of the generator. The generator class has a `send` method, which advances the generator to the next `yield` statement, running all code up to that point. After we call `send`, we can consider the generator paused until we call `send` again, letting us run code in other generators. The `send` method takes in a parameter for any values we want to send as arguments to the generator. Here we don't have any, so we just pass in `None`. Once a generator reaches its end, it raises a `StopIteration` exception. This exception contains any return value from the generator, and here we return it. Finally, we create a loop and run each generator step by step. This has the effect of interleaving the two generators at the same time, giving us the following output:

```
1
5
2
6
3
7
4
8
None
9
None
None
```

Imagine instead of yielding numbers, we yielded to some slow operation. Once the slow operation was complete, we could resume the generator, picking up where we left off, while other generators that aren't paused can run other code. This is the core of how the event loop works. We keep track of generators that have paused their execution on a slow operation. Then, any other generators can run while that other generator is paused. Once the slow operation is finished, we can wake up the previous generator by calling `send` on it again, advancing to its next yield point.

As mentioned, `async` and `await` are just syntactical sugar around generators. We can demonstrate this by creating a coroutine instance and calling `send` on it. Let's make an example with two coroutines that just print simple messages and a third coroutine, which calls the other two with `await` statements. We'll then use the generator's `send` method to see how to call our coroutines.

Listing 14.7 Using coroutines with send

```
async def say_hello():
    print('Hello!')


async def say_goodbye():
    print('Goodbye!')


async def meet_and_greet():
    await say_hello()
    await say_goodbye()


coro = meet_and_greet()

coro.send(None)
```

When we run the code in the preceding listing, we'll see the following output:

```
Hello!
Goodbye!
Traceback (most recent call last):
  File "chapter_14/listing_14_7.py", line 16, in <module>
    coro.send(None)
StopIteration
```

Calling `send` on our coroutine runs all our coroutines in `meet_and_greet`. Since there is nothing that we actually “pause” on waiting for a result, as all code is run right away, even in our `await` statements.

So how do we get a coroutine to pause and wake up on a slow operation? To do this, let’s define how to make a custom awaitable, so we can use `await` syntax instead of generator-style syntax.

Custom awaitables

How do we define awaitables, and how do they work under the hood? We can define an awaitable by implementing the `__await__` method on a class, but how do we implement this method? What should it even return?

The only requirement of the `__await__` method is that it returns an iterator, and by itself this requirement isn’t very helpful. Can we make the concept of an iterator make sense in the context of an event loop? To understand how this works, we’ll implement our own version of an asyncio `Future` we’ll call `CustomFuture`, which we’ll then use in our own event loop implementation.

Recall that a `Future` is a wrapper around a value that may be there at some point in the future, having two states: complete and incomplete. Imagine we’re in an infinite event loop, and we want to check if a future is done with an iterator. If the operation is finished, we can just return the result and the iterator is done. If it isn’t done, we need some way of saying “I’m not finished; check me again later,” and in this case, the iterator can just yield itself!

This is how we’ll implement our `__await__` method for our `CustomFuture` class. If the result is not there yet, our iterator just returns the `CustomFuture` itself; if the result is there, we return the result, and the iterator is complete. If it isn’t done, we just yield `self`. If the result isn’t there, the next time we attempt to advance the iterator we run the code inside of the `__await__` again. In this implementation, we’ll also implement a method to add a callback to our future that runs when the value is set. We’ll need this later when implementing our event loop.

Listing 14.8 A custom future implementation

```
class CustomFuture:

    def __init__(self):
        self._result = None
        self._is_finished = False
        self._done_callback = None

    def result(self):
        return self._result

    def is_finished(self):
        return self._is_finished

    def set_result(self, result):
        self._result = result
        self._is_finished = True
        if self._done_callback:
                self._done_callback(result)

    def add_done_callback(self, fn):
        self._done_callback = fn

    def __await__(self):
        if not self._is_finished:
            yield self
        return self.result()
```

In the preceding listing, we define our `CustomFuture` class with `__await__` defined alongside methods to set the result, get the result, and add a callback. Our `__await__` method checks to see if the future is finished. If it is, we just return the result, and the iterator is done. If it is not finished, we return `self`, meaning our iterator will continue to infinitely return itself until the value is set. In terms of generators, this means we can keep calling `__await__` forever until someone sets the value for us.

Let's look at a small example of this to get a sense for how the flow might look in an event loop. We'll create a custom future and set the value of it after a few iterations, calling `__await__` at each iteration.

Listing 14.9 The custom future in a loop

```
from listing_14_8 import CustomFuture

future = CustomFuture()

i = 0
```

```
while True:
    try:
        print('Checking future...')
        gen = future.__await__()
        gen.send(None)
        print('Future is not done...')
        if i == 1:
            print('Setting future value...')
            future.set_result('Finished!')
        i = i + 1
    except StopIteration as si:
        print(f'Value is: {si.value}')
        break
```

In the preceding listing, we create a custom future and a loop that calls the `await` method and then attempts to advance the iterator. If the future is done, a `StopIteration` exception gets thrown with the result of the future. Otherwise, our iterator will just return the future, and we move on to the next iteration of the loop. In our example, we set the value after a couple of iterations, giving us the following output:

```
Checking future...
Future is not done...
Checking future...
Future is not done...
Setting future value...
Checking future...
Value is: Finished!
```

This example is just to reinforce the way to think about awaitables, we wouldn't write code like this in real life, as we'd normally want something else to set the result of our future. Next, let's extend this to do something more useful with sockets and the selector module.

Using sockets with futures

In chapter 3, we learned a bit about the `selector` module, which lets us register callbacks to be run when a socket event, such as a new connection or data being ready to read, occurs. Now we'll expand on this knowledge by using our custom `future` class to interact with selectors, setting results on futures when socket events occur.

Recall that selectors let us register callbacks to run when an event, such as a read or write, occurs on a socket. This concept nicely fits in with the future we've built. We can register the `set_result` method as a callback when a read happens on a socket. When we want to asynchronously wait for a result from a socket, we create a new future, register that future's `set_result` method with the selector module for that socket, and return the future. We can then await it, and we'll get the result when the selector calls the callback for us.

To see this in action, let's build an application that listens for a connection from a non-blocking socket. Once we get a connection, we'll just return it and let the application terminate.

Listing 14.10 Sockets with custom futures

```
import functools
import selectors
import socket
```

```
from listing_14_8 import CustomFuture
from selectors import BaseSelector


def accept_connection(future: CustomFuture, connection: sock
    print(f'We got a connection from {connection}!')
    future.set_result(connection)


async def sock_accept(sel: BaseSelector, sock) -> socket:
    print('Registering socket to listen for connections')
    future = CustomFuture()
    sel.register(sock, selectors.EVENT_READ, functools.parti
    print('Pausing to listen for connections...')
    connection: socket = await future
    return connection


async def main(sel: BaseSelector):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,

    sock.bind(('127.0.0.1', 8000))
    sock.listen()
    sock.setblocking(False)

    print('Waiting for socket connection!')
    connection = await sock_accept(sel, sock)
    print(f'Got a connection {connection}!')
selector = selectors.DefaultSelector()

coro = main(selector)
```

```
while True:
    try:
        state = coro.send(None)

        events = selector.select()

        for key, mask in events:
            print('Processing selector events...')
            callback = key.data
            callback(key.fileobj)
    except StopIteration as si:
        print('Application finished!')
        break
```

❶ Set the connection socket on the future when a client connects.

❷ Register the accept_connection function with the selector and pause to wait for a client connection.

❸ Wait for the client to connect.

❹ Loop forever, calling send on the main coroutine. Each time a selector event occurs, run the registered callback.

In the preceding listing, we first define an `accept_connection` function. This function takes in a `CustomFuture` as well as a client socket. We print a message that we have a socket and then set that socket as the result of the future. We then define `sock_ accept`; this function takes a server socket as well as a selector and registers `accept_ connection` (bound to a `CustomFuture`) as a callback on read events from the server socket. We then `await` the future, pausing until we have a connection, and then return it.

We then define a main coroutine function. In this function, we create a server socket and then `await` the `sock_accept` coroutine until we receive a connection, logging a message and terminating once we do so. With this we can build a *minimally viable event loop*. We create an instance of our main coroutine function, passing in a selector and then loop forever. In our loop, we first call `send` to advance our main coroutine to its first `await` statement, then we call `selector.select`, which will block until a client connects. Then we call any registered callbacks; in our case, this will always be `accept_connection`. Once someone connects, we'll call send a second time, which will advance all coroutines again and will let our application finish. If you run the following code and connect over Telnet, you should see output similar to the following:

```
Waiting for socket connection!
Registering socket to listen for connections
Pausing to listen for connections...
Processing selector events...
We got a connection from <socket.socket fd=4, family=Address
Got a connection <socket.socket fd=4, family=AddressFamily.A
Application finished!
```

We've now built a basic asynchronous application using only the `async` and `await` keywords without any asyncio! Our `while` loop at the end is a simplistic event loop and demonstrates the key concept of how the asyncio event loop works. Of course, we can't do too much concurrently without the ability to create tasks.

A task implementation

Tasks are a combination of a future and a coroutine. A task's future is complete when the coroutine it wraps finishes. With inheritance, we can wrap a coroutine in a future by subclassing our `CustomFuture` class and writing a constructor that takes a coroutine, but we still need a way to *run* that coroutine. We can do this by building a method we'll call `step` that will call the coroutine's send method and keep track of the result, effectively running one step of our coroutine per call.

One thing we'll need to keep in mind as we implement this method is that `send` may also return other futures. To handle this, we'll need to use the `add_done_callback` method of any futures that `send` returns. We'll register a callback that will call `send` on the task's coroutine with the resulting value when the future is finished.

Listing 14.11 A task implementation

```
from chapter_14.listing_14_8 import CustomFuture


class CustomTask(CustomFuture):

    def __init__(self, coro, loop):
        super(CustomTask, self).__init__()
        self._coro = coro
        self._loop = loop
        self._current_result = None
        self._task_state = None
        loop.register_task(self)
    def step(self):
        try:
            if self._task_state is None:
                self._task_state = self._coro.send(None)
            if isinstance(self._task_state, CustomFuture):
                self._task_state.add_done_callback(self._fut
        except StopIteration as si:
            self.set_result(si.value)

    def _future_done(self, result):
        self._current_result = result
        try:
            self._task_state = self._coro.send(self._current
        except StopIteration as si:
            self.set_result(si.value)
```

1 Register the task with the event loop.

2 Run one step of the coroutines.

3 If the coroutine yields a future, add a done callback.

4 Once the future is done, send the result to the coroutine.

In the preceding listing, we subclass `CustomFuture` and create a constructor that accepts a coroutine and an event loop, registering the task with the loop by calling `loop.register_task`. Then, in our `step` method, we call send on the coroutine, and if the coroutine yields a `CustomFuture`, we add a `done` callback. In this case, our `done` callback will take the result of the future and send it to the coroutine we wrap, advancing it when the future is complete.

Implementing an event loop

We now know how to run coroutines and have created implementations of both futures and tasks, giving us all the building blocks we need to build an event loop. What does our event API need to look like to build an asynchronous socket application? We'll need a few methods with different purposes:

- We'll need a method to accept a main entry coroutine, much like `asyncio.run`.
- We'll need methods to accept connections, receive data, and close a socket. These methods will register and deregister sockets with a selector.
- We'll need a method to register a `CustomTask`; this is just an implementation of the method we used in the `CustomTask` constructor previously.

First, let's talk about our main entry point; we'll call this method `run`. This is the powerhouse of our event loop. This method will take a main entrypoint coroutine and call `send` on it, keeping track of the result of the generator in an infinite loop. If the main coroutine produces a future, we'll add a `done` callback to keep track of the result of the future once it is complete. Once we do this, we'll run the `step` method of any registered tasks and then call the selector waiting for any socket events to fire. Once they run, we'll run the associated callbacks and trigger another iteration of the loop. If at any point our main coroutine throws a `StopIteration` exception, we know our application is finished, and we can exit returning the value inside the exception.

Next, we'll need coroutine methods to accept socket connections and receive data from a client socket. Our strategy here will be to create a `CustomFuture` instance that a callback will set the result of, registering this callback with the selector to fire on read events. We'll then await this future.

Finally, we'll need a method to register tasks with the event loop. This method will simply take a task and add it to a list. Then, on each iteration of the event loop we'll call `step` on any tasks we've registered with the event loop, advancing them if they are ready. Implementing all of this will yield a minimum viable event loop.

Listing 14.12 An event loop implementation

```
import functools
import selectors
from typing import List
from chapter_14.listing_14_11 import CustomTask
from chapter_14.listing_14_8 import CustomFuture


class EventLoop:
    _tasks_to_run: List[CustomTask] = []
    def __init__(self):
        self.selector = selectors.DefaultSelector()
        self.current_result = None

    def _register_socket_to_read(self, sock, callback):
        future = CustomFuture()
        try:
            self.selector.get_key(sock)
        except KeyError:
            sock.setblocking(False)
            self.selector.register(sock, selectors.EVENT_REA
        else:
            self.selector.modify(sock, selectors.EVENT_READ,
        return future
    def _set_current_result(self, result):
        self.current_result = result

    async def sock_recv(self, sock):
        print('Registering socket to listen for data...')
        return await self._register_socket_to_read(sock, sel

    async def sock_accept(self, sock):
        print('Registering socket to accept connections...')
        return await self._register_socket_to_read(sock, sel

    def sock_close(self, sock):
        self.selector.unregister(sock)
        sock.close()

    def register_task(self, task):
        self._tasks_to_run.append(task)

    def recieved_data(self, future, sock):
        data = sock.recv(1024)
        future.set_result(data)

    def accept_connection(self, future, sock):
        result = sock.accept()
        future.set_result(result)

    def run(self, coro):
        self.current_result = coro.send(None)

        while True:
            try:
                if isinstance(self.current_result, CustomFut
```