```
                self.current_result.add_done_callback(se
                if self.current_result.result() is not N
                    self.current_result = coro.send(self
            else:
                self.current_result = coro.send(self.cur
        except StopIteration as si:
            return si.value

        for task in self._tasks_to_run:
            task.step()

        self._tasks_to_run = [task for task in self._tas

        events = self.selector.select()
        print('Selector has an event, processing...')
        for key, mask in events:
            callback = key.data
            callback(key.fileobj)
```

❶ Register a socket with the selector for read events.

❷ Register a socket to receive data from a client.

❸ Register a socket to accept connections from a client.

❹ Register a task with the event loop.

❺ Run a main coroutine until it finishes, executing any pending tasks at each iteration.

We first define a `_register_socket_to_read` convenience method. This method takes in a socket and a callback and registers them with the selector if the socket isn't already registered. If the socket is registered, we replace the callback. The first argument to our callback needs to be a future, and in this method we create a new one and partially apply it to the callback. Finally, we return the `future` bound to the callback, meaning callers of our method can now await it and suspend execution until the callback is complete.

We then define coroutine methods to receive socket data and accept new client connections, `sock_recv`, and `sock_accept`, respectively. These methods call the `_register_socket_to_read` convenience method we just defined, passing in callbacks that handle data and new connections when they are available (these methods just set this data on a `future`).

Finally, we build our run method. This method accepts our main entry point coroutine and calls `send` on it, advancing it to its first suspension point and storing the result from `send`. We then kick off an infinite loop, first checking to see if the current result from the main coroutine is a `CustomFuture`; if it is, we register a callback to store the result, which we can then send back to the main coroutine if needed. If the result is not a `CustomFuture`, we just send it to the coroutine. Once we've controlled the flow of our main coroutine, we run any tasks that are registered with our event loop by calling `step` on them. Once we've run our tasks, we remove any that are finished from our task list.

Finally, we call `selector.select`, blocking until there are any events fired on the sockets we've registered. Once we have a socket event, or set of events, we loop through them, calling the callback we registered for that socket back in `_register_` `socket_to_read`. In our implementation, any socket event will trigger an iteration of the event loop. We've now implemented our `EventLoop` class, and we're ready to create our first asynchronous application without asyncio!

Implementing a server with a custom event loop

Now that we have an event loop, we'll build a very simple server application to log messages we receive from connected clients. We'll create a server socket and write a coroutine function to listen for connections in an infinite loop. Once we have a connection, we'll create a task to read data from that client until they disconnect. This will look very similar to what we built in chapter 3, with the main difference being that here we use our own event loop instead of asyncio's.

Listing 14.13 Implementing a server

```
import socket

from chapter_14.listing_14_11 import CustomTask
from chapter_14.listing_14_12 import EventLoop


async def read_from_client(conn, loop: EventLoop):
    print(f'Reading data from client {conn}')
    try:
        while data := await loop.sock_recv(conn):
            print(f'Got {data} from client!')
    finally:
        loop.sock_close(conn)


async def listen_for_connections(sock, loop: EventLoop):
    while True:
```

```
        print('Waiting for connection...')
        conn, addr = await loop.sock_accept(sock)
        CustomTask(read_from_client(conn, loop), loop)
        print(f'I got a new connection from {sock}!')


async def main(loop: EventLoop):
    server_socket = socket.socket()
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_RE

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

    await listen_for_connections(server_socket, loop)


event_loop = EventLoop()
event_loop.run(main(event_loop))
```

1. Read data from the client, and log it.

2. Listen for client connections, creating a task to read data when a client connects.

3. Create an event loop instance, and run the main coroutines.

In the preceding listing, we first define a coroutine function to read data from a client in a loop, printing the results as we get them. We also define a coroutine function to listen for client connections from a server socket in an infinite loop, creating a `CustomTask` to concurrently listen for data from that client. In our main coroutine, we create a server socket and call our `listen_for_connections` coroutine function. Then, we create an instance of our event loop implementation, passing in the `main` coroutine to the `run` method.

Running this code, you should be able to connect with multiple clients concurrently over Telnet and send messages to the server. For example, two clients connecting and sending a few test messages may look something like the following:

```
Waiting for connection...
Registering socket to accept connections...
Selector has an event, processing...
I got a new connection from <socket.socket fd=4, family=Addr
Waiting for connection...
Registering socket to accept connections...
Reading data from client <socket.socket fd=7, family=Address
Registering socket to listen for data...
Selector has an event, processing...
Got b'test from client one!\r\n' from client!
Registering socket to listen for data...
Selector has an event, processing...
I got a new connection from <socket.socket fd=4, family=Addr
Waiting for connection...
Registering socket to accept connections...
Reading data from client <socket.socket fd=8, family=Address
Registering socket to listen for data...
Selector has an event, processing...
Got b'test from client two!\r\n' from client!
Registering socket to listen for data...
```

In the above output, one client connects, triggering the selector to resume `listen_for_connections` from its suspension point on `loop.sock_accept`. This also registers the client connection with the selector when we create a task for `read_from_client`. The first client sends the message `"test from client one!"`, which again triggers the selector to fire any callbacks. In this case we advance the `read_from_client` task, outputting our client's message to the console. Then, a second client connects, and the same process happens again.

While this isn't a production-worthy event loop by any stretch of the imagination (we don't really handle exceptions properly, and we only allow socket events to trigger event loop iteration, among other shortcomings), this should give you an idea as to how the inner workings of the event loop and asynchronous programming in Python work. An exercise would be to take the concepts here and build a production-ready event loop. Perhaps you can create the next-generation asynchronous Python framework.

ary

- We can check if a callable argument is a coroutine to create APIs that handle both coroutines and regular functions.
- Use context locals when you have state that you need to pass between coroutines, but you want this state independent from your parameters.
- asyncio's `sleep` coroutine can be used to force an iteration of the event loop. This is useful when we need to trigger the event loop to do something but don't have a natural `await` point.

- asyncio is merely Python's standard implementation of an event loop. Other implementations exist, such as uvloop, and we can change them as we wish and still use `async` and `await` syntax. We can also create our own event loop if we'd like to design something with different characteristics to better suit our needs.

index

A

C

D

R

inside back cover

Continued from inside front cover

I want to . . .	How?	Chapter(s)
Offload work to a queue for later processing	Put the work on an asyncio queue	12
Build a producer–consumer workflow	Put items into a queue and process the queue	12
Build a concurrent web crawler	Use queues with a producer-consumer workflow	12
Run existing command-line programs concurrently	Use the asyncio subprocesses API	13
Build APIs that can handle coroutines and functions	Use core asyncio APIs	14
Share state across multiple tasks	Use context variables	14

I want to . . .	How?	Chapter(s)
Use a different event loop	Install a different event loop with asyncio API functions	14
Learn the inner workings of how the asyncio event loop works	Build your own event loop to learn the concepts	14