

```

        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_REUSEADDR, 1)

    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))

```

- ❶ Read data from the client, and log it.
- ❷ Listen for client connections, creating a task to read data when a client connects.
- ❸ 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

AbstractEventLoop class [333](#)

AbstractServer object [209](#)

accept\_connection function [341](#)

accept method [57](#)

acquire method [170](#), [273](#)

add\_done\_callback method [342](#)

add method [329](#)

add\_one function [25](#)

add\_signal\_handler method [69](#)

\_add\_user function [213](#)

after method [183](#)

aiohttp



making web request with [79](#) – [80](#)

overview [76](#)

REST (representational state transfer) API with [218](#) – [228](#)

aiohttp server basics [219](#) – [220](#)

comparing aiohttp with Flask [226](#) – [228](#)

connecting to database and returning results [220](#) – [226](#)

defined [218](#)

setting timeouts with [81](#) – [82](#)

aio-lib project [76](#)

aiomysql library [103](#)

ALL\_COMPLETED option [93](#)

APIs

blocking [44](#) – [45](#)

REST (representational state transfer) API [218](#) – [228](#)

aiohttp server basics [219](#) – [220](#)

comparing aiohttp with Flask [226](#) – [228](#)

connecting to database and returning results [220](#) – [226](#)

defined [218](#)

endpoint with Starlette [230](#) – [231](#)

with coroutines and functions [328](#) – [330](#)

Application class [221](#)

apply\_async method [132](#)

apply method [132](#)

Array object [151](#)

arrays [146](#)

as\_completed function [90](#) – [91](#)

ASGI (asynchronous server gateway interface) [228](#) – [230](#)

with Starlette [230](#) – [234](#)

REST endpoint [230](#) – [231](#)

WebSockets [231](#) – [234](#)

WSGI (web server gateway interface) compared to [228](#) – [230](#)

async

code in synchronous views [242](#)

keyword, creating coroutines with [24](#) – [26](#)

async for loops [126](#)

asynchronous context managers [77](#) – [82](#), [115](#)

making web request with aiohttp [79](#) – [80](#)

setting timeouts with aiohttp [81](#) – [82](#)

asynchronous generators [122](#) – [127](#)

overview [123](#) – [124](#)

with streaming cursor [124](#) – [127](#)

asynchronous queues [290](#) – [311](#)

in web applications [297](#) – [300](#)

LIFO (last in, first out) queues [309](#) – [311](#)

priority queues [303](#) – [309](#)

web crawler queues [300](#) – [303](#)

asynchronous results [132](#) – [133](#)

asynchronous view [240](#) – [242](#)

asyncio [1](#) – [49](#)

advanced techniques [327](#) – [348](#)

APIs with coroutines and functions [328](#) – [330](#)

context variables [330](#) – [331](#)

creating custom event loop [334](#) – [347](#)

forcing event loop iteration [331](#) – [333](#)

using different event loop implementations [333](#) – [334](#)

application [50](#) – [74](#)

blocking sockets [51](#) – [53](#)

connecting to server with Telnet [53](#) – [57](#)

echo server on asyncio event loop [64](#) – [69](#)

non-blocking sockets [57](#) – 61

shutting down [69](#) – [73](#)

using selectors module to build socket event loop [61](#) – [64](#)

awaitables [37](#) – [39](#)

futures [37](#) – [38](#)

relationship between futures, tasks, and coroutines [39](#)

building responsive UI with [178](#) – [185](#)

canceling tasks [34](#) – [35](#)

concurrency [4](#) – [5](#)

coroutines [24](#) – [27](#)

creating with async keyword [24](#) – [26](#)

pausing execution with await keyword [26](#) – [27](#)

pitfalls of tasks and [42](#) – [45](#)

debug mode [47](#) – [49](#)

using asyncio.run [47](#)

using command line arguments [47](#)

using environment variables [48](#) – [49](#)

event loop [45](#) – [47](#)

accessing [46](#) – [47](#)

creating manually [46](#)

global interpreter lock (GIL) [12](#) – [17](#)

input/output (I/O)-bound and CPU-bound [3](#) – [4](#)

long-running coroutines with sleep [27](#) – [29](#)

MapReduce using [136](#) – [144](#)

Google Books Ngram dataset [139](#) – [140](#)

mapping and reducing with asyncio [140](#) – [144](#)

simple MapReduce example [137](#) – [139](#)

measuring coroutine execution time with decorators [40](#) – [42](#)

microservices and [246](#)

multitasking [7](#)

cooperative multitasking [7](#) – [8](#)

preemptive multitasking [7](#)

overview [2](#) – [3](#)

parallelism [5](#) – [7](#)

processes [8](#)

process pool executors with [133](#) – [136](#)

defined [133](#) – [134](#)

partial function application [135](#) – [136](#)

with asyncio event loop [134](#) – [135](#)

running concurrently with tasks [30](#) – [33](#)

creating tasks [30](#) – [31](#)

running multiple tasks concurrently [31](#) – [33](#)

setting timeout and canceling with wait\_for [35](#) – [37](#)

single-threaded concurrency [17](#) – [21](#)

threads with [164](#) – [169](#)

default executors [168](#) – [169](#)

overview [8](#) – [11](#)

requests library [164](#) – [165](#)

thread pool executors [165](#) – [167](#)

@asyncio.coroutine decorator [335](#)

asyncio.create\_task function [30](#)

asyncio.get\_event\_loop function [46](#)

asyncio.get\_running\_loop function [46](#)

asyncio.iscoroutine function [328](#)

asyncio.new\_event\_loop method [46](#)

asyncio.queue.QueueEmpty exception [294](#)

asyncio.run [47](#)

asyncio.run\_coroutine\_threadsafe function [179](#)

asyncio.set\_event\_loop function [333](#)

asyncio.shield function [36](#)

asyncio.sleep function [27](#)



async keyword [24](#), [335](#)

asyncpg

executing queries with [107](#) – [109](#)

overview [103](#)

transactions with [118](#) – [122](#)

manually managing transactions [120](#) – [122](#)

nested transactions [119](#) – [120](#)

asyncpg.connect function [104](#)

asyncpg Record object [109](#)

async with block [80](#), [116](#), [276](#)

awaitables [37](#) – [39](#)

custom [337](#) – [339](#)

futures [37](#) – [38](#)

relationship between futures, tasks, and coroutines [39](#)

await keyword [26](#) – [27](#), [335](#)

`__await__` method [39](#), [338](#)

AWS (Amazon Web Servers) [140](#)

B

backend-for-frontend pattern

implementing [253](#) – [258](#)

overview [246](#) – [247](#)

BaseSelector class [61](#)

base services [249](#) – [253](#)

blocking APIs [44](#) – [45](#)

blocking sockets [51](#) – [53](#), [56](#) – [57](#)

blocking work [240](#) – [242](#)

bounded semaphores [278](#) – [280](#)

brew install telnet command [53](#)

bugs, single-threaded concurrency [268](#) – [272](#)

bursty code [278](#)

## C

call\_soon method [329](#)

canceling tasks

overview [34](#) – [35](#)

with wait\_for [35](#) – [37](#)

cancel method [96](#), [162](#)

cbreak mode [202](#)

\_change\_state method [288](#)

channels library [235](#)

chat clients [211](#) – [216](#)

chat servers [211](#) – [216](#)

ChatServerState class [211](#)

checkout\_customer function [293](#)

circuit breaker pattern [261](#) – [265](#)

clear method [281](#)

ClientEchoThread class [163](#)

close method [163](#), [270](#)

code complexity [245](#)

command line

debug mode [47](#)

non-blocking command line input [198](#) – [208](#)

communicating with subprocesses [322](#) – [325](#)

completable futures [38](#)

concurrency

difference between parallelism and [6](#) – [7](#)

overview [4](#) – [5](#)

single-threaded concurrency bugs [268](#) – [272](#)

concurrent.futures library [165](#)

concurrent.futures module [133](#), [179](#)

concurrent web requests [75](#) – [101](#)

aiohttp [76](#)

asynchronous context managers [77](#) – [82](#)

making web request with aiohttp [79](#) – [80](#)

setting timeouts with aiohttp [81](#) – [82](#)

processing requests as they complete [88](#) – [91](#)

running requests concurrently with gather [84](#) – [88](#)

running tasks concurrently [82](#) – [84](#)

wait function [92](#) – [101](#)

handling timeouts [99](#) – [100](#)

processing results as they complete [96](#) – [98](#)

waiting for tasks to complete [92](#) – [94](#)

watching for exceptions [94](#) – [96](#)

wrapping in task [100](#) – [101](#)

conditions [285](#) – [289](#)

CONNECT command [211](#)

connect function [115](#)

Connection class [124](#), [288](#)

connection\_lost method [198](#)

connection pooling [79](#)

connections

multiple [56](#) – [57](#)

pools [109](#) – [117](#)

creating to run queries concurrently [113](#) – [117](#)

inserting random SKUs into product database [110](#) – [113](#)

context switches [7](#)

context variables [330](#) – [331](#)

cooperative multitasking

benefits of [7](#) – [8](#)

defined [7](#)

coroutine\_function function [329](#)

coroutine object [25](#)

coroutines [17](#), [37](#) – [39](#)

creating custom event loop [335](#)

creating with async keyword [24](#) – [26](#)

deprecated, generator-based [335](#) – [337](#)

futures [37](#) – [38](#)

long-running coroutines with sleep [27](#) – [29](#)

measuring execution time with decorators [40](#) – [42](#)

pausing execution with await keyword [26](#) – [27](#)

pitfalls of [42](#) – [45](#)

running blocking APIs [44](#) – [45](#)

running CPU-bound code [42](#) – [44](#)

relationship between futures, tasks, and [39](#)

running concurrently with tasks [30](#) – [33](#)

creating tasks [30](#) – [31](#)

running multiple tasks concurrently [31](#) – [33](#)

counter variable [170](#)

count function [134](#)

CPU-bound code [42](#) – [44](#)

CPU-bound work [128](#) – [158](#)

MapReduce using asyncio [136](#) – [144](#)

Google Books Ngram dataset [139](#) – [140](#)

mapping and reducing with asyncio [140](#) – [144](#)

simple MapReduce example [137](#) – [139](#)

multiprocessing and multiple event loops [154](#) – [158](#)

multiprocessing library [129](#) – [131](#)

overview [3](#) – [4](#)

process pools [131](#) – [133](#)

executors with asyncio [133](#) – [136](#)

using asynchronous results [132](#) – [133](#)



sharing data [145](#) – [154](#)

race conditions and [146](#) – [148](#)

synchronizing with locks [149](#) – [151](#)

with process pools [151](#) – [154](#)

threads for [185](#) – [190](#)

multithreading with hashlib [185](#) – [187](#)

multithreading with NumPy [188](#) – [190](#)

create statement [107](#)

create\_subprocess\_shell function [313](#)

CREATE TABLE statement [108](#)

critical section [149](#)

cursor method [124](#)

cursors [122](#), [201](#)

customer\_generator function [296](#)

CustomFuture class [338](#)

## D

daemon threads [161](#)

database, connecting to [220](#) – [226](#)

database drivers, non-blocking [102](#) – [127](#)

asynchronous generators [122](#) – [127](#)

overview [123](#) – [124](#)

with streaming cursor [124](#) – [127](#)

asyncpg

executing queries with [107](#) – [109](#)

overview [103](#)

connecting to Postgres database [103](#) – [104](#)

database schema [104](#) – [107](#)

executing queries concurrently with connection pools [109](#) – [117](#)

creating connection pool to run queries concurrently [113](#) – [117](#)

inserting random SKUs into product database [110](#) – [113](#)

transactions with asyncpg [118](#) – [122](#)

manually managing transactions [120](#) – [122](#)

nested transactions [119](#) – [120](#)

database schema [104](#) – [107](#)

deadlocks [173](#) – [175](#)

debug mode [47](#) – [49](#)

using asyncio.run [47](#)

using command line arguments [47](#)

using environment variables [48](#) – [49](#)

decorators [40](#) – [42](#)

default executors [168](#) – [169](#)

DefaultSelector class [61](#)

def keyword [24](#)

delay function [28](#)

delay parameter [89](#)

deprecated coroutines [335](#) – [337](#)

*Design Patterns: Elements of Reusable Object-Oriented Software* [123](#)

Django asynchronous views [235](#) – [242](#)

running blocking work in [240](#) – [242](#)

using async code in synchronous views [242](#)

DOMContentLoaded event [234](#)

done method [34](#)

do\_work method [286](#)

drain method [197](#)

E

echo function [161](#)

echo server on asyncio event loop [64](#) – [69](#)

designing asyncio echo server [65](#) – [67](#)

event loop coroutines for sockets [64](#) – [65](#)

handling errors in tasks [67](#) – [69](#)

echo task [66](#)

encrypt task [320](#)

environment variables [48](#) – [49](#)

eof\_recieved method [194](#)

errors, in tasks [67](#) – [69](#)

escape codes [203](#)

Event class [281](#)

EventLoop class [345](#)

event loops [20](#), [45](#) – [47](#)

accessing [46](#) – [47](#)

asyncio [134](#) – [135](#)

creating custom [334](#) – [347](#)

coroutines and generators [335](#)

custom awaitables [337](#) – [339](#)

generator-based coroutines are deprecated [335](#) – [337](#)

implementing event loop [343](#) – 345

implementing server with custom event loop [346](#) – [347](#)

task implementation [342](#) – [343](#)

using sockets with futures [340](#) – [342](#)

creating manually [46](#)

echo server on [64](#) – [69](#)

designing asyncio echo server [65](#) – [67](#)

event loop coroutines for sockets [64](#) – [65](#)

handling errors in tasks [67](#) – [69](#)

forcing event loop iteration [331](#) – [333](#)

in separate threads [175](#) – [185](#)

building responsive UI with asyncio and threads [178](#) – [185](#)

Tkinter [176](#) – [178](#)

multiple [154](#) – [158](#)

selectors module building socket [61](#) – [64](#)

using different event loop implementations [333](#) – [334](#)

Event object [281](#)

except block [36](#), [122](#), [315](#)

exceptions

watching for [94](#) – [96](#)

with gather [86](#) – [88](#)

execute method [288](#)

Executor abstract class [133](#), [165](#)

executor parameter [168](#)

executors, process pool [133](#) – [136](#)

defined [133](#) – [134](#)

partial function application [135](#) – [136](#)

with asyncio event loop [134](#) – [135](#)

F

failed requests [258](#) – [260](#)

fetch\_status function [89](#)

FileServer class [283](#)

FileUpload class [282](#)

FileUpload object [283](#)

find\_and\_replace method [173](#)

fire\_event method [286](#)

FIRST\_COMPLETED option [96](#)

FIRST\_EXCEPTION option [95](#)

Flask [226](#) – [228](#)

for loop [82](#), [108](#), [137](#), [325](#)

functools module [135](#)

functools.reduce function [138](#)

future class [179](#), [340](#)

futures

overview [37](#) – [38](#)



relationship between tasks and coroutines [39](#)

using sockets with [340](#) – [342](#)

## G

gather function [84](#) – [88](#)

generators

asynchronous [122](#) – [127](#)

overview [123](#) – [124](#)

with streaming cursor [124](#) – 127

creating custom event loop [335](#)

deprecated generator-based coroutines [335](#) – [337](#)

get\_inventory function [265](#)

get method [132](#)

get\_products\_with\_inventory helper function [257](#)

get\_response\_item\_count helper method [257](#)

get\_status\_code function [167](#)

GIL (global interpreter lock) [2](#), [12](#) – [17](#)

asyncio and [17](#)

releasing [15](#) – [17](#)

Goldilocks approach [141](#)

Google Books Ngram dataset [139](#) – [140](#)

grid method [183](#)

gunicorn command [239](#)

H

hashlib [185](#) – [187](#)

heapsort algorithm [308](#)

hello world application [176](#)

hello\_world\_message function [29](#)

HTTPGetClientProtocol class [195](#)

I

I/O (input/output)-bound operations [3](#) – [4](#)

id function [239](#)

id parameter [223](#)

import statement [172](#)

initargs parameter [152](#)

init function [152](#)

initializer parameter [152](#)

input function [198](#)

input\_ready event [325](#)

input\_writer function [325](#)

INSERT statement [108](#)

J

join method [10](#), [130](#)

K

KeyboardInterrupt exception [69](#)

kill command [69](#)

kill method [314](#)

## L

libuv library [333](#)

LIFO (last in, first out) queues [309](#) – [311](#)

LIMIT statement [122](#)

listen\_for\_connections function [347](#)

listen\_for\_messages method [331](#)

LoadTester application [184](#)

## locks

synchronization [149](#) – [151](#), [272](#) – [276](#)

with threads [169](#) – [175](#)

deadlocks [173](#) – [175](#)

reentrant locks [171](#) – [173](#)

long-running coroutines [27](#) – [29](#)

loop.set\_default\_executor method [168](#)

ls -la command [316](#)

## M

mainloop method [184](#)

main thread [44](#)

make\_request function [38](#)

make\_request method [196](#)

\_make\_requests method [180](#)

map\_frequencies function [143](#)

map method [136](#)

map operation [138](#)

MapReduce [136](#) – [144](#)

Google Books Ngram dataset [139](#) – [140](#)

mapping and reducing with asyncio [140](#) – [144](#)

simple example [137](#) – [139](#)

match\_info dictionary [223](#)

max\_failure parameter [263](#)

max\_size parameter [115](#)

mean\_for\_row function [189](#)

mean function [188](#)

microservices [244](#) – [266](#)

backend-for-frontend pattern [246](#) – [247](#)

product listing API [248](#) – [265](#)

circuit breaker pattern [261](#) – [265](#)

implementing backend-for-frontend [253](#) – [258](#)

implementing base services [249](#) – [253](#)

retrying failed requests [258](#) – [260](#)

user favorite service [248](#)

reasons for [245](#) – [246](#)

asyncio and [246](#)

complexity of code [245](#)

scalability [246](#)

team and stack independence [246](#)

min\_size parameter [115](#)

monoliths [244](#)

multiprocessing [154](#) – [158](#)

multiprocessing.cpu\_count() function [132](#)

multiprocessing library [129](#) – [131](#)

multitasking [7](#)

cooperative multitasking

benefits of [7](#) – [8](#)

defined [7](#)

preemptive multitasking [7](#)

multithreading

with hashlib [185](#) – [187](#)

with NumPy [188](#) – [190](#)

mutex (mutual exclusion) [149](#)

## N

nested transactions [119](#) – [120](#)

new\_event\_loop method [184](#)

next function [124](#)

Ngram dataset, Google Books [139](#) – [140](#)

non-blocking command line input [198](#) – [208](#)

non-blocking database drivers [102](#) – [127](#)

asynchronous generators [122](#) – [127](#)

overview [123](#) – [124](#)

with streaming cursor [124](#) – [127](#)

asyncpg

executing queries with [107](#) – [109](#)

overview [103](#)

connecting to Postgres database [103](#) – [104](#)



database schema [104](#) – [107](#)

executing queries concurrently with connection pools [109](#) – [117](#)

creating connection pool to run queries concurrently [113](#) – [117](#)

inserting random SKUs into product database [110](#) – [113](#)

transactions with asyncpg [118](#) – [122](#)

manually managing transactions [120](#) – [122](#)

nested transactions [119](#) – [120](#)

non-blocking sockets [57](#) – [61](#)

notify\_all method [286](#)

NumPy [188](#) – [190](#)

numpy function [188](#)

Nygard, Michael [261](#)

O

on\_cleanup handler [222](#)

onmessage callback [234](#)

on\_startup handler [221](#)

open\_connection function [196](#)

Order class [307](#)

output, controlling standard [315](#) – [318](#)

output\_consumer function [325](#)

P

parallelism

difference between concurrency and [6](#) – [7](#)

overview [5](#) – [6](#)

partial function application [135](#) – [136](#)

paused array [63](#)

pausing execution [26](#) – [27](#)

pending task set [99](#)

PEP-333 (Python enhancement proposal) [229](#)

pip command [164](#)

pipes [199](#)

`_poll_queue` method [183](#)

`Pool.apply_async` method [133](#)

Postgres database [103](#) – [104](#)

postgres database [104](#)

preemptive multitasking [7](#)

prefetch parameter [125](#)

priority queues [303](#) – [309](#)

`Process.stdout` field [316](#)

`Process` class [130](#), [318](#)

processes [8](#)

`process_page` function [302](#)

process pool initializers [151](#)

process pools [44](#), [131](#) – [133](#)

executors with `asyncio` [133](#) – [136](#)

partial function application [135](#) – [136](#)

process pool executors defined [133](#) – [134](#)

with asyncio event loop [134](#) – [135](#)

using asynchronous results [132](#) – [133](#)

with sharing data [151](#) – [154](#)

producer – consumer workflows [290](#)

product listing API [248](#) – [265](#)

circuit breaker pattern [261](#) – [265](#)

implementing backend-for-frontend [253](#) – [258](#)

implementing base services [249](#) – [253](#)

retrying failed requests [258](#) – [260](#)

user favorite service [248](#)

protocol factories [195](#)

protocols [192](#) – [196](#)

protocol variable [196](#)

put method [295](#)

PYTHONASYNCIODEBUG variable [48](#)

## Q

queries

executing concurrently with connection pools [109](#) – [117](#)

creating connection pool to run queries concurrently [113](#) – [117](#)

inserting random SKUs into product database [110](#) – [113](#)

executing with asyncpg [107](#) – [109](#)

queue module [181](#)

queues, asynchronous [290](#) – [311](#)

in web applications [297](#) – [300](#)

last in, first out (LIFO) queues [309](#) – [311](#)

priority queues [303](#) – [309](#)

web crawler queues [300](#) – [303](#)

\_queue\_update method [183](#)

## R

race conditions [12](#), [146](#) – [148](#)

read coroutine [202](#) – [208](#)

readers, stream [196](#) – [198](#)

read\_from\_client task [347](#)

recv method [54](#)

reduce function [143](#)

reduce operation [144](#)

reentrant locks [171](#) – [173](#)

reference counting [12](#)

\_register\_socket\_to\_read method [345](#)

regular\_function function [329](#)

*Release It* (Nygard) [261](#)

release method [170](#), [273](#)

releasing GIL (global interpreter lock) [15](#) – [17](#)

REPL (read eval print loop) [8](#)

Request class [221](#)

request method [263](#)

requests endpoint [242](#)

requests library [44](#), [164](#) – [165](#)

resources [218](#)

REST (representational state transfer) API [218](#) – [228](#)

aiohttp server basics [219](#) – [220](#)

comparing aiohttp with Flask [226](#) – [228](#)

connecting to database and returning results [220](#) – [226](#)

defined [218](#)

endpoint with Starlette [230](#) – [231](#)

result method [37](#), [94](#)

results variable [109](#)

return statement [123](#)

return\_when parameter [96](#)

return\_when string [92](#)

RLock class [172](#)

route object [230](#)

RouteTableDef decorator [219](#)

\_run\_all method [329](#)

run\_forever method [184](#)

run\_in\_executor method [168](#)

run\_in\_new\_loop function [157](#)

run method [162](#), [343](#)

S

salt [186](#)

SAVEPOINT command [119](#)

say\_hello function [132](#), [177](#)

scalability [246](#)



script function [185](#)

select module [160](#)

selector module [340](#)

selectors module [61](#) – [64](#)

select statement [119](#)

semaphores [276](#) – [280](#)

sendall method [54](#)

send method [336](#)

servers

chat server and client [211](#) – [216](#)

connecting to with Telnet [53](#) – [57](#)

multiple connections and blocking [56](#) – [57](#)

reading and writing data to and from socket [54](#) – [56](#)

creating [209](#) – [211](#)

implementing with custom event loop [346](#) – [347](#)

sessions [79](#)

setcbreak function [202](#)

set method [281](#)

set\_result method [37](#), [340](#)

shared memory objects [145](#)

sharing data [145](#) – [154](#)

locks and [169](#) – [175](#)

race conditions and [146](#) – [148](#)

synchronizing with locks [149](#) – [151](#)

with process pools [151](#) – [154](#)

shutdown method [162](#)

shutting down [69](#) – [73](#)

listening for signals [69](#) – [70](#)

waiting for pending tasks to finish [70](#) – [73](#)

SIGKILL signal [314](#)

signals [69](#) – [70](#)

signal.signal function [70](#)

SIGTERM signal [314](#)

single-threaded concurrency [17](#) – [21](#), [268](#) – [272](#)

single-threaded event loop [3](#)

SKUs [110](#) – [113](#)

sleep, long-running coroutines with [27](#) – [29](#)

sleep command [314](#)

sleep function [50](#)

sock\_accept function [341](#)

socket.accept method [64](#)

socket function [52](#)

socket list [234](#)

socket method [64](#)

sockets [17](#) – [19](#)

blocking [51](#) – [53](#), [56](#) – [57](#)

event loops

coroutines for [64](#) – [65](#)

selectors module building [61](#) – [64](#)

non-blocking [57](#) – [61](#)

reading and writing data to and from [54](#) – [56](#)

using with futures [340](#) – [342](#)

stack independence [246](#)

Starlette, ASGI with [230](#) – [234](#)

REST endpoint [230](#) – [231](#)

WebSockets [231](#) – [234](#)

Starlette class [230](#)

`_start` method [183](#)

start method [10](#), [121](#), [130](#), [180](#)

stdin field [323](#)

stdout parameter [315](#)

step method [343](#)

StopIteration exception [336](#)

streaming cursor [124](#) – [127](#)

StreamReader method [196](#)

streams [191](#) – [216](#)

creating chat server and client [211](#) – [216](#)

creating servers [209](#) – [211](#)

non-blocking command line input [198](#) – [208](#)

overview [192](#)

stream readers and stream writers [196](#) – [198](#)

transports and protocols [192](#) – [196](#)

StressTest object [183](#)

submit method [167](#)

subprocesses [312](#) – [326](#)

communicating with [322](#) – [325](#)

creating [313](#) – [321](#)

controlling standard output [315](#) – [318](#)

running subprocesses concurrently [318](#) – [321](#)

synchronization [267](#) – [289](#)

conditions [285](#) – [289](#)

limiting concurrency with semaphores [276](#) – [280](#)

locks [149](#) – [151](#), [272](#) – [276](#)

notifying tasks with events [280](#) – [285](#)

single-threaded concurrency bugs [268](#) – [272](#)

synchronous views [242](#)

sync\_to\_async function [240](#)

T

target function [10](#), [163](#)

Task object [93](#)

task.result() method [93](#)

tasks [37](#) – [39](#)

canceling

overview [34](#) – [35](#)

setting timeout and canceling with wait\_for [35](#) – [37](#)

futures [37](#) – [38](#)

handling errors in [67](#) – [69](#)

implementing [342](#) – [343](#)

notifying with events [280](#) – [285](#)

pitfalls of [42](#) – [45](#)

running blocking APIs [44](#) – [45](#)

running CPU-bound code [42](#) – [44](#)

relationship between futures, and coroutines [39](#)

running concurrently [82](#) – [84](#)

running coroutines concurrently with [30](#) – [33](#)

creating tasks [30](#) – [31](#)

running multiple tasks [31](#) – [33](#)

shutting down and [70](#) – [73](#)

waiting for completion [92](#) – [94](#)

wrapping in [100](#) – [101](#)

team independence [246](#)

Telnet [53](#) – [57](#)

multiple connections and blocking [56](#) – [57](#)

reading and writing data to and from socket [54](#) – [56](#)

terminal raw mode [202](#) – [208](#)

terminate method [314](#)

Thread class [160](#)

threading module [160](#) – [163](#)

thread locals [330](#)



thread-per-connection model [160](#)

thread pool executors [45](#)

overview [165](#) – [167](#)

with asyncio [167](#)

threads [8](#) – [11](#), [159](#) – [190](#)

event loops in separate threads [175](#) – [185](#)

building responsive UI with asyncio and threads [178](#) – [185](#)

Tkinter [176](#) – [178](#)

for CPU-bound work [185](#) – [190](#)

multithreading with hashlib [185](#) – [187](#)

multithreading with NumPy [188](#) – [190](#)

locks with [169](#) – [175](#)

deadlocks [173](#) – [175](#)

reentrant locks [171](#) – [173](#)

threading module [160](#) – [163](#)

with asyncio [164](#) – [169](#)

default executors [168](#) – [169](#)

requests library [164](#) – [165](#)

thread pool executors [165](#) – [167](#)

thread\_sensitive url parameter [242](#)

timeout parameter [99](#)

timeouts

setting [35](#) – [37](#)

wait function [99](#) – [100](#)

with aiohttp [81](#) – [82](#)

with as\_completed [90](#) – [91](#)

time slicing [8](#)

Tkapp\_Call function [182](#)

TK class [181](#)

Tkinter [176](#) – [178](#)

TooManyRetries exception [259](#)

transactions with asyncpg [118](#) – [122](#)

manually managing [120](#) – [122](#)

nested transactions [119](#) – [120](#)

transports [192](#) – [196](#)

try/catch statement [68](#)

try/finally block [77](#)

try block [36](#)

try catch block [88](#), [163](#)

tty module [202](#)

U

UDP (user datagram protocol) [52](#)

UI, building with asyncio [178](#) – [185](#)

`_update_bar` method [183](#)

`user_cart` table [252](#)

user favorite service [248](#)

user\_names\_to\_sockets dictionary [272](#)

\_username\_to\_writer dictionary [214](#)

util module [42](#)

## V

Value object [151](#)

values [146](#)

## W

wait\_for function [35](#) – [37](#)

wait\_for method [286](#)

wait\_for statement [35](#)

wait function [92](#) – [101](#)

handling timeouts [99](#) – [100](#)

processing results as they complete [96](#) – [98](#)

waiting for tasks to complete [92](#) – [94](#)

watching for exceptions [94](#) – [96](#)

wrapping in task [100](#) – [101](#)

web applications [217](#) – [243](#)

ASGI (asynchronous server gateway interface) [228](#) – [230](#)

REST endpoint with Starlette [230](#) – [231](#)

WebSockets with Starlette [231](#) – [234](#)

WSGI (web server gateway interface) compared to [228](#) – [230](#)

Django asynchronous views [235](#) – [242](#)

running blocking work in asynchronous view [240](#) – [242](#)

using async code in synchronous views [242](#)

queues in [297](#) – [300](#)

REST (representational state transfer) API with Aiohttp [218](#) – [228](#)

aiohttp server basics [219](#) – [220](#)

comparing aiohttp with Flask [226](#) – [228](#)

connecting to database and returning results [220](#) – [226](#)

REST defined [218](#)

web crawler queues [300](#) – [303](#)

web module [219](#)

web requests, concurrent [75](#) – [101](#)

aiohttp [76](#)

asynchronous context managers [77](#) – [82](#)

making web request with aiohttp [79](#) – [80](#)

setting timeouts with aiohttp [81](#) – [82](#)

processing requests as they complete [88](#) – [91](#)

running requests concurrently with gather [84](#) – [88](#)

running tasks concurrently [82](#) – [84](#)

wait function [92](#) – [101](#)

handling timeouts [99](#) – [100](#)

processing results as they complete [96](#) – [98](#)

waiting for tasks to complete [92](#) – [94](#)

watching for exceptions [94](#) – [96](#)

wrapping in task [100](#) – [101](#)

WebSockets [231](#) – [234](#)

while loop [34](#), [61](#), [162](#), [205](#)

with block [80](#), [151](#), [172](#)

WorkItem class [302](#)

wrapped [41](#)

wrapping in tasks [100](#) – [101](#)

write method [197](#)

writers, stream [196](#) – [198](#)

WSGI (web server gateway interface) [228](#) – [230](#)

# 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