Item 52: Use subprocess to Manage Child Processes

Python has battle-hardened libraries for running and managing child processes. This makes it a great language for gluing together other tools, such as command-line utilities. When existing shell scripts get complicated, as they often do over time, graduating them to a rewrite in Python for the sake of readability and maintainability is a natural choice.

Child processes started by Python are able to run in parallel, enabling you to use Python to consume all of the CPU cores of a machine and maximize the throughput of programs. Although Python itself may be CPU bound (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"), it's easy to use Python to drive and coordinate CPU-intensive workloads.

Python has many ways to run subprocesses (e.g., `os.popen`, `os.exec*`), but the best choice for managing child processes is to use the `subprocess` built-in module. Running a child process with `subprocess` is simple. Here, I use the module's `run` convenience function to start a process, read its output, and verify that it terminated cleanly:

```
import subprocess

result = subprocess.run(
    ['echo', 'Hello from the child!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode()  # No exception means clean exit
print(result.stdout)
```

```
>>>
Hello from the child!
```

Note

> The examples in this item assume that your system has the `echo`, `sleep`, and `openssl` commands available. On Windows, this may not be the case. Please refer to the full example code for this item to see specific directions on how to run these snippets on Windows.

Child processes run independently from their parent process, the Python interpreter. If I create a subprocess using the `Popen` class instead of the `run` function, I can poll child process status periodically while Python does other work:

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Working...')
```

```
    # Some time-consuming work here
    ...

print('Exit status', proc.poll())
```

```
>>>
Working...
Working...
Working...
Working...
Exit status 0
```

Decoupling the child process from the parent frees up the parent process to run many child processes in parallel. Here, I do this by starting all the child processes together with `Popen` upfront:

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

Later, I wait for them to finish their I/O and terminate with the `communicate` method:

```
for proc in sleep_procs:
    proc.communicate()

end = time.time()
delta = end - start
print(f'Finished in {delta:.3} seconds')
```

```
>>>
Finished in 1.05 seconds
```

If these processes ran in sequence, the total delay would be 10 seconds or more rather than the ~1 second that I measured.

You can also pipe data from a Python program into a subprocess and retrieve its output. This allows you to utilize many other programs to do work in parallel. For example, say that I want to use the `openssl` command-line tool to encrypt some data. Starting the child process with command-line arguments and I/O pipes is easy:

```
import os
def run_encrypt(data):
    env = os.environ.copy()
```

```
    env['password'] = 'zf7ShyBhZOraQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush()  # Ensure that the child gets input
    return proc
```

Here, I pipe random bytes into the encryption function, but in practice this input pipe would be fed data from user input, a file handle, a network socket, and so on:

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_encrypt(data)
    procs.append(proc)
```

The child processes run in parallel and consume their input. Here, I wait for them to finish and then retrieve their final output. The output is random encrypted bytes as expected:

```
for proc in procs:
    out, _ = proc.communicate()
    print(out[-10:])
```

```
>>>
b'\x8c(\xed\xc7m1\xf0F4\xe6'
b'\x0eD\x97\xe9>\x10h{\xbd\xf0'
b'g\x93)\x14U\xa9\xdc\xdd\x04\xd2'
```

It's also possible to create chains of parallel processes, just like UNIX pipelines, connecting the output of one child process to the input of another, and so on. Here's a function that starts the `openssl` command-line tool as a subprocess to generate a Whirlpool hash of the input stream:

```
def run_hash(input_stdin):
    return subprocess.Popen(
        ['openssl', 'dgst', '-whirlpool', '-binary'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
```

Now, I can kick off one set of processes to encrypt some data and another set of processes to subsequently hash their encrypted output. Note that I have to be careful with how the `stdout` instance of the

upstream process is retained by the Python interpreter process that's starting this pipeline of child processes:

```
encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)

    # Ensure that the child consumes the input stream and
    # the communicate() method doesn't inadvertently steal
    # input from the child. Also lets SIGPIPE propagate to
    # the upstream process if the downstream process dies.
    encrypt_proc.stdout.close()
    encrypt_proc.stdout = None
```

The I/O between the child processes happens automatically once they are started. All I need to do is wait for them to finish and print the final output:

```
for proc in encrypt_procs:
    proc.communicate()
    assert proc.returncode == 0

for proc in hash_procs:
    out, _ = proc.communicate()
    print(out[-10:])
    assert proc.returncode == 0
```

```
>>>
b'\xe2j\x98h\xfd\xec\xe7T\xd84'
b'\xf3.i\x01\xd74|\xf2\x94E'
b'5_n\xc3-\xe6j\xeb[i'
```

If I'm worried about the child processes never finishing or somehow blocking on input or output pipes, I can pass the `timeout` parameter to the `communicate` method. This causes an exception to be raised if the child process hasn't finished within the time period, giving me a chance to terminate the misbehaving subprocess:

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)
```

```
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())
```

```
>>>
Exit status -15
```

Things to Remember

- Use the `subprocess` module to run child processes and manage their input and output streams.
- Child processes run in parallel with the Python interpreter, enabling you to maximize your usage of CPU cores.
- Use the `run` convenience function for simple usage, and the `Popen` class for advanced usage like UNIX-style pipelines.
- Use the `timeout` parameter of the `communicate` method to avoid deadlocks and hanging child processes.

Item 53: Use Threads for Blocking I/O, Avoid for Parallelism

The standard implementation of Python is called CPython. CPython runs a Python program in two steps. First, it parses and compiles the source text into *bytecode*, which is a low-level representation of the program as 8-bit instructions. (As of Python 3.6, however, it's technically *wordcode* with 16-bit instructions, but the idea is the same.) Then, CPython runs the bytecode using a stack-based interpreter. The bytecode interpreter has state that must be maintained and coherent while the Python program executes. CPython enforces coherence with a mechanism called the *global interpreter lock* (GIL).

Essentially, the GIL is a mutual-exclusion lock (mutex) that prevents CPython from being affected by preemptive multithreading, where one thread takes control of a program by interrupting another thread. Such an interruption could corrupt the interpreter state (e.g., garbage collection reference counts) if it comes at an unexpected time. The GIL prevents these interruptions and ensures that every bytecode instruction works correctly with the CPython implementation and its C-extension modules.

The GIL has an important negative side effect. With programs written in languages like C++ or Java, having multiple threads of execution

means that a program could utilize multiple CPU cores at the same time. Although Python supports multiple threads of execution, the GIL causes only one of them to ever make forward progress at a time. This means that when you reach for threads to do parallel computation and speed up your Python programs, you will be sorely disappointed.

For example, say that I want to do something computationally intensive with Python. Here, I use a naive number factorization algorithm as a proxy:

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

Factoring a set of numbers in serial takes quite a long time:

```
import time

numbers = [2139079, 1214759, 1516637, 1852285]
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.399 seconds
```

Using multiple threads to do this computation would make sense in other languages because I could take advantage of all the CPU cores of my computer. Let me try that in Python. Here, I define a Python thread for doing the same computation as before:

```
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))
```

Then, I start a thread for each number to factorize in parallel:

```
start = time.time()

threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Finally, I wait for all of the threads to finish:

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

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.446 seconds
```

Surprisingly, this takes even longer than running `factorize` in serial. With one thread per number, you might expect less than a 4x speedup in other languages due to the overhead of creating threads and coordinating with them. You might expect only a 2x speedup on the dual-core machine I used to run this code. But you wouldn't expect the performance of these threads to be worse when there are multiple CPUs to utilize. This demonstrates the effect of the GIL (e.g., lock contention and scheduling overhead) on programs running in the standard CPython interpreter.

There are ways to get CPython to utilize multiple cores, but they don't work with the standard `Thread` class (see Item 64: "Consider `concurrent.futures` for True Parallelism"), and they can require substantial effort. Given these limitations, why does Python support threads at all? There are two good reasons.

First, multiple threads make it easy for a program to seem like it's doing multiple things at the same time. Managing the juggling act of simultaneous tasks is difficult to implement yourself (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for an example). With threads, you can leave it to Python to run your functions concurrently. This works because CPython ensures a level of fairness between Python threads of execution, even though only one of them makes forward progress at a time due to the GIL.

The second reason Python supports threads is to deal with blocking I/O, which happens when Python does certain types of system calls.

A Python program uses system calls to ask the computer's operating system to interact with the external environment on its behalf. Blocking I/O includes things like reading and writing files, interacting with networks, communicating with devices like displays, and so on. Threads help handle blocking I/O by insulating a program from the time it takes for the operating system to respond to requests.

For example, say that I want to send a signal to a remote-controlled helicopter through a serial port. I'll use a slow system call (`select`) as a proxy for this activity. This function asks the operating system to block for 0.1 seconds and then return control to my program, which is similar to what would happen when using a synchronous serial port:

```
import select
import socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

Running this system call in serial requires a linearly increasing amount of time:

```
start = time.time()

for _ in range(5):
    slow_systemcall()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.510 seconds
```

The problem is that while the `slow_systemcall` function is running, my program can't make any other progress. My program's main thread of execution is blocked on the `select` system call. This situation is awful in practice. You need to be able to compute your helicopter's next move while you're sending it a signal; otherwise, it'll crash. When you find yourself needing to do blocking I/O and computation simultaneously, it's time to consider moving your system calls to threads.

Here, I run multiple invocations of the `slow_systemcall` function in separate threads. This would allow me to communicate with multiple serial ports (and helicopters) at the same time while leaving the main thread to do whatever computation is required:

```
start = time.time()
```

```
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

With the threads started, here I do some work to calculate the next helicopter move before waiting for the system call threads to finish:

```
def compute_helicopter_location(index):
    ...

for i in range(5):
    compute_helicopter_location(i)

for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')
```

```
>>>
Took 0.108 seconds
```

The parallel time is ~5x less than the serial time. This shows that all the system calls will run in parallel from multiple Python threads even though they're limited by the GIL. The GIL prevents my Python code from running in parallel, but it doesn't have an effect on system calls. This works because Python threads release the GIL just before they make system calls, and they reacquire the GIL as soon as the system calls are done.

There are many other ways to deal with blocking I/O besides using threads, such as the `asyncio` built-in module, and these alternatives have important benefits. But those options might require extra work in refactoring your code to fit a different model of execution (see Item 60: "Achieve Highly Concurrent I/O with Coroutines" and Item 62: "Mix Threads and Coroutines to Ease the Transition to `asyncio`"). Using threads is the simplest way to do blocking I/O in parallel with minimal changes to your program.

Things to Remember

- ✦ Python threads can't run in parallel on multiple CPU cores because of the global interpreter lock (GIL).

- Python threads are still useful despite the GIL because they provide an easy way to do multiple things seemingly at the same time.
- Use Python threads to make multiple system calls in parallel. This allows you to do blocking I/O at the same time as computation.

Item 54: Use Lock to Prevent Data Races in Threads

After learning about the global interpreter lock (GIL) (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"), many new Python programmers assume they can forgo using mutual-exclusion locks (also called *mutexes*) in their code altogether. If the GIL is already preventing Python threads from running on multiple CPU cores in parallel, it must also act as a lock for a program's data structures, right? Some testing on types like lists and dictionaries may even show that this assumption appears to hold.

But beware, this is not truly the case. The GIL will not protect you. Although only one Python thread runs at a time, a thread's operations on data structures can be interrupted between any two bytecode instructions in the Python interpreter. This is dangerous if you access the same objects from multiple threads simultaneously. The invariants of your data structures could be violated at practically any time because of these interruptions, leaving your program in a corrupted state.

For example, say that I want to write a program that counts many things in parallel, like sampling light levels from a whole network of sensors. If I want to determine the total number of light samples over time, I can aggregate them with a new class:

```
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Imagine that each sensor has its own worker thread because reading from the sensor requires blocking I/O. After each sensor measurement, the worker thread increments the counter up to a maximum number of desired readings:

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Read from the sensor
        ...
        counter.increment(1)
```

Here, I run one `worker` thread for each sensor in parallel and wait for them all to finish their readings:

```
from threading import Thread

how_many = 10**5
counter = Counter()

threads = []
for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')
```

```
>>>
Counter should be 500000, got 246760
```

This seemed straightforward, and the outcome should have been obvious, but the result is way off! What happened here? How could something so simple go so wrong, especially since only one Python interpreter thread can run at a time?

The Python interpreter enforces fairness between all of the threads that are executing to ensure they get roughly equal processing time. To do this, Python suspends a thread as it's running and resumes another thread in turn. The problem is that you don't know exactly when Python will suspend your threads. A thread can even be paused seemingly halfway through what looks like an atomic operation. That's what happened in this case.

The body of the `Counter` object's `increment` method looks simple, and is equivalent to this statement from the perspective of the worker thread:

```
counter.count += 1
```

But the += operator used on an object attribute actually instructs Python to do three separate operations behind the scenes. The statement above is equivalent to this:

```
value = getattr(counter, 'count')
result = value + 1
setattr(counter, 'count', result)
```

Python threads incrementing the counter can be suspended between any two of these operations. This is problematic if the way the operations interleave causes old versions of `value` to be assigned to the counter. Here's an example of bad interaction between two threads, A and B:

```
# Running in Thread A
value_a = getattr(counter, 'count')
# Context switch to Thread B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Context switch back to Thread A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

Thread B interrupted thread A before it had completely finished. Thread B ran and finished, but then thread A resumed mid-execution, overwriting all of thread B's progress in incrementing the counter. This is exactly what happened in the light sensor example above.

To prevent data races like these, and other forms of data structure corruption, Python includes a robust set of tools in the `threading` built-in module. The simplest and most useful of them is the `Lock` class, a mutual-exclusion lock (mutex).

By using a lock, I can have the `Counter` class protect its current value against simultaneous accesses from multiple threads. Only one thread will be able to acquire the lock at a time. Here, I use a `with` statement to acquire and release the lock; this makes it easier to see which code is executing while the lock is held (see Item 66: "Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior" for background):

```
from threading import Lock

class LockingCounter:
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Now, I run the `worker` threads as before but use a `LockingCounter` instead:

```
counter = LockingCounter()

for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')
```

```
>>>
Counter should be 500000, got 500000
```

The result is exactly what I expect. `Lock` solved the problem.

Things to Remember

- Even though Python has a global interpreter lock, you're still responsible for protecting against data races between the threads in your programs.
- Your programs will corrupt their data structures if you allow multiple threads to modify the same objects without mutual-exclusion locks (mutexes).
- Use the `Lock` class from the `threading` built-in module to enforce your program's invariants between multiple threads.

Item 55: Use `Queue` to Coordinate Work Between Threads

Python programs that do many things concurrently often need to coordinate their work. One of the most useful arrangements for concurrent work is a pipeline of functions.

A pipeline works like an assembly line used in manufacturing. Pipelines have many phases in serial, with a specific function for each phase. New pieces of work are constantly being added to the beginning of the pipeline. The functions can operate concurrently, each

working on the piece of work in its phase. The work moves forward as each function completes until there are no phases remaining. This approach is especially good for work that includes blocking I/O or subprocesses—activities that can easily be parallelized using Python (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism").

For example, say I want to build a system that will take a constant stream of images from my digital camera, resize them, and then add them to a photo gallery online. Such a program could be split into three phases of a pipeline. New images are retrieved in the first phase. The downloaded images are passed through the resize function in the second phase. The resized images are consumed by the upload function in the final phase.

Imagine that I've already written Python functions that execute the phases: `download`, `resize`, `upload`. How do I assemble a pipeline to do the work concurrently?

```
def download(item):
    ...

def resize(item):
    ...

def upload(item):
    ...
```

The first thing I need is a way to hand off work between the pipeline phases. This can be modeled as a thread-safe producer–consumer queue (see Item 54: "Use Lock to Prevent Data Races in Threads" to understand the importance of thread safety in Python; see Item 71: "Prefer deque for Producer–Consumer Queues" to understand queue performance):

```
from collections import deque
from threading import Lock

class MyQueue:
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

The producer, my digital camera, adds new images to the end of the deque of pending items:

```
    def put(self, item):
        with self.lock:
            self.items.append(item)
```

The consumer, the first phase of the processing pipeline, removes images from the front of the deque of pending items:

```
    def get(self):
        with self.lock:
            return self.items.popleft()
```

Here, I represent each phase of the pipeline as a Python thread that takes work from one queue like this, runs a function on it, and puts the result on another queue. I also track how many times the worker has checked for new input and how much work it's completed:

```
from threading import Thread
import time

class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

The trickiest part is that the worker thread must properly handle the case where the input queue is empty because the previous phase hasn't completed its work yet. This happens where I catch the `IndexError` exception below. You can think of this as a holdup in the assembly line:

```
    def run(self):
        while True:
            self.polled_count += 1
            try:
                item = self.in_queue.get()
            except IndexError:
                time.sleep(0.01)  # No work to do
            else:
                result = self.func(item)
                self.out_queue.put(result)
                self.work_done += 1
```

Now, I can connect the three phases together by creating the queues for their coordination points and the corresponding worker threads:

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
```

```
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

I can start the threads and then inject a bunch of work into the first phase of the pipeline. Here, I use a plain `object` instance as a proxy for the real data required by the `download` function:

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

for _ in range(1000):
    download_queue.put(object())
```

Now, I wait for all of the items to be processed by the pipeline and end up in the `done_queue`:

```
while len(done_queue.items) < 1000:
    # Do something useful while waiting
    ...
```

This runs properly, but there's an interesting side effect caused by the threads polling their input queues for new work. The tricky part, where I catch `IndexError` exceptions in the `run` method, executes a large number of times:

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print(f'Processed {processed} items after '
      f'polling {polled} times')
```

```
>>>
Processed 1000 items after polling 3035 times
```

When the worker functions vary in their respective speeds, an earlier phase can prevent progress in later phases, backing up the pipeline. This causes later phases to starve and constantly check their input queues for new work in a tight loop. The outcome is that worker threads waste CPU time doing nothing useful; they're constantly raising and catching `IndexError` exceptions.

But that's just the beginning of what's wrong with this implementation. There are three more problems that you should also avoid. First, determining that all of the input work is complete requires yet another busy wait on the `done_queue`. Second, in `Worker`, the `run` method will execute forever in its busy loop. There's no obvious way to signal to a worker thread that it's time to exit.

Third, and worst of all, a backup in the pipeline can cause the program to crash arbitrarily. If the first phase makes rapid progress but the second phase makes slow progress, then the queue connecting the first phase to the second phase will constantly increase in size. The second phase won't be able to keep up. Given enough time and input data, the program will eventually run out of memory and die.

The lesson here isn't that pipelines are bad; it's that it's hard to build a good producer–consumer queue yourself. So why even try?

`Queue` to the Rescue

The `Queue` class from the `queue` built-in module provides all of the functionality you need to solve the problems outlined above.

`Queue` eliminates the busy waiting in the worker by making the `get` method block until new data is available. For example, here I start a thread that waits for some input data on a queue:

```
from queue import Queue

my_queue = Queue()

def consumer():
    print('Consumer waiting')
    my_queue.get()              # Runs after put() below
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

Even though the thread is running first, it won't finish until an item is `put` on the `Queue` instance and the `get` method has something to return:

```
print('Producer putting')
my_queue.put(object())          # Runs before get() above
print('Producer done')
thread.join()
```

```
>>>
Consumer waiting
Producer putting
Producer done
Consumer done
```

To solve the pipeline backup issue, the `Queue` class lets you specify the maximum amount of pending work to allow between two phases.

This buffer size causes calls to put to block when the queue is already full. For example, here I define a thread that waits for a while before consuming a queue:

```
my_queue = Queue(1)              # Buffer size of 1

def consumer():
    time.sleep(0.1)              # Wait
    my_queue.get()               # Runs second
    print('Consumer got 1')
    my_queue.get()               # Runs fourth
    print('Consumer got 2')
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

The wait should allow the producer thread to put both objects on the queue before the consumer thread ever calls get. But the Queue size is one. This means the producer adding items to the queue will have to wait for the consumer thread to call get at least once before the second call to put will stop blocking and add the second item to the queue:

```
my_queue.put(object())           # Runs first
print('Producer put 1')
my_queue.put(object())           # Runs third
print('Producer put 2')
print('Producer done')
thread.join()
```

```
>>>
Producer put 1
Consumer got 1
Producer put 2
Producer done
Consumer got 2
Consumer done
```

The Queue class can also track the progress of work using the task_done method. This lets you wait for a phase's input queue to drain and eliminates the need to poll the last phase of a pipeline (as with the done_queue above). For example, here I define a consumer thread that calls task_done when it finishes working on an item:

```
in_queue = Queue()
```

```
def consumer():
    print('Consumer waiting')
    work = in_queue.get()       # Runs second
    print('Consumer working')
    # Doing work
    ...
    print('Consumer done')
    in_queue.task_done()        # Runs third

thread = Thread(target=consumer)
thread.start()
```

Now, the producer code doesn't have to join the consumer thread or poll. The producer can just wait for the `in_queue` to finish by calling `join` on the `Queue` instance. Even once it's empty, the `in_queue` won't be joinable until after `task_done` is called for every item that was ever enqueued:

```
print('Producer putting')
in_queue.put(object())          # Runs first
print('Producer waiting')
in_queue.join()                 # Runs fourth
print('Producer done')
thread.join()
```

```
>>>
Consumer waiting
Producer putting
Producer waiting
Consumer working
Consumer done
Producer done
```

I can put all these behaviors together into a `Queue` subclass that also tells the worker thread when it should stop processing. Here, I define a `close` method that adds a special *sentinel* item to the queue that indicates there will be no more input items after it:

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Then, I define an iterator for the queue that looks for this special `object` and stops iteration when it's found. This `__iter__` method also calls `task_done` at appropriate times, letting me track the progress of

work on the queue (see Item 31: "Be Defensive When Iterating Over Arguments" for details about __iter__):

```
    def __iter__(self):
        while True:
            item = self.get()
            try:
                if item is self.SENTINEL:
                    return  # Cause the thread to exit
                yield item
            finally:
                self.task_done()
```

Now, I can redefine my worker thread to rely on the behavior of the ClosableQueue class. The thread will exit when the for loop is exhausted:

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

I re-create the set of worker threads using the new worker class:

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    StoppableWorker(resize, resize_queue, upload_queue),
    StoppableWorker(upload, upload_queue, done_queue),
]
```

After running the worker threads as before, I also send the stop signal after all the input work has been injected by closing the input queue of the first phase:

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

```
for _ in range(1000):
    download_queue.put(object())

download_queue.close()
```

Finally, I wait for the work to finish by joining the queues that connect the phases. Each time one phase is done, I signal the next phase to stop by closing its input queue. At the end, the `done_queue` contains all of the output objects, as expected:

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')

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

```
>>>
1000 items finished
```

This approach can be extended to use multiple worker threads per phase, which can increase I/O parallelism and speed up this type of program significantly. To do this, first I define some helper functions that start and stop multiple threads. The way `stop_threads` works is by calling `close` on each input queue once per consuming thread, which ensures that all of the workers exit cleanly:

```
def start_threads(count, *args):
    threads = [StoppableWorker(*args) for _ in range(count)]
    for thread in threads:
        thread.start()
    return threads

def stop_threads(closable_queue, threads):
    for _ in threads:
        closable_queue.close()

    closable_queue.join()

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

Then, I connect the pieces together as before, putting objects to process into the top of the pipeline, joining queues and threads along the way, and finally consuming the results:

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()

download_threads = start_threads(
    3, download, download_queue, resize_queue)
resize_threads = start_threads(
    4, resize, resize_queue, upload_queue)
upload_threads = start_threads(
    5, upload, upload_queue, done_queue)

for _ in range(1000):
    download_queue.put(object())

stop_threads(download_queue, download_threads)
stop_threads(resize_queue, resize_threads)
stop_threads(upload_queue, upload_threads)

print(done_queue.qsize(), 'items finished')
```

```
>>>
1000 items finished
```

Although Queue works well in this case of a linear pipeline, there are many other situations for which there are better tools that you should consider (see Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- Pipelines are a great way to organize sequences of work—especially I/O-bound programs—that run concurrently using multiple Python threads.
- Be aware of the many problems in building concurrent pipelines: busy waiting, how to tell workers to stop, and potential memory explosion.
- The Queue class has all the facilities you need to build robust pipelines: blocking operations, buffer sizes, and joining.

Item 56: Know How to Recognize When Concurrency Is Necessary

Inevitably, as the scope of a program grows, it also becomes more complicated. Dealing with expanding requirements in a way that maintains clarity, testability, and efficiency is one of the most difficult parts of programming. Perhaps the hardest type of change to handle is moving from a single-threaded program to one that needs multiple concurrent lines of execution.

Let me demonstrate how you might encounter this problem with an example. Say that I want to implement Conway's Game of Life, a classic illustration of finite state automata. The rules of the game are simple: You have a two-dimensional grid of an arbitrary size. Each cell in the grid can either be alive or empty:

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

The game progresses one tick of the clock at a time. Every tick, each cell counts how many of its neighboring eight cells are still alive. Based on its neighbor count, a cell decides if it will keep living, die, or regenerate. (I'll explain the specific rules further below.) Here's an example of a 5 × 5 Game of Life grid after four generations with time going to the right:

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

I can represent the state of each cell with a simple container class. The class must have methods that allow me to get and set the value of any coordinate. Coordinates that are out of bounds should wrap around, making the grid act like an infinite looping space:

```
class Grid:
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)
```

```
    def get(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def set(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state

    def __str__(self):
        ...
```

To see this class in action, I can create a `Grid` instance and set its initial state to a classic shape called a glider:

```
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)
print(grid)
```

```
>>>
---*-----
----*----
--***----
---------
---------
```

Now, I need a way to retrieve the status of neighboring cells. I can do this with a helper function that queries the grid and returns the count of living neighbors. I use a simple function for the `get` parameter instead of passing in a whole `Grid` instance in order to reduce coupling (see Item 38: "Accept Functions Instead of Classes for Simple Interfaces" for more about this approach):

```
def count_neighbors(y, x, get):
    n_ = get(y - 1, x + 0)  # North
    ne = get(y - 1, x + 1)  # Northeast
    e_ = get(y + 0, x + 1)  # East
    se = get(y + 1, x + 1)  # Southeast
    s_ = get(y + 1, x + 0)  # South
    sw = get(y + 1, x - 1)  # Southwest
    w_ = get(y + 0, x - 1)  # West
    nw = get(y - 1, x - 1)  # Northwest
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
```

```
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Now, I define the simple logic for Conway's Game of Life, based on the game's three rules: Die if a cell has fewer than two neighbors, die if a cell has more than three neighbors, or become alive if an empty cell has exactly three neighbors:

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY     # Die: Too few
        elif neighbors > 3:
            return EMPTY     # Die: Too many
    else:
        if neighbors == 3:
            return ALIVE     # Regenerate
    return state
```

I can connect `count_neighbors` and `game_logic` together in another function that transitions the state of a cell. This function will be called each generation to figure out a cell's current state, inspect the neighboring cells around it, determine what its next state should be, and update the resulting grid accordingly. Again, I use a function interface for `set` instead of passing in the `Grid` instance to make this code more decoupled:

```
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)
```

Finally, I can define a function that progresses the whole grid of cells forward by a single step and then returns a new grid containing the state for the next generation. The important detail here is that I need all dependent functions to call the `get` method on the previous generation's `Grid` instance, and to call the `set` method on the next generation's `Grid` instance. This is how I ensure that all of the cells move in lockstep, which is an essential part of how the game works. This is easy to achieve because I used function interfaces for `get` and `set` instead of passing `Grid` instances:

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

```
    for y in range(grid.height):
        for x in range(grid.width):
            step_cell(y, x, grid.get, next_grid.set)
    return next_grid
```

Now, I can progress the grid forward one generation at a time. You can see how the glider moves down and to the right on the grid based on the simple rules from the `game_logic` function:

```
class ColumnPrinter:
    ...

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

print(columns)
```

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

This works great for a program that can run in one thread on a single machine. But imagine that the program's requirements have changed—as I alluded to above—and now I need to do some I/O (e.g., with a socket) from within the `game_logic` function. For example, this might be required if I'm trying to build a massively multiplayer online game where the state transitions are determined by a combination of the grid state and communication with other players over the Internet.

How can I extend this implementation to support such functionality? The simplest thing to do is to add blocking I/O directly into the `game_logic` function:

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

The problem with this approach is that it's going to slow down the whole program. If the latency of the I/O required is 100 milliseconds (i.e., a reasonably good cross-country, round-trip latency on the

Internet), and there are 45 cells in the grid, then each generation will take a minimum of 4.5 seconds to evaluate because each cell is processed serially in the `simulate` function. That's far too slow and will make the game unplayable. It also scales poorly: If I later wanted to expand the grid to 10,000 cells, I would need over 15 minutes to evaluate each generation.

The solution is to do the I/O in parallel so each generation takes roughly 100 milliseconds, regardless of how big the grid is. The process of spawning a concurrent line of execution for each unit of work—a cell in this case—is called *fan-out*. Waiting for all of those concurrent units of work to finish before moving on to the next phase in a coordinated process—a generation in this case—is called *fan-in*.

Python provides many built-in tools for achieving fan-out and fan-in with various trade-offs. You should understand the pros and cons of each approach and choose the best tool for the job, depending on the situation. See the items that follow for details based on this Game of Life example program (Item 57: "Avoid Creating New `Thread` Instances for On-demand Fan-out," Item 58: "Understand How Using `Queue` for Concurrency Requires Refactoring," Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency," and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- A program often grows to require multiple concurrent lines of execution as its scope and complexity increases.
- The most common types of concurrency coordination are fan-out (generating new units of concurrency) and fan-in (waiting for existing units of concurrency to complete).
- Python has many different ways of achieving fan-out and fan-in.

Item 57: Avoid Creating New `Thread` Instances for On-demand Fan-out

Threads are the natural first tool to reach for in order to do parallel I/O in Python (see Item 53: "Use Threads for Blocking I/O, Avoid for Parallelism"). However, they have significant downsides when you try to use them for fanning out to many concurrent lines of execution.

To demonstrate this, I'll continue with the Game of Life example from before (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for background and the implementations of various functions and classes below). I'll use threads to solve the latency problem

caused by doing I/O in the game_logic function. To begin, threads require coordination using locks to ensure that assumptions within data structures are maintained properly. I can create a subclass of the Grid class that adds locking behavior so an instance can be used by multiple threads simultaneously:

```
from threading import Lock

ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    def __init__(self, height, width):
        super().__init__(height, width)
        self.lock = Lock()

    def __str__(self):
        with self.lock:
            return super().__str__()

    def get(self, y, x):
        with self.lock:
            return super().get(y, x)

    def set(self, y, x, state):
        with self.lock:
            return super().set(y, x, state)
```

Then, I can reimplement the simulate function to *fan out* by creating a thread for each call to step_cell. The threads will run in parallel and won't have to wait on each other's I/O. I can then *fan in* by waiting for all of the threads to complete before moving on to the next generation:

```
from threading import Thread

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)

def simulate_threaded(grid):
    next_grid = LockingGrid(grid.height, grid.width)

    threads = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            thread = Thread(target=step_cell, args=args)
            thread.start()  # Fan out
            threads.append(thread)

    for thread in threads:
        thread.join()       # Fan in

    return next_grid
```

I can run this code using the same implementation of `step_cell` and the same driving code as before with only two lines changed to use the `LockingGrid` and `simulate_threaded` implementations:

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)            # Changed
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 = simulate_threaded(grid)  # Changed

print(columns)
```

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

This works as expected, and the I/O is now parallelized between the threads. However, this code has three big problems:

- The Thread instances require special tools to coordinate with each other safely (see Item 54: "Use Lock to Prevent Data Races in Threads"). This makes the code that uses threads harder to reason about than the procedural, single-threaded code from before. This complexity makes threaded code more difficult to extend and maintain over time.
- Threads require a lot of memory—about 8 MB per executing thread. On many computers, that amount of memory doesn't matter for the 45 threads I'd need in this example. But if the game grid had to grow to 10,000 cells, I would need to create that many threads, which couldn't even fit in the memory of my machine. Running a thread per concurrent activity just won't work.
- Starting a thread is costly, and threads have a negative performance impact when they run due to context switching between them. In this case, all of the threads are started and stopped each generation of the game, which has high overhead and will increase latency beyond the expected I/O time of 100 milliseconds.

This code would also be very difficult to debug if something went wrong. For example, imagine that the game_logic function raises an exception, which is highly likely due to the generally flaky nature of I/O:

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

I can test what this would do by running a Thread instance pointed at this function and redirecting the sys.stderr output from the program to an in-memory StringIO buffer:

```
import contextlib
import io
```

```
fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    thread = Thread(target=game_logic, args=(ALIVE, 3))
    thread.start()
    thread.join()

print(fake_stderr.getvalue())
```

```
>>>
Exception in thread Thread-226:
Traceback (most recent call last):
  File "threading.py", line 917, in _bootstrap_inner
    self.run()
  File "threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "example.py", line 193, in game_logic
    raise OSError('Problem with I/O')
OSError: Problem with I/O
```

An `OSError` exception is raised as expected, but somehow the code that created the `Thread` and called `join` on it is unaffected. How can this be? The reason is that the `Thread` class will independently catch any exceptions that are raised by the `target` function and then write their traceback to `sys.stderr`. Such exceptions are never re-raised to the caller that started the thread in the first place.

Given all of these issues, it's clear that threads are not the solution if you need to constantly create and finish new concurrent functions. Python provides other solutions that are a better fit (see Item 58: "Understand How Using `Queue` for Concurrency Requires Refactoring," Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency", and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- Threads have many downsides: They're costly to start and run if you need a lot of them, they each require a significant amount of memory, and they require special tools like `Lock` instances for coordination.
- Threads do not provide a built-in way to raise exceptions back in the code that started a thread or that is waiting for one to finish, which makes them difficult to debug.

Item 58: Understand How Using Queue for Concurrency Requires Refactoring

In the previous item (see Item 57: "Avoid Creating New Thread Instances for On-demand Fan-out") I covered the downsides of using Thread to solve the parallel I/O problem in the Game of Life example from earlier (see Item 56: "Know How to Recognize When Concurrency Is Necessary" for background and the implementations of various functions and classes below).

The next approach to try is to implement a threaded pipeline using the Queue class from the queue built-in module (see Item 55: "Use Queue to Coordinate Work Between Threads" for background; I rely on the implementations of ClosableQueue and StoppableWorker from that item in the example code below).

Here's the general approach: Instead of creating one thread per cell per generation of the Game of Life, I can create a fixed number of worker threads upfront and have them do parallelized I/O as needed. This will keep my resource usage under control and eliminate the overhead of frequently starting new threads.

To do this, I need two ClosableQueue instances to use for communicating to and from the worker threads that execute the game_logic function:

```
from queue import Queue

class ClosableQueue(Queue):
    ...

in_queue = ClosableQueue()
out_queue = ClosableQueue()
```

I can start multiple threads that will consume items from the in_queue, process them by calling game_logic, and put the results on out_queue. These threads will run concurrently, allowing for parallel I/O and reduced latency for each generation:

```
from threading import Thread

class StoppableWorker(Thread):
    ...

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

```
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def game_logic_thread(item):
    y, x, state, neighbors = item
    try:
        next_state = game_logic(state, neighbors)
    except Exception as e:
        next_state = e
    return (y, x, next_state)

# Start the threads upfront
threads = []
for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, in_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Now, I can redefine the `simulate` function to interact with these queues to request state transition decisions and receive corresponding responses. Adding items to `in_queue` causes *fan-out*, and consuming items from `out_queue` until it's empty causes *fan-in*:

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

class SimulationError(Exception):
    pass

class Grid:
    ...

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

def simulate_pipeline(grid, in_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            neighbors = count_neighbors(y, x, grid.get)
            in_queue.put((y, x, state, neighbors))  # Fan out

    in_queue.join()
    out_queue.close()
```

```
    next_grid = Grid(grid.height, grid.width)
    for item in out_queue:                          # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid
```

The calls to `Grid.get` and `Grid.set` both happen within this new `simulate_pipeline` function, which means I can use the single-threaded implementation of `Grid` instead of the implementation that requires `Lock` instances for synchronization.

This code is also easier to debug than the `Thread` approach used in the previous item. If an exception occurs while doing I/O in the `game_logic` function, it will be caught, propagated to the `out_queue`, and then re-raised in the main thread:

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

simulate_pipeline(Grid(1, 1), in_queue, out_queue)
```

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

The above exception was the direct cause of the following
➥exception:

Traceback ...
SimulationError: (0, 0)
```

I can drive this multithreaded pipeline for repeated generations by calling `simulate_pipeline` in a loop:

```
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 = simulate_pipeline(grid, in_queue, out_queue)

print(columns)

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

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

The results are the same as before. Although I've addressed the memory explosion problem, startup costs, and debugging issues of using threads on their own, many issues remain:

- The `simulate_pipeline` function is even harder to follow than the `simulate_threaded` approach from the previous item.
- Extra support classes were required for `ClosableQueue` and `StoppableWorker` in order to make the code easier to read, at the expense of increased complexity.
- I have to specify the amount of potential parallelism—the number of threads running `game_logic_thread`—upfront based on my expectations of the workload instead of having the system automatically scale up parallelism as needed.
- In order to enable debugging, I have to manually catch exceptions in worker threads, propagate them on a `Queue`, and then re-raise them in the main thread.

However, the biggest problem with this code is apparent if the requirements change again. Imagine that later I needed to do I/O within the `count_neighbors` function in addition to the I/O that was needed within `game_logic`:

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

```
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...
```

In order to make this parallelizable, I need to add another stage to the pipeline that runs `count_neighbors` in a thread. I need to make sure that exceptions propagate correctly between the worker threads and the main thread. And I need to use a `Lock` for the `Grid` class in order to ensure safe synchronization between the worker threads (see Item 54: "Use Lock to Prevent Data Races in Threads" for background and Item 57: "Avoid Creating New Thread Instances for On-demand Fan-out" for the implementation of `LockingGrid`):

```
def count_neighbors_thread(item):
    y, x, state, get = item
    try:
        neighbors = count_neighbors(y, x, get)
    except Exception as e:
        neighbors = e
    return (y, x, state, neighbors)

def game_logic_thread(item):
    y, x, state, neighbors = item
    if isinstance(neighbors, Exception):
        next_state = neighbors
    else:
        try:
            next_state = game_logic(state, neighbors)
        except Exception as e:
            next_state = e
    return (y, x, next_state)

class LockingGrid(Grid):
    ...
```

I have to create another set of `Queue` instances for the `count_neighbors_thread` workers and the corresponding `Thread` instances:

```
in_queue = ClosableQueue()
logic_queue = ClosableQueue()
out_queue = ClosableQueue()

threads = []
```

```
for _ in range(5):
    thread = StoppableWorker(
        count_neighbors_thread, in_queue, logic_queue)
    thread.start()
    threads.append(thread)

for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, logic_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Finally, I need to update `simulate_pipeline` to coordinate the multiple phases in the pipeline and ensure that work fans out and back in correctly:

```
def simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            item = (y, x, state, grid.get)
            in_queue.put(item)          # Fan out

    in_queue.join()
    logic_queue.join()                  # Pipeline sequencing
    out_queue.close()

    next_grid = LockingGrid(grid.height, grid.width)
    for item in out_queue:              # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid
```

With these updated implementations, now I can run the multiphase pipeline end-to-end:

```
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()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    logic_queue.close()
for thread in threads:
    thread.join()
```

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

Again, this works as expected, but it required a lot of changes and boilerplate. The point here is that `Queue` does make it possible to solve fan-out and fan-in problems, but the overhead is very high. Although using `Queue` is a better approach than using `Thread` instances on their own, it's still not nearly as good as some of the other tools provided by Python (see Item 59: "Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency" and Item 60: "Achieve Highly Concurrent I/O with Coroutines").

Things to Remember

- Using `Queue` instances with a fixed number of worker threads improves the scalability of fan-out and fan-in using threads.
- It takes a significant amount of work to refactor existing code to use `Queue`, especially when multiple stages of a pipeline are required.
- Using `Queue` fundamentally limits the total amount of I/O parallelism a program can leverage compared to alternative approaches provided by other built-in Python features and modules.

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.

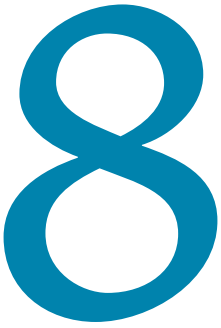
I suggest that you initially avoid all parts of the `multiprocessing` built-in module. You can start by using the `ThreadPoolExecutor` class to run isolated, high-leverage functions in threads. Later you can move to the `ProcessPoolExecutor` to get a speedup. Finally, when

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')
```