Item 59: Consider ThreadPoolExecutor When Threads Are Necessary for Concurrency

Python includes the concurrent.futures built-in module, which provides the ThreadPoolExecutor class. It combines the best of the Thread (see Item 57: "Avoid Creating New Thread Instances for On-demand Fan-out") and Queue (see Item 58: "Understand How Using Queue for Concurrency Requires Refactoring") approaches to solving the parallel I/O problem from the Game of Life example (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for background and the implementations of various functions and classes below):

```
ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    ...

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)
```

Instead of starting a new Thread instance for each Grid square, I can *fan out* by submitting a function to an executor that will be run in a separate thread. Later, I can wait for the result of all tasks in order to fan in:

```
from concurrent.futures import ThreadPoolExecutor

def simulate_pool(pool, grid):
    next_grid = LockingGrid(grid.height, grid.width)
```

```
    futures = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            future = pool.submit(step_cell, *args)  # Fan out
            futures.append(future)

    for future in futures:
        future.result()                             # Fan in

    return next_grid
```

The threads used for the executor can be allocated in advance, which means I don't have to pay the startup cost on each execution of `simulate_pool`. I can also specify the maximum number of threads to use for the pool—using the `max_workers` parameter—to prevent the memory blow-up issues associated with the naive `Thread` solution to the parallel I/O problem:

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
with ThreadPoolExecutor(max_workers=10) as pool:
    for i in range(5):
        columns.append(str(grid))
        grid = simulate_pool(pool, grid)

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

The best part about the ThreadPoolExecutor class is that it automatically propagates exceptions back to the caller when the result method is called on the Future instance returned by the submit method:

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O')
    ...

with ThreadPoolExecutor(max_workers=10) as pool:
    task = pool.submit(game_logic, ALIVE, 3)
    task.result()
```

```
>>>
Traceback ...
OSError: Problem with I/O
```

If I needed to provide I/O parallelism for the count_neighbors function in addition to game_logic, no modifications to the program would be required since ThreadPoolExecutor already runs these functions concurrently as part of step_cell. It's even possible to achieve CPU parallelism by using the same interface if necessary (see Item 64: "Consider concurrent.futures for True Parallelism").

However, the big problem that remains is the limited amount of I/O parallelism that ThreadPoolExecutor provides. Even if I use a max_workers parameter of 100, this solution still won't scale if I need 10,000+ cells in the grid that require simultaneous I/O. ThreadPoolExecutor is a good choice for situations where there is no asynchronous solution (e.g., file I/O), but there are better ways to maximize I/O parallelism in many cases (see Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- ThreadPoolExecutor enables simple I/O parallelism with limited refactoring, easily avoiding the cost of thread startup each time fan-out concurrency is required.
- Although ThreadPoolExecutor eliminates the potential memory blow-up issues of using threads directly, it also limits I/O parallelism by requiring max_workers to be specified upfront.

Item 60: Achieve Highly Concurrent I/O with Coroutines

The previous items have tried to solve the parallel I/O problem for the Game of Life example with varying degrees of success. (See Item 56: "Know How to Recognize When Concurrency Is Necessary" for

background and the implementations of various functions and classes below.) All of the other approaches fall short in their ability to handle thousands of simultaneously concurrent functions (see Item 57: "Avoid Creating New `Thread` Instances for On-demand Fan-out," Item 58: "Understand How Using `Queue` for Concurrency Requires Refactoring," and Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency").

Python addresses the need for highly concurrent I/O with *coroutines*. Coroutines let you have a very large number of seemingly simultaneous functions in your Python programs. They're implemented using the `async` and `await` keywords along with the same infrastructure that powers generators (see Item 30: "Consider Generators Instead of Returning Lists," Item 34: "Avoid Injecting Data into Generators with `send`," and Item 35: "Avoid Causing State Transitions in Generators with `throw`").

The cost of starting a coroutine is a function call. Once a coroutine is active, it uses less than 1 KB of memory until it's exhausted. Like threads, coroutines are independent functions that can consume inputs from their environment and produce resulting outputs. The difference is that coroutines pause at each `await` expression and resume executing an `async` function after the pending *awaitable* is resolved (similar to how `yield` behaves in generators).

Many separate `async` functions advanced in lockstep all seem to run simultaneously, mimicking the concurrent behavior of Python threads. However, coroutines do this without the memory overhead, startup and context switching costs, or complex locking and synchronization code that's required for threads. The magical mechanism powering coroutines is the *event loop*, which can do highly concurrent I/O efficiently, while rapidly interleaving execution between appropriately written functions.

I can use coroutines to implement the Game of Life. My goal is to allow for I/O to occur within the `game_logic` function while overcoming the problems from the `Thread` and `Queue` approaches in the previous items. To do this, first I indicate that the `game_logic` function is a coroutine by defining it using `async def` instead of `def`. This will allow me to use the `await` syntax for I/O, such as an asynchronous `read` from a socket:

```
ALIVE = '*'
EMPTY = '-'
```

```
class Grid:
    ...

def count_neighbors(y, x, get):
    ...

async def game_logic(state, neighbors):
    ...
    # Do some input/output in here:
    data = await my_socket.read(50)
    ...
```

Similarly, I can turn `step_cell` into a coroutine by adding `async` to its definition and using `await` for the call to the `game_logic` function:

```
async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)
```

The `simulate` function also needs to become a coroutine:

```
import asyncio

async def simulate(grid):
    next_grid = Grid(grid.height, grid.width)

    tasks = []
    for y in range(grid.height):
        for x in range(grid.width):
            task = step_cell(
                y, x, grid.get, next_grid.set)      # Fan out
            tasks.append(task)

    await asyncio.gather(*tasks)                    # Fan in

    return next_grid
```

The coroutine version of the `simulate` function requires some explanation:

- Calling `step_cell` doesn't immediately run that function. Instead, it returns a `coroutine` instance that can be used with an `await` expression at a later time. This is similar to how generator functions that use `yield` return a generator instance when they're called instead of executing immediately. Deferring execution like this is the mechanism that causes *fan-out*.

- The `gather` function from the `asyncio` built-in library causes *fan-in*. The `await` expression on `gather` instructs the event loop to run the `step_cell` coroutines concurrently and resume execution of the `simulate` coroutine when all of them have been completed.
- No locks are required for the `Grid` instance since all execution occurs within a single thread. The I/O becomes parallelized as part of the event loop that's provided by `asyncio`.

Finally, I can drive this code with a one-line change to the original example. This relies on the `asyncio.run` function to execute the `simulate` coroutine in an event loop and carry out its dependent I/O:

```
class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))   # Run the event loop

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

The result is the same as before. All of the overhead associated with threads has been eliminated. Whereas the `Queue` and `ThreadPoolExecutor` approaches are limited in their exception handling—merely re-raising exceptions across thread boundaries—with coroutines I can actually use the interactive debugger to step through the code line by line (see Item 80: "Consider Interactive Debugging with `pdb`"):

```
async def game_logic(state, neighbors):
    ...
```

```
    raise OSError('Problem with I/O')
    ...

asyncio.run(game_logic(ALIVE, 3))
```

```
>>>
Traceback ...
OSError: Problem with I/O
```

Later, if my requirements change and I also need to do I/O from within `count_neighbors`, I can easily accomplish this by adding `async` and `await` keywords to the existing functions and call sites instead of having to restructure everything as I would have had to do if I were using `Thread` or `Queue` instances (see Item 61: "Know How to Port Threaded I/O to `asyncio`" for another example):

```
async def count_neighbors(y, x, get):
    ...

async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = await count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

The beauty of coroutines is that they decouple your code's instructions for the external environment (i.e., I/O) from the implementation that carries out your wishes (i.e., the event loop). They let you focus on the logic of what you're trying to do instead of wasting time trying to figure out how you're going to accomplish your goals concurrently.

Things to Remember

- Functions that are defined using the `async` keyword are called coroutines. A caller can receive the result of a dependent coroutine by using the `await` keyword.
- Coroutines provide an efficient way to run tens of thousands of functions seemingly at the same time.
- Coroutines can use fan-out and fan-in in order to parallelize I/O, while also overcoming all of the problems associated with doing I/O in threads.

Item 61: Know How to Port Threaded I/O to asyncio

Once you understand the advantage of coroutines (see Item 60: "Achieve Highly Concurrent I/O with Coroutines"), it may seem daunting to port an existing codebase to use them. Luckily, Python's support for asynchronous execution is well integrated into the language. This makes it straightforward to move code that does threaded, blocking I/O over to coroutines and asynchronous I/O.

For example, say that I have a TCP-based server for playing a game involving guessing a number. The server takes `lower` and `upper` parameters that determine the range of numbers to consider. Then, the server returns guesses for integer values in that range as they are requested by the client. Finally, the server collects reports from the client on whether each of those numbers was closer (warmer) or further away (colder) from the client's secret number.

The most common way to build this type of client/server system is by using blocking I/O and threads (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"). To do this, I need a helper class that can manage sending and receiving of messages. For my purposes, each line sent or received represents a command to be processed:

```
class EOFError(Exception):
    pass

class ConnectionBase:
    def __init__(self, connection):
```

```
        self.connection = connection
        self.file = connection.makefile('rb')

    def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.connection.send(data)

    def receive(self):
        line = self.file.readline()
        if not line:
            raise EOFError('Connection closed')
        return line[:-1].decode()
```

The server is implemented as a class that handles one connection at a time and maintains the client's session state:

```
import random

WARMER = 'Warmer'
COLDER = 'Colder'
UNSURE = 'Unsure'
CORRECT = 'Correct'

class UnknownCommandError(Exception):
    pass

class Session(ConnectionBase):
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state(None, None)

    def _clear_state(self, lower, upper):
        self.lower = lower
        self.upper = upper
        self.secret = None
        self.guesses = []
```

It has one primary method that handles incoming commands from the client and dispatches them to methods as needed. Note that here I'm using an assignment expression (introduced in Python 3.8; see Item 10: "Prevent Repetition with Assignment Expressions") to keep the code short:

```
    def loop(self):
        while command := self.receive():
```

```
        parts = command.split(' ')
        if parts[0] == 'PARAMS':
            self.set_params(parts)
        elif parts[0] == 'NUMBER':
            self.send_number()
        elif parts[0] == 'REPORT':
            self.receive_report(parts)
        else:
            raise UnknownCommandError(command)
```

The first command sets the lower and upper bounds for the numbers that the server is trying to guess:

```
def set_params(self, parts):
    assert len(parts) == 3
    lower = int(parts[1])
    upper = int(parts[2])
    self._clear_state(lower, upper)
```

The second command makes a new guess based on the previous state that's stored in the client's `Session` instance. Specifically, this code ensures that the server will never try to guess the same number more than once per parameter assignment:

```
def next_guess(self):
    if self.secret is not None:
        return self.secret

    while True:
        guess = random.randint(self.lower, self.upper)
        if guess not in self.guesses:
            return guess

def send_number(self):
    guess = self.next_guess()
    self.guesses.append(guess)
    self.send(format(guess))
```

The third command receives the decision from the client of whether the guess was warmer or colder, and it updates the `Session` state accordingly:

```
def receive_report(self, parts):
    assert len(parts) == 2
    decision = parts[1]

    last = self.guesses[-1]
```

```
        if decision == CORRECT:
            self.secret = last

        print(f'Server: {last} is {decision}')
```

The client is also implemented using a stateful class:

```
import contextlib
import math

class Client(ConnectionBase):
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state()

    def _clear_state(self):
        self.secret = None
        self.last_distance = None
```

The parameters of each guessing game are set using a `with` statement to ensure that state is correctly managed on the server side (see Item 66: "Consider `contextlib` and `with` Statements for Reusable `try`/`finally` Behavior" for background and Item 63: "Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness" for another example). This method sends the first command to the server:

```
    @contextlib.contextmanager
    def session(self, lower, upper, secret):
        print(f'Guess a number between {lower} and {upper}!'
              f' Shhhhh, it\'s {secret}.')
        self.secret = secret
        self.send(f'PARAMS {lower} {upper}')
        try:
            yield
        finally:
            self._clear_state()
            self.send('PARAMS 0 -1')
```

New guesses are requested from the server, using another method that implements the second command:

```
    def request_numbers(self, count):
        for _ in range(count):
            self.send('NUMBER')
            data = self.receive()
            yield int(data)
            if self.last_distance == 0:
                return
```

Whether each guess from the server was warmer or colder than the last is reported using the third command in the final method:

```
    def report_outcome(self, number):
        new_distance = math.fabs(number - self.secret)
        decision = UNSURE

        if new_distance == 0:
            decision = CORRECT
        elif self.last_distance is None:
            pass
        elif new_distance < self.last_distance:
            decision = WARMER
        elif new_distance > self.last_distance:
            decision = COLDER

        self.last_distance = new_distance

        self.send(f'REPORT {decision}')
        return decision
```

I can run the server by having one thread listen on a socket and spawn additional threads to handle the new connections:

```
import socket
from threading import Thread

def handle_connection(connection):
    with connection:
        session = Session(connection)
        try:
            session.loop()
        except EOFError:
            pass

def run_server(address):
    with socket.socket() as listener:
        listener.bind(address)
        listener.listen()
        while True:
            connection, _ = listener.accept()
            thread = Thread(target=handle_connection,
                            args=(connection,),
                            daemon=True)
            thread.start()
```

The client runs in the main thread and returns the results of the guessing game to the caller. This code explicitly exercises a variety of Python language features (for loops, with statements, generators, comprehensions) so that below I can show what it takes to port these over to using coroutines:

```
def run_client(address):
    with socket.create_connection(address) as connection:
        client = Client(connection)

        with client.session(1, 5, 3):
            results = [(x, client.report_outcome(x))
                       for x in client.request_numbers(5)]

        with client.session(10, 15, 12):
            for number in client.request_numbers(5):
                outcome = client.report_outcome(number)
                results.append((number, outcome))

    return results
```

Finally, I can glue all of this together and confirm that it works as expected:

```
def main():
    address = ('127.0.0.1', 1234)
    server_thread = Thread(
        target=run_server, args=(address,), daemon=True)
    server_thread.start()

    results = run_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')

main()
```

```
>>>
Guess a number between 1 and 5! Shhhhh, it's 3.
Server: 4 is Unsure
Server: 1 is Colder
Server: 5 is Unsure
Server: 3 is Correct
Guess a number between 10 and 15! Shhhhh, it's 12.
Server: 11 is Unsure
Server: 10 is Colder
Server: 12 is Correct
```

```
Client: 4 is Unsure
Client: 1 is Colder
Client: 5 is Unsure
Client: 3 is Correct
Client: 11 is Unsure
Client: 10 is Colder
Client: 12 is Correct
```

How much effort is needed to convert this example to using `async`, `await`, and the `asyncio` built-in module?

First, I need to update my `ConnectionBase` class to provide coroutines for `send` and `receive` instead of blocking I/O methods. I've marked each line that's changed with a `# Changed` comment to make it clear what the delta is between this new example and the code above:

```
class AsyncConnectionBase:
    def __init__(self, reader, writer):              # Changed
        self.reader = reader                         # Changed
        self.writer = writer                         # Changed

    async def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.writer.write(data)                      # Changed
        await self.writer.drain()                    # Changed

    async def receive(self):
        line = await self.reader.readline()          # Changed
        if not line:
            raise EOFError('Connection closed')
        return line[:-1].decode()
```

I can create another stateful class to represent the session state for a single connection. The only changes here are the class's name and inheriting from `AsyncConnectionBase` instead of `ConnectionBase`:

```
class AsyncSession(AsyncConnectionBase):             # Changed
    def __init__(self, *args):
        ...

    def _clear_values(self, lower, upper):
        ...
```

The primary entry point for the server's command processing loop requires only minimal changes to become a coroutine:

```
    async def loop(self):                            # Changed
```

```
        while command := await self.receive():      # Changed
            parts = command.split(' ')
            if parts[0] == 'PARAMS':
                self.set_params(parts)
            elif parts[0] == 'NUMBER':
                await self.send_number()            # Changed
            elif parts[0] == 'REPORT':
                self.receive_report(parts)
            else:
                raise UnknownCommandError(command)
```

No changes are required for handling the first command:

```
    def set_params(self, parts):
        ...
```

The only change required for the second command is allowing asynchronous I/O to be used when guesses are transmitted to the client:

```
    def next_guess(self):
        ...

    async def send_number(self):                    # Changed
        guess = self.next_guess()
        self.guesses.append(guess)
        await self.send(format(guess))              # Changed
```

No changes are required for processing the third command:

```
    def receive_report(self, parts):
        ...
```

Similarly, the client class needs to be reimplemented to inherit from `AsyncConnectionBase`:

```
class AsyncClient(AsyncConnectionBase):             # Changed
    def __init__(self, *args):
        ...

    def _clear_state(self):
        ...
```

The first command method for the client requires a few `async` and `await` keywords to be added. It also needs to use the `asynccontextmanager` helper function from the `contextlib` built-in module:

```
    @contextlib.asynccontextmanager                 # Changed
    async def session(self, lower, upper, secret):  # Changed
        print(f'Guess a number between {lower} and {upper}!'
              f' Shhhhh, it\'s {secret}.')
```

```
        self.secret = secret
        await self.send(f'PARAMS {lower} {upper}')  # Changed
        try:
            yield
        finally:
            self._clear_state()
            await self.send('PARAMS 0 -1')          # Changed
```

The second command again only requires the addition of `async` and `await` anywhere coroutine behavior is required:

```
    async def request_numbers(self, count):         # Changed
        for _ in range(count):
            await self.send('NUMBER')               # Changed
            data = await self.receive()             # Changed
            yield int(data)
            if self.last_distance == 0:
                return
```

The third command only requires adding one `async` and one `await` keyword:

```
    async def report_outcome(self, number):         # Changed
        ...
        await self.send(f'REPORT {decision}')       # Changed
        ...
```

The code that runs the server needs to be completely reimplemented to use the `asyncio` built-in module and its `start_server` function:

```
import asyncio

async def handle_async_connection(reader, writer):
    session = AsyncSession(reader, writer)
    try:
        await session.loop()
    except EOFError:
        pass

async def run_async_server(address):
    server = await asyncio.start_server(
        handle_async_connection, *address)
    async with server:
        await server.serve_forever()
```

The `run_client` function that initiates the game requires changes on nearly every line. Any code that previously interacted with the blocking `socket` instances has to be replaced with `asyncio` versions of

similar functionality (which are marked with `# New` below). All other lines in the function that require interaction with coroutines need to use `async` and `await` keywords as appropriate. If you forget to add one of these keywords in a necessary place, an exception will be raised at runtime.

```
async def run_async_client(address):
    streams = await asyncio.open_connection(*address)    # New
    client = AsyncClient(*streams)                       # New

    async with client.session(1, 5, 3):
        results = [(x, await client.report_outcome(x))
                   async for x in client.request_numbers(5)]

    async with client.session(10, 15, 12):
        async for number in client.request_numbers(5):
            outcome = await client.report_outcome(number)
            results.append((number, outcome))

    _, writer = streams                                  # New
    writer.close()                                       # New
    await writer.wait_closed()                           # New

    return results
```

What's most interesting about `run_async_client` is that I didn't have to restructure any of the substantive parts of interacting with the `AsyncClient` in order to port this function over to use coroutines. Each of the language features that I needed has a corresponding asynchronous version, which made the migration easy to do.

This won't always be the case, though. There are currently no asynchronous versions of the `next` and `iter` built-in functions (see Item 31: "Be Defensive When Iterating Over Arguments" for background); you have to `await` on the `__anext__` and `__aiter__` methods directly. There's also no asynchronous version of `yield from` (see Item 33: "Compose Multiple Generators with `yield from`"), which makes it noisier to compose generators. But given the rapid pace at which `async` functionality is being added to Python, it's only a matter of time before these features become available.

Finally, the glue needs to be updated to run this new asynchronous example end-to-end. I use the `asyncio.create_task` function to enqueue the server for execution on the event loop so that it runs in parallel with the client when the `await` expression is reached. This is

another approach to causing fan-out with different behavior than the asyncio.gather function:

```
async def main_async():
    address = ('127.0.0.1', 4321)

    server = run_async_server(address)
    asyncio.create_task(server)

    results = await run_async_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')

asyncio.run(main_async())
```

```
>>>
Guess a number between 1 and 5! Shhhhh, it's 3.
Server: 5 is Unsure
Server: 4 is Warmer
Server: 2 is Unsure
Server: 1 is Colder
Server: 3 is Correct
Guess a number between 10 and 15! Shhhhh, it's 12.
Server: 14 is Unsure
Server: 10 is Unsure
Server: 15 is Colder
Server: 12 is Correct
Client: 5 is Unsure
Client: 4 is Warmer
Client: 2 is Unsure
Client: 1 is Colder
Client: 3 is Correct
Client: 14 is Unsure
Client: 10 is Unsure
Client: 15 is Colder
Client: 12 is Correct
```

This works as expected. The coroutine version is easier to follow because all of the interactions with threads have been removed. The asyncio built-in module also provides many helper functions and shortens the amount of socket boilerplate required to write a server like this.

Your use case may be more complex and harder to port for a variety of reasons. The asyncio module has a vast number of I/O, synchronization, and task management features that could make adopting

coroutines easier for you (see Item 62: "Mix Threads and Coroutines to Ease the Transition to `asyncio`" and Item 63: "Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness"). Be sure to check out the online documentation for the library (https://docs.python.org/3/library/asyncio.html) to understand its full potential.

Things to Remember

- Python provides asynchronous versions of `for` loops, `with` statements, generators, comprehensions, and library helper functions that can be used as drop-in replacements in coroutines.
- The `asyncio` built-in module makes it straightforward to port existing code that uses threads and blocking I/O over to coroutines and asynchronous I/O.

Item 62: Mix Threads and Coroutines to Ease the Transition to `asyncio`

In the previous item (see Item 61: "Know How to Port Threaded I/O to `asyncio`"), I ported a TCP server that does blocking I/O with threads over to use `asyncio` with coroutines. The transition was big-bang: I moved all of the code to the new style in one go. But it's rarely feasible to port a large program this way. Instead, you usually need to incrementally migrate your codebase while also updating your tests as needed and verifying that everything works at each step along the way.

In order to do that, your codebase needs to be able to use threads for blocking I/O (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism") and coroutines for asynchronous I/O (see Item 60: "Achieve Highly Concurrent I/O with Coroutines") at the same time in a way that's mutually compatible. Practically, this means that you need threads to be able to run coroutines, and you need coroutines to be able to start and wait on threads. Luckily, `asyncio` includes built-in facilities for making this type of interoperability straightforward.

For example, say that I'm writing a program that merges log files into one output stream to aid with debugging. Given a file handle for an input log, I need a way to detect whether new data is available and return the next line of input. I can do this using the `tell` method of the file handle to check whether the current read position matches the length of the file. When no new data is present, an exception should be raised (see Item 20: "Prefer Raising Exceptions to Returning `None`" for background):

```
class NoNewData(Exception):
    pass
```

```
def readline(handle):
    offset = handle.tell()
    handle.seek(0, 2)
    length = handle.tell()

    if length == offset:
        raise NoNewData

    handle.seek(offset, 0)
    return handle.readline()
```

By wrapping this function in a `while` loop, I can turn it into a worker thread. When a new line is available, I call a given callback function to write it to the output log (see Item 38: "Accept Functions Instead of Classes for Simple Interfaces" for why to use a function interface for this instead of a class). When no data is available, the thread sleeps to reduce the amount of busy waiting caused by polling for new data. When the input file handle is closed, the worker thread exits:

```
import time

def tail_file(handle, interval, write_func):
    while not handle.closed:
        try:
            line = readline(handle)
        except NoNewData:
            time.sleep(interval)
        else:
            write_func(line)
```

Now, I can start one worker thread per input file and unify their output into a single output file. The `write` helper function below needs to use a `Lock` instance (see Item 54: "Use Lock to Prevent Data Races in Threads") in order to serialize writes to the output stream and make sure that there are no intra-line conflicts:

```
from threading import Lock, Thread

def run_threads(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        lock = Lock()
        def write(data):
            with lock:
                output.write(data)
```

```
    threads = []
    for handle in handles:
        args = (handle, interval, write)
        thread = Thread(target=tail_file, args=args)
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()
```

As long as an input file handle is still alive, its corresponding worker thread will also stay alive. That means it's sufficient to wait for the `join` method from each thread to complete in order to know that the whole process is done.

Given a set of input paths and an output path, I can call `run_threads` and confirm that it works as expected. How the input file handles are created or separately closed isn't important in order to demonstrate this code's behavior, nor is the output verification function—defined in `confirm_merge` that follows—which is why I've left them out here:

```
def confirm_merge(input_paths, output_path):
    ...

input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)
```

With this threaded implementation as the starting point, how can I incrementally convert this code to use `asyncio` and coroutines instead? There are two approaches: top-down and bottom-up.

Top-down means starting at the highest parts of a codebase, like in the `main` entry points, and working down to the individual functions and classes that are the leaves of the call hierarchy. This approach can be useful when you maintain a lot of common modules that you use across many different programs. By porting the entry points first, you can wait to port the common modules until you're already using coroutines everywhere else.

The concrete steps are:

1. Change a top function to use `async def` instead of `def`.
2. Wrap all of its calls that do I/O—potentially blocking the event loop—to use `asyncio.run_in_executor` instead.

3. Ensure that the resources or callbacks used by run_in_executor invocations are properly synchronized (i.e., using Lock or the asyncio.run_coroutine_threadsafe function).
4. Try to eliminate get_event_loop and run_in_executor calls by moving downward through the call hierarchy and converting intermediate functions and methods to coroutines (following the first three steps).

Here, I apply steps 1–3 to the run_threads function:

```
import asyncio

async def run_tasks_mixed(handles, interval, output_path):
    loop = asyncio.get_event_loop()

    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        def write(data):
            coro = write_async(data)
            future = asyncio.run_coroutine_threadsafe(
                coro, loop)
            future.result()

        tasks = []
        for handle in handles:
            task = loop.run_in_executor(
                None, tail_file, handle, interval, write)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

The run_in_executor method instructs the event loop to run a given function—tail_file in this case—using a specific ThreadPoolExecutor (see Item 59: "Consider ThreadPoolExecutor When Threads Are Necessary for Concurrency") or a default executor instance when the first parameter is None. By making multiple calls to run_in_executor without corresponding await expressions, the run_tasks_mixed coroutine fans out to have one concurrent line of execution for each input file. Then, the asyncio.gather function along with an await expression fans in the tail_file threads until they all complete (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for more about fan-out and fan-in).

This code eliminates the need for the `Lock` instance in the `write` helper by using `asyncio.run_coroutine_threadsafe`. This function allows plain old worker threads to call a coroutine—`write_async` in this case—and have it execute in the event loop from the main thread (or from any other thread, if necessary). This effectively synchronizes the threads together and ensures that all writes to the output file are only done by the event loop in the main thread. Once the `asyncio.gather` awaitable is resolved, I can assume that all writes to the output file have also completed, and thus I can `close` the output file handle in the `with` statement without having to worry about race conditions.

I can verify that this code works as expected. I use the `asyncio.run` function to start the coroutine and run the main event loop:

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks_mixed(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Now, I can apply step 4 to the `run_tasks_mixed` function by moving down the call stack. I can redefine the `tail_file` dependent function to be an asynchronous coroutine instead of doing blocking I/O by following steps 1–3:

```
async def tail_async(handle, interval, write_func):
    loop = asyncio.get_event_loop()

    while not handle.closed:
        try:
            line = await loop.run_in_executor(
                None, readline, handle)
        except NoNewData:
            await asyncio.sleep(interval)
        else:
            await write_func(line)
```

This new implementation of `tail_async` allows me to push calls to `get_event_loop` and `run_in_executor` down the stack and out of the `run_tasks_mixed` function entirely. What's left is clean and much easier to follow:

```
async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)
```

```
    tasks = []
    for handle in handles:
        coro = tail_async(handle, interval, write_async)
        task = asyncio.create_task(coro)
        tasks.append(task)

    await asyncio.gather(*tasks)
```

I can verify that `run_tasks` works as expected, too:

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

It's possible to continue this iterative refactoring pattern and convert `readline` into an asynchronous coroutine as well. However, that function requires so many blocking file I/O operations that it doesn't seem worth porting, given how much that would reduce the clarity of the code and hurt performance. In some situations, it makes sense to move everything to `asyncio`, and in others it doesn't.

The bottom-up approach to adopting coroutines has four steps that are similar to the steps of the top-down style, but the process traverses the call hierarchy in the opposite direction: from leaves to entry points.

The concrete steps are:

1. Create a new asynchronous coroutine version of each leaf function that you're trying to port.
2. Change the existing synchronous functions so they call the coroutine versions and run the event loop instead of implementing any real behavior.
3. Move up a level of the call hierarchy, make another layer of coroutines, and replace existing calls to synchronous functions with calls to the coroutines defined in step 1.
4. Delete synchronous wrappers around coroutines created in step 2 as you stop requiring them to glue the pieces together.

For the example above, I would start with the `tail_file` function since I decided that the `readline` function should keep using blocking I/O. I can rewrite `tail_file` so it merely wraps the `tail_async` coroutine that I defined above. To run that coroutine until it finishes, I need to

create an event loop for each `tail_file` worker thread and then call its `run_until_complete` method. This method will block the current thread and drive the event loop until the `tail_async` coroutine exits, achieving the same behavior as the threaded, blocking I/O version of `tail_file`:

```
def tail_file(handle, interval, write_func):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)

    async def write_async(data):
        write_func(data)

    coro = tail_async(handle, interval, write_async)
    loop.run_until_complete(coro)
```

This new `tail_file` function is a drop-in replacement for the old one. I can verify that everything works as expected by calling `run_threads` again:

```
input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)
```

After wrapping `tail_async` with `tail_file`, the next step is to convert the `run_threads` function to a coroutine. This ends up being the same work as step 4 of the top-down approach above, so at this point, the styles converge.

This is a great start for adopting `asyncio`, but there's even more that you could do to increase the responsiveness of your program (see Item 63: "Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness").

Things to Remember

- The awaitable `run_in_executor` method of the `asyncio` event loop enables coroutines to run synchronous functions in `ThreadPoolExecutor` pools. This facilitates top-down migrations to `asyncio`.
- The `run_until_complete` method of the `asyncio` event loop enables synchronous code to run a coroutine until it finishes. The `asyncio.run_coroutine_threadsafe` function provides the same functionality across thread boundaries. Together these help with bottom-up migrations to `asyncio`.

Item 63: Avoid Blocking the asyncio Event Loop to Maximize Responsiveness

In the previous item I showed how to migrate to asyncio incrementally (see Item 62: "Mix Threads and Coroutines to Ease the Transition to asyncio" for background and the implementation of various functions below). The resulting coroutine properly tails input files and merges them into a single output:

```
import asyncio

async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

However, it still has one big problem: The open, close, and write calls for the output file handle happen in the main event loop. These operations all require making system calls to the program's host operating system, which may block the event loop for significant amounts of time and prevent other coroutines from making progress. This could hurt overall responsiveness and increase latency, especially for programs such as highly concurrent servers.

I can detect when this problem happens by passing the debug=True parameter to the asyncio.run function. Here, I show how the file and line of a bad coroutine, presumably blocked on a slow system call, can be identified:

```
import time

async def slow_coroutine():
    time.sleep(0.5)  # Simulating slow I/O

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

```
>>>
Executing <Task finished name='Task-1' coro=<slow_coroutine()
➥done, defined at example.py:29> result=None created
➥at .../asyncio/base_events.py:487> took 0.503 seconds
...
```

If I want the most responsive program possible, I need to minimize the potential system calls that are made from within the event loop. In this case, I can create a new `Thread` subclass (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism") that encapsulates everything required to write to the output file using its own event loop:

```
from threading import Thread

class WriteThread(Thread):
    def __init__(self, output_path):
        super().__init__()
        self.output_path = output_path
        self.output = None
        self.loop = asyncio.new_event_loop()

    def run(self):
        asyncio.set_event_loop(self.loop)
        with open(self.output_path, 'wb') as self.output:
            self.loop.run_forever()

        # Run one final round of callbacks so the await on
        # stop() in another event loop will be resolved.
        self.loop.run_until_complete(asyncio.sleep(0))
```

Coroutines in other threads can directly call and `await` on the `write` method of this class, since it's merely a thread-safe wrapper around the `real_write` method that actually does the I/O. This eliminates the need for a `Lock` (see Item 54: "Use `Lock` to Prevent Data Races in Threads"):

```
    async def real_write(self, data):
        self.output.write(data)

    async def write(self, data):
        coro = self.real_write(data)
        future = asyncio.run_coroutine_threadsafe(
            coro, self.loop)
        await asyncio.wrap_future(future)
```

Other coroutines can tell the worker thread when to stop in a thread-safe manner, using similar boilerplate:

```
    async def real_stop(self):
        self.loop.stop()
```

```
    async def stop(self):
        coro = self.real_stop()
        future = asyncio.run_coroutine_threadsafe(
            coro, self.loop)
        await asyncio.wrap_future(future)
```

I can also define the `__aenter__` and `__aexit__` methods to allow this class to be used in `with` statements (see Item 66: "Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior"). This ensures that the worker thread starts and stops at the right times without slowing down the main event loop thread:

```
    async def __aenter__(self):
        loop = asyncio.get_event_loop()
        await loop.run_in_executor(None, self.start)
        return self

    async def __aexit__(self, *_):
        await self.stop()
```

With this new `WriteThread` class, I can refactor `run_tasks` into a fully asynchronous version that's easy to read and completely avoids running slow system calls in the main event loop thread:

```
def readline(handle):
    ...

async def tail_async(handle, interval, write_func):
    ...

async def run_fully_async(handles, interval, output_path):
    async with WriteThread(output_path) as output:
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, output.write)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

I can verify that this works as expected, given a set of input handles and an output file path:

```
def confirm_merge(input_paths, output_path):
    ...
```

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_fully_async(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Things to Remember

- Making system calls in coroutines—including blocking I/O and starting threads—can reduce program responsiveness and increase the perception of latency.
- Pass the `debug=True` parameter to `asyncio.run` in order to detect when certain coroutines are preventing the event loop from reacting quickly.

Item 64: Consider `concurrent.futures` for True Parallelism

At some point in writing Python programs, you may hit the performance wall. Even after optimizing your code (see Item 70: "Profile Before Optimizing"), your program's execution may still be too slow for your needs. On modern computers that have an increasing number of CPU cores, it's reasonable to assume that one solution would be parallelism. What if you could split your code's computation into independent pieces of work that run simultaneously across multiple CPU cores?

Unfortunately, Python's global interpreter lock (GIL) prevents true parallelism in threads (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"), so that option is out. Another common suggestion is to rewrite your most performance-critical code as an extension module, using the C language. C gets you closer to the bare metal and can run faster than Python, eliminating the need for parallelism in some cases. C extensions can also start native threads independent of the Python interpreter that run in parallel and utilize multiple CPU cores with no concern for the GIL. Python's API for C extensions is well documented and a good choice for an escape hatch. It's also worth checking out tools like SWIG (https://github.com/swig/swig) and CLIF (https://github.com/google/clif) to aid in extension development.

But rewriting your code in C has a high cost. Code that is short and understandable in Python can become verbose and complicated in C. Such a port requires extensive testing to ensure that the functionality

is equivalent to the original Python code and that no bugs have been introduced. Sometimes it's worth it, which explains the large ecosystem of C-extension modules in the Python community that speed up things like text parsing, image compositing, and matrix math. There are even open source tools such as Cython (https://cython.org) and Numba (https://numba.pydata.org) that can ease the transition to C.

The problem is that moving one piece of your program to C isn't sufficient most of the time. Optimized Python programs usually don't have one major source of slowness; rather, there are often many significant contributors. To get the benefits of C's bare metal and threads, you'd need to port large parts of your program, drastically increasing testing needs and risk. There must be a better way to preserve your investment in Python to solve difficult computational problems.

The `multiprocessing` built-in module, which is easily accessed via the `concurrent.futures` built-in module, may be exactly what you need (see Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency" for a related example). It enables Python to utilize multiple CPU cores in parallel by running additional interpreters as child processes. These child processes are separate from the main interpreter, so their global interpreter locks are also separate. Each child can fully utilize one CPU core. Each child has a link to the main process where it receives instructions to do computation and returns results.

For example, say that I want to do something computationally intensive with Python and utilize multiple CPU cores. I'll use an implementation of finding the greatest common divisor of two numbers as a proxy for a more computationally intense algorithm (like simulating fluid dynamics with the Navier–Stokes equation):

```
# my_module.py
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
    assert False, 'Not reachable'
```

Running this function in serial takes a linearly increasing amount of time because there is no parallelism:

```
# run_serial.py
import my_module
import time
```

```
NUMBERS = [
    (1963309, 2265973), (2030677, 3814172),
    (1551645, 2229620), (2039045, 2020802),
    (1823712, 1924928), (2293129, 1020491),
    (1281238, 2273782), (3823812, 4237281),
    (3812741, 4729139), (1292391, 2123811),
]

def main():
    start = time.time()
    results = list(map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()
```

```
>>>
Took 1.173 seconds
```

Running this code on multiple Python threads will yield no speed improvement because the GIL prevents Python from using multiple CPU cores in parallel. Here, I do the same computation as above but using the `concurrent.futures` module with its `ThreadPoolExecutor` class and two worker threads (to match the number of CPU cores on my computer):

```
# run_threads.py
import my_module
from concurrent.futures import ThreadPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ThreadPoolExecutor(max_workers=2)
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')
```

```
if __name__ == '__main__':
    main()
```

```
>>>
Took 1.436 seconds
```

It's even slower this time because of the overhead of starting and communicating with the pool of threads.

Now for the surprising part: Changing a single line of code causes something magical to happen. If I replace the `ThreadPoolExecutor` with the `ProcessPoolExecutor` from the `concurrent.futures` module, everything speeds up:

```
# run_parallel.py
import my_module
from concurrent.futures import ProcessPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ProcessPoolExecutor(max_workers=2)  # The one change
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()
```

```
>>>
Took 0.683 seconds
```

Running on my dual-core machine, this is significantly faster! How is this possible? Here's what the `ProcessPoolExecutor` class actually does (via the low-level constructs provided by the `multiprocessing` module):

1. It takes each item from the `numbers` input data to `map`.
2. It serializes the item into binary data by using the `pickle` module (see Item 68: "Make `pickle` Reliable with `copyreg`").
3. It copies the serialized data from the main interpreter process to a child interpreter process over a local socket.

4. It deserializes the data back into Python objects, using `pickle` in the child process.
5. It imports the Python module containing the `gcd` function.
6. It runs the function on the input data in parallel with other child processes.
7. It serializes the result back into binary data.
8. It copies that binary data back through the socket.
9. It deserializes the binary data back into Python objects in the parent process.
10. It merges the results from multiple children into a single `list` to return.

Although it looks simple to the programmer, the `multiprocessing` module and `ProcessPoolExecutor` class do a huge amount of work to make parallelism possible. In most other languages, the only touch point you need to coordinate two threads is a single lock or atomic operation (see Item 54: "Use `Lock` to Prevent Data Races in Threads" for an example). The overhead of using `multiprocessing` via `ProcessPoolExecutor` is high because of all of the serialization and deserialization that must happen between the parent and child processes.

This scheme is well suited to certain types of isolated, high-leverage tasks. By *isolated*, I mean functions that don't need to share state with other parts of the program. By *high-leverage tasks*, I mean situations in which only a small amount of data must be transferred between the parent and child processes to enable a large amount of computation. The greatest common divisor algorithm is one example of this, but many other mathematical algorithms work similarly.

If your computation doesn't have these characteristics, then the overhead of `ProcessPoolExecutor` may prevent it from speeding up your program through parallelization. When that happens, `multiprocessing` provides more advanced facilities for shared memory, cross-process locks, queues, and proxies. But all of these features are very complex. It's hard enough to reason about such tools in the memory space of a single process shared between Python threads. Extending that complexity to other processes and involving sockets makes this much more difficult to understand.

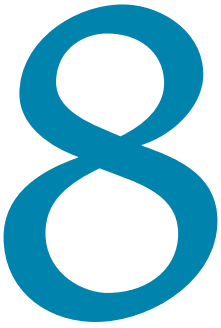
I suggest that you initially avoid all parts of the `multiprocessing` built-in module. You can start by using the `ThreadPoolExecutor` class to run isolated, high-leverage functions in threads. Later you can move to the `ProcessPoolExecutor` to get a speedup. Finally, when

you've completely exhausted the other options, you can consider using the `multiprocessing` module directly.

Things to Remember

- Moving CPU bottlenecks to C-extension modules can be an effective way to improve performance while maximizing your investment in Python code. However, doing so has a high cost and may introduce bugs.
- The `multiprocessing` module provides powerful tools that can parallelize certain types of Python computation with minimal effort.
- The power of `multiprocessing` is best accessed through the `concurrent.futures` built-in module and its simple `ProcessPoolExecutor` class.
- Avoid the advanced (and complicated) parts of the `multiprocessing` module until you've exhausted all other options.

This page intentionally left blank



Robustness and Performance

Once you've written a useful Python program, the next step is to *productionize* your code so it's bulletproof. Making programs dependable when they encounter unexpected circumstances is just as important as making programs with correct functionality. Python has built-in features and modules that aid in hardening your programs so they are robust in a wide variety of situations.

One dimension of robustness is scalability and performance. When you're implementing Python programs that handle a non-trivial amount of data, you'll often see slowdowns caused by the algorithmic complexity of your code or other types of computational overhead. Luckily, Python includes many of the algorithms and data structures you need to achieve high performance with minimal effort.

Item 65: Take Advantage of Each Block in `try/except/else/finally`

There are four distinct times when you might want to take action during exception handling in Python. These are captured in the functionality of `try`, `except`, `else`, and `finally` blocks. Each block serves a unique purpose in the compound statement, and their various combinations are useful (see Item 87: "Define a Root `Exception` to Insulate Callers from APIs" for another example).

`finally` Blocks

Use `try/finally` when you want exceptions to propagate up but also want to run cleanup code even when exceptions occur. One common usage of `try/finally` is for reliably closing file handles (see Item 66: "Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior" for another—likely better—approach):

```
def try_finally_example(filename):
    print('* Opening file')
```