```
asyncio.run(coroutine(), debug=True)
```

Using command-line arguments

Debug mode can be enabled by passing a command-line argument when we start our Python application. To do this we apply `-X dev`:

```
python3 -X dev program.py
```

Using environment variables

We can also use environment variables to enable debug mode by setting the `PYTHONASYNCIODEBUG` variable to `1`:

```
PYTHONASYINCIODEBUG=1 python3 program.py
```

NOTE In versions of Python older than 3.9, there is a bug within debug mode. When using `asyncio.run,` only the `boolean` debug parameter will work. Command-line arguments and environment variables will only work when manually managing the event loop.

In debug mode, we'll see informative messages logged when a coroutine takes too long. Let's test this out by trying to run CPU-bound code in a task to see if we get a warning, as shown in the following listing.

Listing 2.23 Running CPU-bound code in debug mode

```
import asyncio
from util import async_timed


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


async def main() -> None:
    task_one = asyncio.create_task(cpu_bound_work())
    await task_one

asyncio.run(main(), debug=True)
```

When running this, we'll see a helpful message that `task_one` was taking too long, therefore blocking the event loop from running any other tasks:

```
Executing <Task finished name='Task-2' coro=<cpu_bound_work(
```

This can be helpful for debugging issues where we may inadvertently be making a call that is blocking. The default settings will log a warning if a coroutine takes longer than 100 milliseconds, but this may be longer or shorter than we'd like. To change this value, we can set the slow callback duration by accessing the event loop as we do in listing 2.24 and setting `slow_callback_duration`. This is a floating-point value representing the seconds we want the slow callback duration to be.

Listing 2.24 Changing the slow callback duration

```
import asyncio

async def main():
    loop = asyncio.get_event_loop()
    loop.slow_callback_duration = .250


asyncio.run(main(), debug=True)
```

The preceding listing will set the slow callback duration to 250 milliseconds, meaning we'll get a message printed out if any coroutine takes longer than 250 milliseconds of CPU time to run.

ary

- We've learned how to create coroutines with the `async` keyword. Coroutines can suspend their execution on a blocking operation. This allows for other coroutines to run. Once the operation where the coroutine suspended completes, our coroutine will wake up and resume where it left off.
- We learned to use `await` in front of a call to a coroutine to run it and wait for it to return a value. To do so, the coroutine with the `await` inside it will suspend its execution, while waiting for a result. This allows other coroutines to run while the first coroutine is awaiting its result.
- We've learned how to use `asyncio.run` to execute a single coroutine. We can use this function to run the coroutine that is the main entry point into our application.

- We've learned how to use tasks to run multiple long-running operations concurrently. Tasks are wrappers around coroutines that will then be run on the event loop. When we create a task, it is scheduled to run on the event loop as soon as possible.
- We've learned how to cancel tasks if we want to stop them and how to add a timeout to a task to prevent them from taking forever. Canceling a task will make it raise a `CancelledError` while we await it. If we have time limits on how long a task should take, we can set timeouts on it by using `asycio.wait_for`.
- We've learned to avoid common issues that newcomers make when using asyncio. The first is running CPU-bound code in coroutines. CPU-bound code will block the event loop from running other coroutines since we're still single-threaded. The second is blocking I/O, since we can't use normal libraries with asyncio, and you must use asyncio-specific ones that return coroutines. If your coroutine does not have an `await` in it, you should consider it suspicious. There are still ways to use CPU-bound and blocking I/O with asyncio, which we will address in chapters 6 and 7.
- We've learned how to use debug mode. Debug mode can help us diagnose common issues in asyncio code, such as running CPU-intensive code in a coroutine.

3 A first asyncio application

This chapter covers

- Using sockets to transfer data over a network
- Using telnet to communicate with a socket-based application
- Using selectors to build a simple event loop for non-blocking sockets
- Creating a non-blocking echo server that allows for multiple connections
- Handling exceptions in tasks
- Adding custom shutdown logic to an asyncio application

In chapters 1 and 2, we introduced coroutines, tasks, and the event loop. We also examined how to run long operations concurrently and explored some of asyncio's APIs that facilitate this. Up to this point however, we've only simulated long operations with the `sleep` function.

Since we'd like to build more than just demo applications, we'll use some real-world blocking operations to demonstrate how to create a server that can handle multiple users concurrently. We'll do this with only one thread, leading to a more resource-efficient and simpler application than other solutions that would involve threads or multiple processes. We'll take what we've learned about coroutines, tasks, and asyncio's API methods to build a working command-line echo server application using sockets to demonstrate this. By the end of this chapter, you'll be able to build socket-based network applications with asyncio that can handle multiple users simultaneously with one thread.

First, we'll learn the basics of how to send and receive data with blocking sockets. We'll then use these sockets to attempt building a multi-client echo server. In doing

so, we'll demonstrate that we can't build an echo server that works properly for more than one client at a time with only a single thread. We'll then learn how to resolve these issues by making our sockets non-blocking and using the operating system's event notification system. This will help us understand how the underlying machinery of the asyncio event loop works. Then we'll use asyncio's non-blocking socket coroutines to allow multiple clients to connect properly. This application will let multiple users connect simultaneously, letting them send and receive messages concurrently. Finally, we'll add custom shutdown logic to our application, so when our server shuts down, we'll give in-flight messages some time to complete.

›rking with blocking sockets

In chapter 1, we introduced the concept of sockets. Recall that a socket is a way to read and write data over a network. We can think of a socket as a mailbox: we put a letter in, and it is delivered to the recipient's address. The recipient can then read that message, and possibly send us another back.

To get started, we'll create the main mailbox socket, which we'll call our server socket. This socket will first accept connection messages from clients that want to communicate with us. Once that connection is acknowledged by our server socket, we'll create a socket that we can use to communicate with the client. This means our server starts to look more like a post office with multiple PO boxes rather than just one mailbox. The client side can still be thought of as having a single mailbox as they will have one socket to communicate with us. When a client connects to our server, we provide them a PO box. We then use that PO box to send and receive messages to and from that client (see figure 3.1).

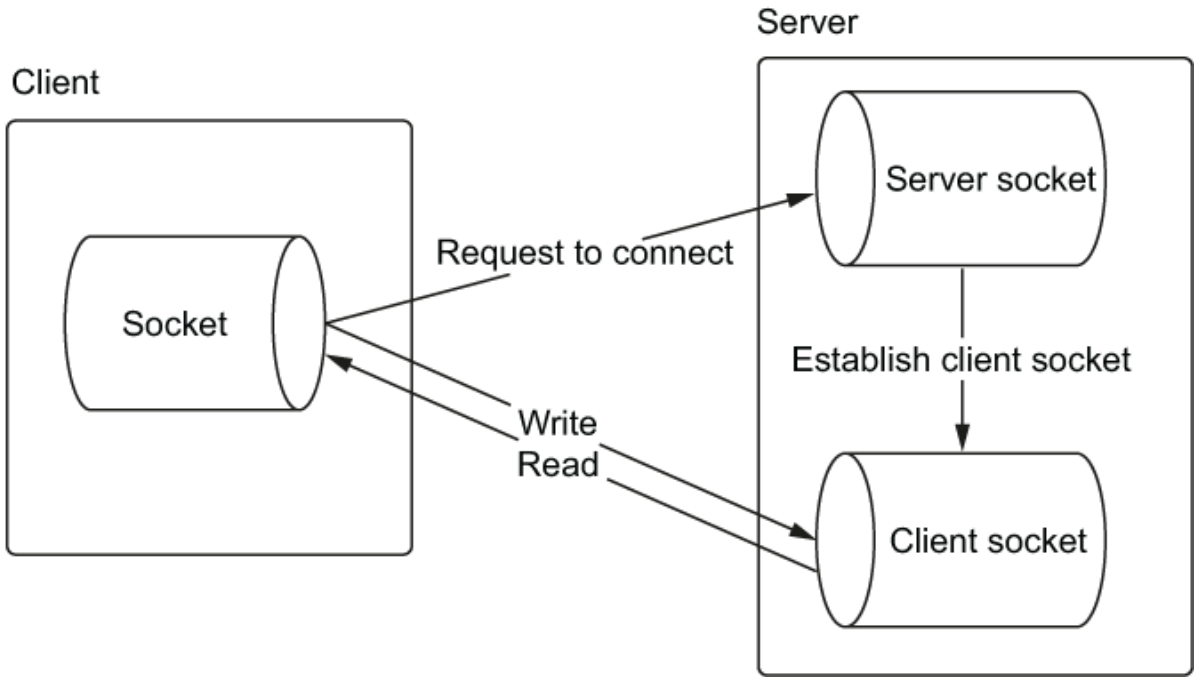

Figure 3.1 A client connects to our server socket. The server then creates a new socket to communicate with the client.

We can create this server socket with Python's built-in socket module. This module provides functionality for reading, writing, and manipulating sockets. To get started creating sockets, we'll create a simple server which listens for a connection from a client and prints a message on a successful connection. This socket will be bound to both a hostname and a port and will be the main "server socket" that any clients will communicate with.

It takes a few steps to create a socket. We first use the `socket` function to create a socket:

```
import socket
```

```
  server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
  server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
```

Here, we specify two parameters to the socket function. The first is `socket.AF_INET` —this tells us what type of address our socket will be able to interact with; in this case a hostname and a port number. The second is `socket.SOCK_STREAM`; this means that we use the TCP protocol for our communication.

What is the TCP protocol?

TCP, or transmission control protocol, is a protocol designed to transfer data between applications over a network. This protocol is designed with reliability in mind. It performs error checking, delivers data in order, and can retransmit data when needed. This reliability comes at the cost of some overhead. The vast majority of the web is built on TCP. TCP is in contrast to UDP, or user datagram protocol, which is less reliable but has much less overhead than TCP and tends to be more performant. We will exclusively focus on TCP sockets in this book.

We also call `setsockopt` to set the `SO_REUSEADDR` flag to `1`. This will allow us to reuse the port number after we stop and restart the application, avoiding any *address already in use* errors. If we didn't do this, it might take some time for the operating system to unbind this port and have our application start without error.

Calling `socket.socket` lets us create a socket, but we can't start communicating with it yet because we haven't bound it to an address that clients can talk to (our post office needs an address!). For this example, we'll bind the socket to an address on our own computer at `127.0.0.1`, and we'll pick an arbitrary port number of 8000:

```
address = (127.0.0.1, 8000)
server_socket.bind(server_address)
```

Now we've set our socket up at the address 127.0.0.1:8000. This means that clients will be able to use this address to send data to our server, and if we write data to a client, they will see this as the address that it's coming from.

Next, we need to actively listen for connections from clients who want to connect to our server. To do this, we can call the listen method on our socket. This tells the socket to listen for incoming connections, which will allow clients to connect to our server socket. Then, we wait for a connection by calling the accept method on our socket. This method will block until we get a connection and when we do, it will return a connection and the address of the client that connected. The connection is just another socket we can use to read data from and write data to our client:

```
server_socket.listen()
connection, client_address = server_socket.accept()
```

With these pieces, we have all the building blocks we need to create a socket-based server application that will wait for a connection and print a message once we have one.

Listing 3.1 Starting a server and listening for a connection

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
```

```
server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

connection, client_address = server_socket.accept()
print(f'I got a connection from {client_address}!')
```

❶ Create a TCP server socket.

❷ Set the address of the socket to 127.0.0.1:8000.

❸ Listen for connections or “open the post office.”

❹ Wait for a connection and assign the client a PO box.

In the preceding listing, when a client connects, we get their connection socket as well as their address and print that we got a connection.

So now that we’ve built this application, how do we connect to it to test it out? While there are quite a few tools for this, in this chapter we’ll use the telnet command-line application.

nnecting to a server with Telnet

Our simple example of accepting connections left us with no way to connect. There are many command-line applications to read and write data to and from a server, but a popular application that has been around for quite some time is Telnet.

Telnet was first developed in 1969 and is short for “teletype network.” Telnet establishes a TCP connection to a server and a host we specify. Once we do so, a terminal is established and we’re free to send and receive bytes, all of which will be displayed in the terminal.

On Mac OS you can install telnet with Homebrew with the command `brew install telnet` (see https://brew.sh/ to install Homebrew). On Linux distributions you will need to use the system package manager to install (`apt-get install telnet` or similar). On Windows, PuTTy is the best option, and you can download this from https://putty.org.

NOTE With PuTTY you’ll need to turn on local line editing for code samples in this book to work. To do this go to *Terminal* on the left-hand side of the PuTTy configuration window and set *Local line editing* to *Force on*.

To connect to the server we built in listing 3.1, we can use the Telnet command on a command line and specify that we’d like to connect to localhost on port 8000:

```
telnet localhost 8000
```

Once we do this, we’ll see some output on our terminal telling us that we’ve successfully connected. Telnet then will display a cursor, which allows us to type and select [Enter] to send data to the server.

```
telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

In the console output of our server application, we should now see output like the following, showing that we've established a connection with our Telnet client:

```
I got a connection from ('127.0.0.1', 56526)!
```

You'll also see a `Connection closed by foreign host` message as the server code exits, indicating the server has shut down the connection to our client. We now have a way to connect to a server and write and read bytes to and from it, but our server can't read or send any data itself. We can do this with our client socket's `sendall` and `recv` methods.

Reading and writing data to and from a socket

Now that we've created a server capable of accepting connections, let's examine how to read data from our connections. The socket class has a method named `recv` that we can use to get data from a particular socket. This method takes an integer representing the number of bytes we wish to read at a given time. This is important because we can't read all data from a socket at once; we need to buffer until we reach the end of the input.

In this case, we'll treat the end of input as a carriage return plus a line feed or `'\r\n'`. This is what gets appended to the input when a user presses [Enter] in telnet. To demonstrate how buffering works with small messages, we'll set a buffer size intentionally low. In a real-world application, we would use a larger buffer size, such as 1024 bytes. We would typically want a larger buffer size, as this will take advantage of the buffering that occurs at the operating system-level, which is more efficient than doing it in your application.

Listing 3.2 Reading data from a socket

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

try:
    connection, client_address = server_socket.accept()
    print(f'I got a connection from {client_address}!')

    buffer = b''

    while buffer[-2:] != b'\r\n':
        data = connection.recv(2)
        if not data:
            break
        else:
            print(f'I got data: {data}!')
            buffer = buffer + data

    print(f"All the data is: {buffer}")
finally:
    server_socket.close()
```

In the preceding listing, we wait for a connection with `server_socket.accept`, as before. Once we get a connection, we try to receive two bytes and store it in our buffer. Then, we go into a loop, checking each iteration to see if our buffer ends in a

carriage return and a line feed. If it does not, we get two more bytes and print out which bytes we received and append that to the buffer. If we get `'\r\n'`, then we end the loop and we print out the full message we got from the client. We also close the server socket in a `finally` block. This ensures that we close the connection even if an exception occurs while reading data. If we connect to this application with telnet and send a message `'testing123'`, we'll see this output:

```
I got a connection from ('127.0.0.1', 49721)!
I got data: b'te'!
I got data: b'st'!
I got data: b'in'!
I got data: b'g1'!
I got data: b'23'!
I got data: b'\r\n'!
All the data is: b'testing123\r\n'
```

Now, we're able to read data from a socket, but how do we write data back to a client? Sockets have a method named `sendall` that will take a message and write it back to the client for us. We can adapt our code in listing 3.2 to echo the message the client sent to us by calling `connection.sendall` with the buffer once it is filled:

```
    while buffer[-2:] != b'\r\n':
        data = connection.recv(2)
        if not data:
            break
        else:
            print(f'I got data: {data}!')
            buffer = buffer + data
```

```
        print(f"All the data is: {buffer}")
        connection.sendall(buffer)
```

Now when we connect to this application and send it a message from Telnet, we should see that message printed back on our telnet terminal. We've created a very basic echo server with sockets!

This application handles one client at a time right now, but multiple clients can connect to a single server socket. Let's adapt this example to allow multiple clients to connect at the same time. In doing this we'll demonstrate how we can't properly support multiple clients with blocking sockets.

Allowing multiple connections and the dangers of blocking

A socket in listen mode allows multiple client connections simultaneously. This means that we can call `socket.accept` repeatedly, and each time a client connects we will get a new connection socket to read and write data to and from that client. With that knowledge, we can straightforwardly adapt our previous example to handle multiple clients. We loop forever, calling `socket.accept` to listen for new connections. Each time we get one, we append it to a list of connections we've got so far. Then, we loop over each connection, receiving data as it comes in and writing that data back out to the client connection.

Listing 3.3 Allowing multiple clients to connect

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
```

```
server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()

connections = []

try:
    while True:
        connection, client_address = server_socket.accept()
        print(f'I got a connection from {client_address}!')
        connections.append(connection)
        for connection in connections:
            buffer = b''

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                if not data:
                    break
                else:
                    print(f'I got data: {data}!')
                    buffer = buffer + data

            print(f"All the data is: {buffer}")

            connection.send(buffer)
finally:
    server_socket.close()
```

We can try this by making one connection with telnet and typing a message. Then, once we have done that, we can connect with a second telnet client and send another

message. However, if we do this, we will notice a problem right away. Our first client will work fine and will echo messages back as we'd expect, but our second client won't get anything echoed back to it. This is due to the default blocking behavior of sockets. The methods `accept` and `recv` block until they receive data. This means that once the first client connects, we will block waiting for it to send its first echo message to us. This causes other clients to be stuck waiting for the next iteration of the loop, which won't happen until the first client sends us data (figure 3.2).

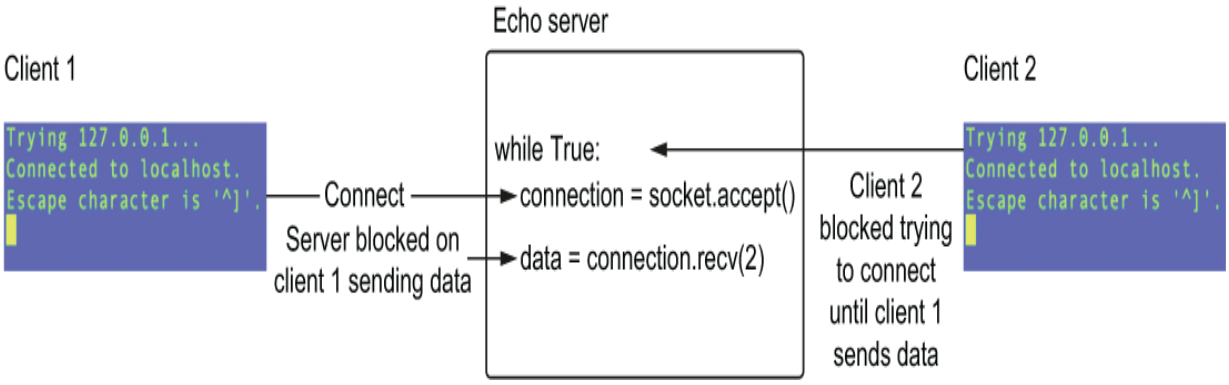

Figure 3.2 With blocking sockets, Client 1 connects, but Client 2 is blocked until client one sends data.

This obviously isn't a satisfactory user experience; we've created something that won't properly scale when we have more than one user. We can solve this issue by putting our sockets in non-blocking mode. When we mark a socket as non-blocking, its methods will not block waiting to receive data before moving on to execute the next line of code.

›rking with non-blocking sockets

Our previous echo server allowed multiple clients to connect; however, when more than one connected, we ran into issues where one client could cause others to wait for

it to send data. We can address this issue by putting sockets into non-blocking mode. When we do this, any time we call a method that would block, such as `recv`, it is guaranteed to return instantly. If the socket has data ready for processing, then we will get data returned as we would with a blocking socket. If not, the socket will instantly let us know it does not have any data ready, and we are free to move on to execute other code.

Listing 3.4 Creating a non-blocking socket

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
server_socket.bind(('127.0.0.1', 8000))
server_socket.listen()
server_socket.setblocking(False)
```

Fundamentally, creating a non-blocking socket is no different from creating a blocking one, except that we must call `setblocking` with `False`. By default, a socket will have this value set to `True`, indicating it is blocking. Now let's see what happens when we do this in our original application. Does this fix the issue?

Listing 3.5 A first attempt at a non-blocking server

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA
```

```
server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
server_socket.listen()
server_socket.setblocking(False)

connections = []

try:
    while True:
        connection, client_address = server_socket.accept()
        connection.setblocking(False)
        print(f'I got a connection from {client_address}!')
        connections.append(connection)

        for connection in connections:
            buffer = b''

            while buffer[-2:] != b'\r\n':
                data = connection.recv(2)
                if not data:
                    break
                else:
                    print(f'I got data: {data}!')
                    buffer = buffer + data

            print(f"All the data is: {buffer}")
            connection.send(buffer)
finally:
    server_socket.close()
```

❶ Mark the server socket as non-blocking.

❷ Mark the client socket as non-blocking.

When we run listing 3.5, we'll notice something different right away. Our application crashes almost instantly! We'll get thrown a `BlockingIOError` because our server socket has no connection yet and therefore no data to process:

```
Traceback (most recent call last):
  File "echo_server.py", line 14, in <module>
    connection, client_address = server_socket.accept()
  File " python3.8/socket.py", line 292, in accept
    fd, addr = self._accept()
BlockingIOError: [Errno 35] Resource temporarily unavailable
```

This is the socket's somewhat unintuitive way of telling us, "I don't have any data, try calling me again later." There is no easy way for us to tell if a socket has data right now, so one solution is to just catch the exception, ignore it, and keep looping until we have data. With this tactic, we'll constantly be checking for new connections and data as fast as we can. This should solve the issue that our blocking socket echo server had.

Listing 3.6 Catching and ignoring blocking IO errors

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_ST
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEA

server_address = ('127.0.0.1', 8000)
server_socket.bind(server_address)
```