
Concurrency Models in Python

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

Not the same, but related.

One is about structure, one is about execution.

Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

—Rob Pike, co-inventor of the Go language¹

This chapter is about how to make Python deal with “lots of things at once.” This may involve concurrent or parallel programming—even academics who are keen on jargon disagree on how to use those terms. I will adopt Rob Pike’s informal definitions in this chapter’s epigraph, but note that I’ve found papers and books that claim to be about parallel computing but are mostly about concurrency.²

Parallelism is a special case of concurrency, in Pike’s view. All parallel systems are concurrent, but not all concurrent systems are parallel. In the early 2000s we used single-core machines that handled 100 processes concurrently on GNU Linux. A modern laptop with 4 CPU cores is routinely running more than 200 processes at any given time under normal, casual use. To execute 200 tasks in parallel, you’d need 200 cores. So, in practice, most computing is concurrent and not parallel. The OS

¹ Slide 8 of the talk “[Concurrency Is Not Parallelism](#)”.

² I studied and worked with Prof. Imre Simon, who liked to say there are two major sins in science: using different words to mean the same thing and using one word to mean different things. Imre Simon (1943–2009) was a pioneer of computer science in Brazil who made seminal contributions to Automata Theory and started the field of Tropical Mathematics. He was also an advocate of free software and free culture.

manages hundreds of processes, making sure each has an opportunity to make progress, even if the CPU itself can't do more than four things at once.

This chapter assumes no prior knowledge of concurrent or parallel programming. After a brief conceptual introduction, we will study simple examples to introduce and compare Python's core packages for concurrent programming: `threading`, `multi` `processing`, and `asyncio`.

The last 30% of the chapter is a high-level overview of third-party tools, libraries, application servers, and distributed task queues—all of which can enhance the performance and scalability of Python applications. These are all important topics, but beyond the scope of a book focused on core Python language features. Nevertheless, I felt it was important to address these themes in this second edition of *Fluent Python*, because Python's fitness for concurrent and parallel computing is not limited to what the standard library provides. That's why YouTube, DropBox, Instagram, Reddit, and others were able to achieve web scale when they started, using Python as their primary language—despite persistent claims that “Python doesn't scale.”

What's New in This Chapter

This chapter is new in the second edition of *Fluent Python*. The spinner examples in “A Concurrent Hello World” on page 701 previously were in the chapter about *asyncio*. Here they are improved, and provide the first illustration of Python's three approaches to concurrency: threads, processes, and native coroutines.

The remaining content is new, except for a few paragraphs that originally appeared in the chapters on `concurrent.futures` and *asyncio*.

“Python in the Multicore World” on page 725 is different from the rest of the book: there are no code examples. The goal is to mention important tools that you may want to study to achieve high-performance concurrency and parallelism beyond what's possible with Python's standard library.

The Big Picture

There are many factors that make concurrent programming hard, but I want to touch on the most basic factor: starting threads or processes is easy enough, but how do you keep track of them?³

When you call a function, the calling code is blocked until the function returns. So you know when the function is done, and you can easily get the value it returned. If

³ This section was suggested by my friend Bruce Eckel—author of books about Kotlin, Scala, Java, and C++.

the function raises an exception, the calling code can surround the call site with `try/except` to catch the error.

Those familiar options are not available when you start a thread or process: you don't automatically know when it's done, and getting back results or errors requires setting up some communication channel, such as a message queue.

Additionally, starting a thread or a process is not cheap, so you don't want to start one of them just to perform a single computation and quit. Often you want to amortize the startup cost by making each thread or process into a “worker” that enters a loop and stands by for inputs to work on. This further complicates communications and introduces more questions. How do you make a worker quit when you don't need it anymore? And how do you make it quit without interrupting a job partway, leaving half-baked data and unreleased resources—like open files? Again the usual answers involve messages and queues.

A coroutine is cheap to start. If you start a coroutine using the `await` keyword, it's easy to get a value returned by it, it can be safely cancelled, and you have a clear site to catch exceptions. But coroutines are often started by the asynchronous framework, and that can make them as hard to monitor as threads or processes.

Finally, Python coroutines and threads are not suitable for CPU-intensive tasks, as we'll see.

That's why concurrent programming requires learning new concepts and coding patterns. Let's first make sure we are on the same page regarding some core concepts.

A Bit of Jargon

Here are some terms I will use for the rest of this chapter and the next two:

Concurrency

The ability to handle multiple pending tasks, making progress one at a time or in parallel (if possible) so that each of them eventually succeeds or fails. A single-core CPU is capable of concurrency if it runs an OS scheduler that interleaves the execution of the pending tasks. Also known as multitasking.

Parallelism

The ability to execute multiple computations at the same time. This requires a multicore CPU, multiple CPUs, a **GPU**, or multiple computers in a cluster.

Execution unit

General term for objects that execute code concurrently, each with independent state and call stack. Python natively supports three kinds of execution units: *processes*, *threads*, and *coroutines*.

Process

An instance of a computer program while it is running, using memory and a slice of the CPU time. Modern desktop operating systems routinely manage hundreds of processes concurrently, with each process isolated in its own private memory space. Processes communicate via pipes, sockets, or memory mapped files—all of which can only carry raw bytes. Python objects must be serialized (converted) into raw bytes to pass from one process to another. This is costly, and not all Python objects are serializable. A process can spawn subprocesses, each called a child process. These are also isolated from each other and from the parent. Processes allow *preemptive multitasking*: the OS scheduler *preempts*—i.e., suspends—each running process periodically to allow other processes to run. This means that a frozen process can't freeze the whole system—in theory.

Thread

An execution unit within a single process. When a process starts, it uses a single thread: the main thread. A process can create more threads to operate concurrently by calling operating system APIs. Threads within a process share the same memory space, which holds live Python objects. This allows easy data sharing between threads, but can also lead to corrupted data when more than one thread updates the same object concurrently. Like processes, threads also enable *preemptive multitasking* under the supervision of the OS scheduler. A thread consumes less resources than a process doing the same job.

Coroutine

A function that can suspend itself and resume later. In Python, *classic coroutines* are built from generator functions, and *native coroutines* are defined with `async def`. “[Classic Coroutines](#)” on page 641 introduced the concept, and [Chapter 21](#) covers the use of native coroutines. Python coroutines usually run within a single thread under the supervision of an *event loop*, also in the same thread. Asynchronous programming frameworks such as *asyncio*, *Curio*, or *Trio* provide an event loop and supporting libraries for nonblocking, coroutine-based I/O. Coroutines support *cooperative multitasking*: each coroutine must explicitly cede control with the `yield` or `await` keyword, so that another may proceed concurrently (but not in parallel). This means that any blocking code in a coroutine blocks the execution of the event loop and all other coroutines—in contrast with the *preemptive multitasking* supported by processes and threads. On the other hand, each coroutine consumes less resources than a thread or process doing the same job.

Queue

A data structure that lets us put and get items, usually in FIFO order: first in, first out. Queues allow separate execution units to exchange application data and control messages, such as error codes and signals to terminate. The implementation of a queue varies according to the underlying concurrency model: the queue

package in Python’s standard library provides queue classes to support threads, while the multiprocessing and asyncio packages implement their own queue classes. The queue and asyncio packages also include queues that are not FIFO: LifoQueue and PriorityQueue.

Lock

An object that execution units can use to synchronize their actions and avoid corrupting data. While updating a shared data structure, the running code should hold an associated lock. This signals other parts of the program to wait until the lock is released before accessing the same data structure. The simplest type of lock is also known as a mutex (for mutual exclusion). The implementation of a lock depends on the underlying concurrency model.

Contention

Dispute over a limited asset. Resource contention happens when multiple execution units try to access a shared resource—such as a lock or storage. There’s also CPU contention, when compute-intensive processes or threads must wait for the OS scheduler to give them a share of the CPU time.

Now let’s use some of that jargon to understand concurrency support in Python.

Processes, Threads, and Python’s Infamous GIL

Here is how the concepts we just saw apply to Python programming, in 10 points:

1. Each instance of the Python interpreter is a process. You can start additional Python processes using the *multiprocessing* or *concurrent.futures* libraries. Python’s *subprocess* library is designed to launch processes to run external programs, regardless of the languages used to write them.
2. The Python interpreter uses a single thread to run the user’s program and the memory garbage collector. You can start additional Python threads using the *threading* or *concurrent.futures* libraries.
3. Access to object reference counts and other internal interpreter state is controlled by a lock, the Global Interpreter Lock (GIL). Only one Python thread can hold the GIL at any time. This means that only one thread can execute Python code at any time, regardless of the number of CPU cores.
4. To prevent a Python thread from holding the GIL indefinitely, Python’s bytecode interpreter pauses the current Python thread every 5ms by default,⁴ releasing the GIL. The thread can then try to reacquire the GIL, but if there are other threads waiting for it, the OS scheduler may pick one of them to proceed.

⁴ Call `sys.getswitchinterval()` to get the interval; change it with `sys.setswitchinterval(s)`.

5. When we write Python code, we have no control over the GIL. But a built-in function or an extension written in C—or any language that interfaces at the Python/C API level—can release the GIL while running time-consuming tasks.
6. Every Python standard library function that makes a syscall⁵ releases the GIL. This includes all functions that perform disk I/O, network I/O, and `time.sleep()`. Many CPU-intensive functions in the NumPy/SciPy libraries, as well as the compressing/decompressing functions from the `zlib` and `bz2` modules, also release the GIL.⁶
7. Extensions that integrate at the Python/C API level can also launch other non-Python threads that are not affected by the GIL. Such GIL-free threads generally cannot change Python objects, but they can read from and write to the memory underlying objects that support the **buffer protocol**, such as `bytearray`, `array.array`, and *NumPy* arrays.
8. The effect of the GIL on network programming with Python threads is relatively small, because the I/O functions release the GIL, and reading or writing to the network always implies high latency—compared to reading and writing to memory. Consequently, each individual thread spends a lot of time waiting anyway, so their execution can be interleaved without major impact on the overall throughput. That’s why David Beazley says: “Python threads are great at doing nothing.”⁷
9. Contention over the GIL slows down compute-intensive Python threads. Sequential, single-threaded code is simpler and faster for such tasks.
10. To run CPU-intensive Python code on multiple cores, you must use multiple Python processes.

Here is a good summary from the `threading` module documentation:⁸

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multicore machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However,

5 A syscall is a call from user code to a function of the operating system kernel. I/O, timers, and locks are some of the kernel services available through syscalls. To learn more, read the Wikipedia “**System call**” article.

6 The `zlib` and `bz2` modules are specifically mentioned in a **python-dev message by Antoine Pitrou**, who contributed the time-slicing GIL logic to Python 3.2.

7 Source: slide 106 of Beazley’s “**Generators: The Final Frontier**” tutorial.

8 Source: last paragraph of the “**Thread objects**” section.

threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

The previous paragraph starts with “CPython implementation detail” because the GIL is not part of the Python language definition. The Jython and IronPython implementations don’t have a GIL. Unfortunately, both are lagging behind—still tracking Python 2.7. The highly performant **PyPy interpreter** also has a GIL in its 2.7 and 3.7 versions—the latest as of June 2021.



This section did not mention coroutines, because by default they share the same Python thread among themselves and with the supervising event loop provided by an asynchronous framework, therefore the GIL does not affect them. It is possible to use multiple threads in an asynchronous program, but the best practice is that one thread runs the event loop and all coroutines, while additional threads carry out specific tasks. This will be explained in “**Delegating Tasks to Executors**” on page 797.

Enough concepts for now. Let’s see some code.

A Concurrent Hello World

During a discussion about threads and how to avoid the GIL, Python contributor Michele Simionato **posted an example** that is like a concurrent “Hello World”: the simplest program to show how Python can “walk and chew gum.”

Simionato’s program uses multiprocessing, but I adapted it to introduce threading and `asyncio` as well. Let’s start with the threading version, which may look familiar if you’ve studied threads in Java or C.

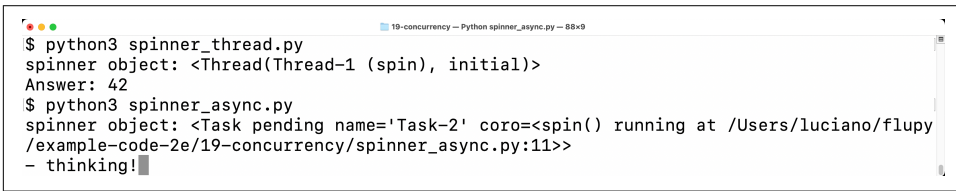
Spinner with Threads

The idea of the next few examples is simple: start a function that blocks for 3 seconds while animating characters in the terminal to let the user know that the program is “thinking” and not stalled.

The script makes an animated spinner displaying each character in the string “\|/-” in the same screen position.⁹ When the slow computation finishes, the line with the spinner is cleared and the result is shown: **Answer: 42.**

⁹ Unicode has lots of characters useful for simple animations, like the **Braille patterns** for example. I used the ASCII characters “\|/-” to keep the examples simple.

Figure 19-1 shows the output of two versions of the spinning example: first with threads, then with coroutines. If you're away from the computer, imagine the \ in the last line is spinning.



```
$ python3 spinner_thread.py
spinner object: <Thread(Thread-1 (spin), initial)>
Answer: 42
$ python3 spinner_async.py
spinner object: <Task pending name='Task-2' coro=<spin() running at /Users/luciano/flupy
/example-code-2e/19-concurrency/spinner_async.py:11>>
- thinking!
```

Figure 19-1. The scripts `spinner_thread.py` and `spinner_async.py` produce similar output: the repr of a spinner object and the text “Answer: 42”. In the screenshot, `spinner_async.py` is still running, and the animated message “/ thinking!” is shown; that line will be replaced by “Answer: 42” after 3 seconds.

Let’s review the `spinner_thread.py` script first. Example 19-1 lists the first two functions in the script, and Example 19-2 shows the rest.

Example 19-1. `spinner_thread.py`: the `spin` and `slow` functions

```
import itertools
import time
from threading import Thread, Event

def spin(msg: str, done: Event) -> None: ❶
    for char in itertools.cycle(r'\|/-'): ❷
        status = f'\r{char} {msg}' ❸
        print(status, end='', flush=True)
        if done.wait(.1): ❹
            break ❺
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='') ❻

def slow() -> int:
    time.sleep(3) ❼
    return 42
```

- ❶ This function will run in a separate thread. The `done` argument is an instance of `threading.Event`, a simple object to synchronize threads.
- ❷ This is an infinite loop because `itertools.cycle` yields one character at a time, cycling through the string forever.
- ❸ The trick for text-mode animation: move the cursor back to the start of the line with the carriage return ASCII control character (`'\r'`).

- ④ The `Event.wait(timeout=None)` method returns `True` when the event is set by another thread; if the `timeout` elapses, it returns `False`. The `.1s` `timeout` sets the “frame rate” of the animation to 10 FPS. If you want the spinner to go faster, use a smaller `timeout`.
- ⑤ Exit the infinite loop.
- ⑥ Clear the status line by overwriting with spaces and moving the cursor back to the beginning.
- ⑦ `slow()` will be called by the main thread. Imagine this is a slow API call over the network. Calling `sleep` blocks the main thread, but the GIL is released so the spinner thread can proceed.



The first important insight of this example is that `time.sleep()` blocks the calling thread but releases the GIL, allowing other Python threads to run.

The `spin` and `slow` functions will execute concurrently. The main thread—the only thread when the program starts—will start a new thread to run `spin` and then call `slow`. By design, there is no API for terminating a thread in Python. You must send it a message to shut down.

The `threading.Event` class is Python’s simplest signalling mechanism to coordinate threads. An `Event` instance has an internal boolean flag that starts as `False`. Calling `Event.set()` sets the flag to `True`. While the flag is false, if a thread calls `Event.wait()`, it is blocked until another thread calls `Event.set()`, at which time `Event.wait()` returns `True`. If a `timeout` in seconds is given to `Event.wait(s)`, this call returns `False` when the `timeout` elapses, or returns `True` as soon as `Event.set()` is called by another thread.

The `supervisor` function, listed in [Example 19-2](#), uses an `Event` to signal the `spin` function to exit.

Example 19-2. `spinner_thread.py`: the supervisor and main functions

```
def supervisor() -> int: ①
    done = Event() ②
    spinner = Thread(target=spin, args=('thinking!', done)) ③
    print(f'spinner object: {spinner}') ④
    spinner.start() ⑤
    result = slow() ⑥
    done.set() ⑦
```

```

    spinner.join() ❸
    return result

def main() -> None:
    result = supervisor() ❹
    print(f'Answer: {result}')

if __name__ == '__main__':
    main()

```

- ❶ supervisor will return the result of slow.
- ❷ The `threading.Event` instance is the key to coordinate the activities of the `main` thread and the `spinner` thread, as explained further down.
- ❸ To create a new `Thread`, provide a function as the `target` keyword argument, and positional arguments to the `target` as a tuple passed via `args`.
- ❹ Display the `spinner` object. The output is `<Thread(Thread-1, initial)>`, where `initial` is the state of the thread—meaning it has not started.
- ❺ Start the `spinner` thread.
- ❻ Call `slow`, which blocks the `main` thread. Meanwhile, the secondary thread is running the `spinner` animation.
- ❼ Set the `Event` flag to `True`; this will terminate the `for` loop inside the `spin` function.
- ❽ Wait until the `spinner` thread finishes.
- ❾ Run the `supervisor` function. I wrote separate `main` and `supervisor` functions to make this example look more like the `asyncio` version in [Example 19-4](#).

When the `main` thread sets the `done` event, the `spinner` thread will eventually notice and exit cleanly.

Now let's take a look at a similar example using the `multiprocessing` package.

Spinner with Processes

The `multiprocessing` package supports running concurrent tasks in separate Python processes instead of threads. When you create a `multiprocessing.Process` instance, a whole new Python interpreter is started as a child process in the background. Since each Python process has its own GIL, this allows your program to use all available

CPU cores—but that ultimately depends on the operating system scheduler. We’ll see practical effects in “A Homegrown Process Pool” on page 716, but for this simple program it makes no real difference.

The point of this section is to introduce multiprocessing and show that its API emulates the threading API, making it easy to convert simple programs from threads to processes, as shown in *spinner_proc.py* (Example 19-3).

Example 19-3. spinner_proc.py: only the changed parts are shown; everything else is the same as spinner_thread.py

```
import itertools
import time
from multiprocessing import Process, Event ❶
from multiprocessing import synchronize ❷

def spin(msg: str, done: synchronize.Event) -> None: ❸

# [snip] the rest of spin and slow functions are unchanged from spinner_thread.py

def supervisor() -> int:
    done = Event()
    spinner = Process(target=spin, ❹
                      args=('thinking!', done))
    print(f'spinner object: {spinner}') ❺
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result

# [snip] main function is unchanged as well
```

- ❶ The basic multiprocessing API imitates the threading API, but type hints and Mypy expose this difference: `multiprocessing.Event` is a function (not a class like `threading.Event`) which returns a `synchronize.Event` instance...
- ❷ ...forcing us to import `multiprocessing.synchronize`...
- ❸ ...to write this type hint.
- ❹ Basic usage of the `Process` class is similar to `Thread`.
- ❺ The `spinner` object is displayed as `<Process name='Process-1' parent=14868 initial>`, where 14868 is the process ID of the Python instance running *spinner_proc.py*.

The basic API of threading and multiprocessing are similar, but their implementation is very different, and multiprocessing has a much larger API to handle the added complexity of multiprocess programming. For example, one challenge when converting from threads to processes is how to communicate between processes that are isolated by the operating system and can't share Python objects. This means that objects crossing process boundaries have to be serialized and deserialized, which creates overhead. In [Example 19-3](#), the only data that crosses the process boundary is the Event state, which is implemented with a low-level OS semaphore in the C code underlying the multiprocessing module.¹⁰



Since Python 3.8, there's a `multiprocessing.shared_memory` package in the standard library, but it does not support instances of user-defined classes. Besides raw bytes, the package allows processes to share a `ShareableList`, a mutable sequence type that can hold a fixed number of items of types `int`, `float`, `bool`, and `None`, as well as `str` and `bytes` up to 10 MB per item. See the [ShareableList](#) documentation for more.

Now let's see how the same behavior can be achieved with coroutines instead of threads or processes.

Spinner with Coroutines



[Chapter 21](#) is entirely devoted to asynchronous programming with coroutines. This is just a high-level introduction to contrast this approach with the threads and processes concurrency models. As such, we will overlook many details.

It is the job of OS schedulers to allocate CPU time to drive threads and processes. In contrast, coroutines are driven by an application-level event loop that manages a queue of pending coroutines, drives them one by one, monitors events triggered by I/O operations initiated by coroutines, and passes control back to the corresponding coroutine when each event happens. The event loop and the library coroutines and the user coroutines all execute in a single thread. Therefore, any time spent in a coroutine slows down the event loop—and all other coroutines.

¹⁰ The semaphore is a fundamental building block that can be used to implement other synchronization mechanisms. Python provides different semaphore classes for use with threads, processes, and coroutines. We'll see `asyncio.Semaphore` in [“Using `asyncio.as_completed` and a Thread” on page 788 \(Chapter 21\)](#).

The coroutine version of the spinner program is easier to understand if we start from the main function, then study the supervisor. That's what [Example 19-4](#) shows.

Example 19-4. spinner_async.py: the main function and supervisor coroutine

```
def main() -> None: ❶
    result = asyncio.run(supervisor()) ❷
    print(f'Answer: {result}')

async def supervisor() -> int: ❸
    spinner = asyncio.create_task(spin('thinking!')) ❹
    print(f'spinner object: {spinner}') ❺
    result = await slow() ❻
    spinner.cancel() ❼
    return result

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

- ❶ main is the only regular function defined in this program—the others are coroutines.
- ❷ The `asyncio.run` function starts the event loop to drive the coroutine that will eventually set the other coroutines in motion. The main function will stay blocked until supervisor returns. The return value of supervisor will be the return value of `asyncio.run`.
- ❸ Native coroutines are defined with `async def`.
- ❹ `asyncio.create_task` schedules the eventual execution of `spin`, immediately returning an instance of `asyncio.Task`.
- ❺ The repr of the spinner object looks like `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:11>>`.
- ❻ The `await` keyword calls `slow`, blocking supervisor until `slow` returns. The return value of `slow` will be assigned to `result`.
- ❼ The `Task.cancel` method raises a `CancelledError` exception inside the `spin` coroutine, as we'll see in [Example 19-5](#).

[Example 19-4](#) demonstrates the three main ways of running a coroutine:

`asyncio.run(coro())`

Called from a regular function to drive a coroutine object that usually is the entry point for all the asynchronous code in the program, like the supervisor in this example. This call blocks until the body of `coro` returns. The return value of the `run()` call is whatever the body of `coro` returns.

`asyncio.create_task(coro())`

Called from a coroutine to schedule another coroutine to execute eventually. This call does not suspend the current coroutine. It returns a `Task` instance, an object that wraps the coroutine object and provides methods to control and query its state.

`await coro()`

Called from a coroutine to transfer control to the coroutine object returned by `coro()`. This suspends the current coroutine until the body of `coro` returns. The value of the `await` expression is whatever the body of `coro` returns.



Remember: invoking a coroutine as `coro()` immediately returns a coroutine object, but does not run the body of the `coro` function. Driving the body of coroutines is the job of the event loop.

Now let's study the `spin` and `slow` coroutines in [Example 19-5](#).

Example 19-5. `spinner_async.py`: the `spin` and `slow` coroutines

```
import asyncio
import itertools

async def spin(msg: str) -> None: ❶
    for char in itertools.cycle(r'\|/-'):
        status = f'\r{char} {msg}'
        print(status, flush=True, end='')
        try:
            await asyncio.sleep(.1) ❷
        except asyncio.CancelledError: ❸
            break
    blanks = ' ' * len(status)
    print(f'\r{blanks}\r', end='')

async def slow() -> int:
    await asyncio.sleep(3) ❹
    return 42
```

- ❶ We don't need the `Event` argument that was used to signal that `slow` had completed its job in `spinner_thread.py` (Example 19-1).
- ❷ Use `await asyncio.sleep(.1)` instead of `time.sleep(.1)`, to pause without blocking other coroutines. See the experiment after this example.
- ❸ `asyncio.CancelledError` is raised when the `cancel` method is called on the `Task` controlling this coroutine. Time to exit the loop.
- ❹ The `slow` coroutine also uses `await asyncio.sleep` instead of `time.sleep`.

Experiment: Break the spinner for an insight

Here is an experiment I recommend to understand how `spinner_async.py` works. Import the `time` module, then go to the `slow` coroutine and replace the line `await asyncio.sleep(3)` with a call to `time.sleep(3)`, like in Example 19-6.

Example 19-6. `spinner_async.py`: replacing `await asyncio.sleep(3)` with `time.sleep(3)`

```
async def slow() -> int:
    time.sleep(3)
    return 42
```

Watching the behavior is more memorable than reading about it. Go ahead, I'll wait.

When you run the experiment, this is what you see:

1. The spinner object is shown, similar to this: `<Task pending name='Task-2' coro=<spin() running at /path/to/spinner_async.py:12>>`.
2. The spinner never appears. The program hangs for 3 seconds.
3. Answer: 42 is displayed and the program ends.

To understand what is happening, recall that Python code using `asyncio` has only one flow of execution, unless you've explicitly started additional threads or processes. That means only one coroutine executes at any point in time. Concurrency is achieved by control passing from one coroutine to another. In Example 19-7, let's focus on what happens in the supervisor and `slow` coroutines during the proposed experiment.

Example 19-7. `spinner_async_experiment.py`: the supervisor and `slow` coroutines

```
async def slow() -> int:
    time.sleep(3) ❹
```

```
return 42
```

```
async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!')) ❶
    print(f'spinner object: {spinner}') ❷
    result = await slow() ❸
    spinner.cancel() ❺
    return result
```

- ❶ The `spinner` task is created, to eventually drive the execution of `spin`.
- ❷ The display shows the Task is “pending.”
- ❸ The `await` expression transfers control to the `slow` coroutine.
- ❹ `time.sleep(3)` blocks for 3 seconds; nothing else can happen in the program, because the main thread is blocked—and it is the only thread. The operating system will continue with other activities. After 3 seconds, `sleep` unblocks, and `slow` returns.
- ❺ Right after `slow` returns, the `spinner` task is cancelled. The flow of control never reached the body of the `spin` coroutine.

The *spinner_async_experiment.py* teaches an important lesson, as explained in the following warning.



Never use `time.sleep(...)` in `asyncio` coroutines unless you want to pause your whole program. If a coroutine needs to spend some time doing nothing, it should await `asyncio.sleep(DELAY)`. This yields control back to the `asyncio` event loop, which can drive other pending coroutines.

Greenlet and gevent

As we discuss concurrency with coroutines, it’s important to mention the *greenlet* package, which has been around for many years and is used at scale.¹¹ The package supports cooperative multitasking through lightweight coroutines—named *greenlets*—that don’t require any special syntax such as `yield` or `await`, therefore are easier to integrate into existing, sequential codebases. *SQLAlchemy 1.4 ORM* uses *greenlets* internally to implement its new *asynchronous API* compatible with *asyncio*.

¹¹ Thanks to tech reviewers Caleb Hattingh and Jürgen Gmach who did not let me overlook *greenlet* and *gevent*.

The *gevent* networking library monkey patches Python’s standard `socket` module making it nonblocking by replacing some of its code with greenlets. To a large extent, *gevent* is transparent to the surrounding code, making it easier to adapt sequential applications and libraries—such as database drivers—to perform concurrent network I/O. Numerous open source projects use *gevent*, including the widely deployed *Gunicorn*—mentioned in “WSGI Application Servers” on page 730.

Supervisors Side-by-Side

The line count of *spinner_thread.py* and *spinner_async.py* is nearly the same. The supervisor functions are the heart of these examples. Let’s compare them in detail.

Example 19-8 lists only the supervisor from Example 19-2.

Example 19-8. spinner_thread.py: the threaded supervisor function

```
def supervisor() -> int:
    done = Event()
    spinner = Thread(target=spin,
                     args=('thinking!', done))
    print('spinner object:', spinner)
    spinner.start()
    result = slow()
    done.set()
    spinner.join()
    return result
```

For comparison, Example 19-9 shows the supervisor coroutine from Example 19-4.

Example 19-9. spinner_async.py: the asynchronous supervisor coroutine

```
async def supervisor() -> int:
    spinner = asyncio.create_task(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow()
    spinner.cancel()
    return result
```

Here is a summary of the differences and similarities to note between the two supervisor implementations:

- An `asyncio.Task` is roughly the equivalent of a `threading.Thread`.
- A `Task` drives a coroutine object, and a `Thread` invokes a callable.
- A coroutine yields control explicitly with the `await` keyword.

- You don't instantiate Task objects yourself, you get them by passing a coroutine to `asyncio.create_task(...)`.
- When `asyncio.create_task(...)` returns a Task object, it is already scheduled to run, but a Thread instance must be explicitly told to run by calling its `start` method.
- In the threaded supervisor, `slow` is a plain function and is directly invoked by the main thread. In the asynchronous supervisor, `slow` is a coroutine driven by `await`.
- There's no API to terminate a thread from the outside; instead, you must send a signal—like setting the `done` Event object. For tasks, there is the `Task.cancel()` instance method, which raises `CancelledError` at the `await` expression where the coroutine body is currently suspended.
- The supervisor coroutine must be started with `asyncio.run` in the main function.

This comparison should help you understand how concurrent jobs are orchestrated with *asyncio*, in contrast to how it's done with the `Threading` module, which may be more familiar to you.

One final point related to threads versus coroutines: if you've done any nontrivial programming with threads, you know how challenging it is to reason about the program because the scheduler can interrupt a thread at any time. You must remember to hold locks to protect the critical sections of your program, to avoid getting interrupted in the middle of a multistep operation—which could leave data in an invalid state.

With coroutines, your code is protected against interruption by default. You must explicitly `await` to let the rest of the program run. Instead of holding locks to synchronize the operations of multiple threads, coroutines are “synchronized” by definition: only one of them is running at any time. When you want to give up control, you use `await` to yield control back to the scheduler. That's why it is possible to safely cancel a coroutine: by definition, a coroutine can only be cancelled when it's suspended at an `await` expression, so you can perform cleanup by handling the `CancelledError` exception.

The `time.sleep()` call blocks but does nothing. Now we'll experiment with a CPU-intensive call to get a better understanding of the GIL, as well as the effect of CPU-intensive functions in asynchronous code.

The Real Impact of the GIL

In the threading code (Example 19-1), you can replace the `time.sleep(3)` call in the `slow` function with an HTTP client request from your favorite library, and the spinner will keep spinning. That's because a well-designed network library will release the GIL while waiting for the network.

You can also replace the `asyncio.sleep(3)` expression in the `slow` coroutine to `await` for a response from a well-designed asynchronous network library, because such libraries provide coroutines that yield control back to the event loop while waiting for the network. Meanwhile, the spinner will keep spinning.

With CPU-intensive code, the story is different. Consider the function `is_prime` in Example 19-10, which returns `True` if the argument is a prime number, `False` if it's not.

Example 19-10. `primes.py`: an easy to read primality check, from Python's `ProcessPoolExecutor` example

```
def is_prime(n: int) -> bool:
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
    return True
```

The call `is_prime(5_000_111_000_222_021)` takes about 3.3s on the company laptop I am using now.¹²

Quick Quiz

Given what we've seen so far, please take the time to consider the following three-part question. One part of the answer is tricky (at least it was for me).

What would happen to the spinner animation if you made the following changes, assuming that `n = 5_000_111_000_222_021`—that prime which my machine takes 3.3s to verify:

¹² It's a 15" MacBook Pro 2018 with a 6-core, 2.2 GHz Intel Core i7 CPU.

1. In *spinner_proc.py*, replace `time.sleep(3)` with a call to `is_prime(n)`?
2. In *spinner_thread.py*, replace `time.sleep(3)` with a call to `is_prime(n)`?
3. In *spinner_async.py*, replace `await asyncio.sleep(3)` with a call to `is_prime(n)`?

Before you run the code or read on, I recommend figuring out the answers on your own. Then, you may want to copy and modify the *spinner_*.py* examples as suggested.

Now the answers, from easier to hardest.

1. Answer for multiprocessing

The spinner is controlled by a child process, so it continues spinning while the primality test is computed by the parent process.¹³

2. Answer for threading

The spinner is controlled by a secondary thread, so it continues spinning while the primality test is computed by the main thread.

I did not get this answer right at first: I was expecting the spinner to freeze because I overestimated the impact of the GIL.

In this particular example, the spinner keeps spinning because Python suspends the running thread every 5ms (by default), making the GIL available to other pending threads. Therefore, the main thread running `is_prime` is interrupted every 5ms, allowing the secondary thread to wake up and iterate once through the for loop, until it calls the `wait` method of the done event, at which time it will release the GIL. The main thread will then grab the GIL, and the `is_prime` computation will proceed for another 5ms.

This does not have a visible impact on the running time of this specific example, because the `spin` function quickly iterates once and releases the GIL as it waits for the done event, so there is not much contention for the GIL. The main thread running `is_prime` will have the GIL most of the time.

We got away with a compute-intensive task using threading in this simple experiment because there are only two threads: one hogging the CPU, and the other waking up only 10 times per second to update the spinner.

¹³ This is true today because you are probably using a modern OS with *preemptive multitasking*. Windows before the NT era and macOS before the OSX era were not “preemptive,” therefore any process could take over 100% of the CPU and freeze the whole system. We are not completely free of this kind of problem today but trust this graybeard: this troubled every user in the 1990s, and a hard reset was the only cure.

But if you have two or more threads vying for a lot of CPU time, your program will be slower than sequential code.

3. Answer for asyncio

If you call `is_prime(5_000_111_000_222_021)` in the slow coroutine of the *spinner_async.py* example, the spinner will never appear. The effect would be the same we had in [Example 19-6](#), when we replaced `await asyncio.sleep(3)` with `time.sleep(3)`: no spinning at all. The flow of control will pass from supervisor to slow, and then to `is_prime`. When `is_prime` returns, slow returns as well, and supervisor resumes, cancelling the spinner task before it is executed even once. The program appears frozen for about 3s, then shows the answer.

Power Napping with `sleep(0)`

One way to keep the spinner alive is to rewrite `is_prime` as a coroutine, and periodically call `asyncio.sleep(0)` in an `await` expression to yield control back to the event loop, like in [Example 19-11](#).

Example 19-11. `spinner_async_nap.py`: `is_prime` is now a coroutine

```
async def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    root = math.isqrt(n)
    for i in range(3, root + 1, 2):
        if n % i == 0:
            return False
        if i % 100_000 == 1:
            await asyncio.sleep(0) ❶
    return True
```

❶ Sleep once every 50,000 iterations (because the step in the range is 2).

[Issue #284](#) in the asyncio repository has an informative discussion about the use of `asyncio.sleep(0)`.

However, be aware this will slow down `is_prime`, and—more importantly—will still slow down the event loop and your whole program with it. When I used `await asyncio.sleep(0)` every 100,000 iterations, the spinner was smooth but the program ran in 4.9s on my machine, almost 50% longer than the original `primes.is_prime` function by itself with the same argument (5_000_111_000_222_021).

Using `await asyncio.sleep(0)` should be considered a stopgap measure before you refactor your asynchronous code to delegate CPU-intensive computations to another process. We'll see one way of doing that with `asyncio.loop.run_in_executor`, covered in [Chapter 21](#). Another option would be a task queue, which we'll briefly discuss in [“Distributed Task Queues” on page 732](#).

So far, we've only experimented with a single call to a CPU-intensive function. The next section presents concurrent execution of multiple CPU-intensive calls.

A Homegrown Process Pool



I wrote this section to show the use of multiple processes for CPU-intensive tasks, and the common pattern of using queues to distribute tasks and collect results. [Chapter 20](#) will show a simpler way of distributing tasks to processes: a `ProcessPoolExecutor` from the `concurrent.futures` package, which uses queues internally.

In this section we'll write programs to compute the primality of a sample of 20 integers, from 2 to 9,999,999,999,999,999—i.e., $10^{16} - 1$, or more than 2^{53} . The sample includes small and large primes, as well as composite numbers with small and large prime factors.

The *sequential.py* program provides the performance baseline. Here is a sample run:

```
$ python3 sequential.py
      2 P 0.000001s
142702110479723 P 0.568328s
299593572317531 P 0.796773s
3333333333333301 P 2.648625s
3333333333333333 0.000007s
3333335652092209 2.672323s
4444444444444423 P 3.052667s
4444444444444444 0.000001s
4444444488888889 3.061083s
5555553133149889 3.451833s
5555555555555503 P 3.556867s
5555555555555555 0.000007s
6666666666666666 0.000001s
6666666666666719 P 3.781064s
6666667141414921 3.778166s
7777777536340681 4.120069s
7777777777777753 P 4.141530s
7777777777777777 0.000007s
9999999999999917 P 4.678164s
9999999999999999 0.000007s
Total time: 40.31
```

The results are shown in three columns:

- The number to be checked.
- P if it's a prime number, blank if not.
- Elapsed time for checking the primality for that specific number.

In this example, the total time is approximately the sum of the times for each check, but it is computed separately, as you can see in [Example 19-12](#).

Example 19-12. sequential.py: sequential primality check for a small dataset

```
#!/usr/bin/env python3

"""
sequential.py: baseline for comparing sequential, multiprocessing,
and threading code for CPU-intensive work.
"""

from time import perf_counter
from typing import NamedTuple

from primes import is_prime, NUMBERS

class Result(NamedTuple): ❶
    prime: bool
    elapsed: float

def check(n: int) -> Result: ❷
    t0 = perf_counter()
    prime = is_prime(n)
    return Result(prime, perf_counter() - t0)

def main() -> None:
    print(f'Checking {len(NUMBERS)} numbers sequentially:')
    t0 = perf_counter()
    for n in NUMBERS: ❸
        prime, elapsed = check(n)
        label = 'P' if prime else ' '
        print(f'{n:16} {label} {elapsed:9.6f}s')

    elapsed = perf_counter() - t0 ❹
    print(f'Total time: {elapsed:.2f}s')

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

- ❶ The `check` function (in the next callout) returns a `Result` tuple with the boolean value of the `is_prime` call and the elapsed time.
- ❷ `check(n)` calls `is_prime(n)` and computes the elapsed time to return a `Result`.
- ❸ For each number in the sample, we call `check` and display the result.
- ❹ Compute and display the total elapsed time.

Process-Based Solution

The next example, *procs.py*, shows the use of multiple processes to distribute the primality checks across multiple CPU cores. These are the times I get with *procs.py*:

```
$ python3 procs.py
Checking 20 numbers with 12 processes:
      2 P 0.000002s
333333333333333333 0.000021s
444444444444444444 0.000002s
555555555555555555 0.000018s
666666666666666666 0.000002s
 142702110479723 P 1.350982s
777777777777777777 0.000009s
 299593572317531 P 1.981411s
999999999999999999 0.000008s
333333333333333301 P 6.328173s
3333335652092209 6.419249s
4444444488888889 7.051267s
44444444444444423 P 7.122004s
5555553133149889 7.412735s
55555555555555503 P 7.603327s
66666666666666719 P 7.934670s
6666667141414921 8.017599s
777777536340681 8.339623s
7777777777777753 P 8.38859s
9999999999999917 P 8.117313s
20 checks in 9.58s
```

The last line of the output shows that *procs.py* was 4.2 times faster than *sequential.py*.

Understanding the Elapsed Times

Note that the elapsed time in the first column is for checking that specific number. For example, `is_prime(7777777777777753)` took almost 8.4s to return `True`. Meanwhile, other processes were checking other numbers in parallel.

There were 20 numbers to check. I wrote *procs.py* to start a number of worker processes equal to the number of CPU cores, as determined by `multiprocessing.cpu_count()`.

The total time in this case is much less than the sum of the elapsed time for the individual checks. There is some overhead in spinning up processes and in inter-process communication, so the end result is that the multiprocessing version is only about 4.2 times faster than the sequential. That’s good, but a little disappointing considering the code launches 12 processes to use all cores on this laptop.



The `multiprocessing.cpu_count()` function returns 12 on the MacBook Pro I’m using to write this chapter. It’s actually a 6-CPU Core-i7, but the OS reports 12 CPUs because of hyperthreading, an Intel technology which executes 2 threads per core. However, hyperthreading works better when one of the threads is not working as hard as the other thread in the same core—perhaps the first is stalled waiting for data after a cache miss, and the other is crunching numbers. Anyway, there’s no free lunch: this laptop performs like a 6-CPU machine for compute-intensive work that doesn’t use a lot of memory—like that simple primality test.

Code for the Multicore Prime Checker

When we delegate computing to threads or processes, our code does not call the worker function directly, so we can’t simply get a return value. Instead, the worker is driven by the thread or process library, and it eventually produces a result that needs to be stored somewhere. Coordinating workers and collecting results are common uses of queues in concurrent programming—and also in distributed systems.

Much of the new code in *procs.py* has to do with setting up and using queues. The top of the file is in [Example 19-13](#).



`SimpleQueue` was added to `multiprocessing` in Python 3.9. If you’re using an earlier version of Python, you can replace `SimpleQueue` with `Queue` in [Example 19-13](#).

Example 19-13. procs.py: multiprocess primality check; imports, types, and functions

```
import sys
from time import perf_counter
from typing import NamedTuple
from multiprocessing import Process, SimpleQueue, cpu_count ❶
from multiprocessing import queues ❷

from primes import is_prime, NUMBERS

class PrimeResult(NamedTuple): ❸
    n: int
```

```

    prime: bool
    elapsed: float

JobQueue = queues.SimpleQueue[int] ❷
ResultQueue = queues.SimpleQueue[PrimeResult] ❸

def check(n: int) -> PrimeResult: ❹
    t0 = perf_counter()
    res = is_prime(n)
    return PrimeResult(n, res, perf_counter() - t0)

def worker(jobs: JobQueue, results: ResultQueue) -> None: ❺
    while n := jobs.get(): ❻
        results.put(check(n)) ❼
        results.put(PrimeResult(0, False, 0.0)) ❽

def start_jobs(
    procs: int, jobs: JobQueue, results: ResultQueue ❾
) -> None:
    for n in NUMBERS:
        jobs.put(n) ❿
    for _ in range(procs):
        proc = Process(target=worker, args=(jobs, results)) ❾
        proc.start() ❿
        jobs.put(0) ⓫

```

- ❶ Trying to emulate threading, multiprocessing provides multiprocessing.SimpleQueue, but this is a method bound to a predefined instance of a lower-level BaseContext class. We must call this SimpleQueue to build a queue, we can't use it in type hints.
- ❷ multiprocessing.queues has the SimpleQueue class we need for type hints.
- ❸ PrimeResult includes the number checked for primality. Keeping n together with the other result fields simplifies displaying results later.
- ❹ This is a type alias for a SimpleQueue that the main function (Example 19-14) will use to send numbers to the processes that will do the work.
- ❺ Type alias for a second SimpleQueue that will collect the results in main. The values in the queue will be tuples made of the number to be tested for primality, and a Result tuple.
- ❻ This is similar to *sequential.py*.
- ❼ worker gets a queue with the numbers to be checked, and another to put results.

- ❸ In this code, I use the number 0 as a *poison pill*: a signal for the worker to finish. If `n` is not 0, proceed with the loop.¹⁴
- ❹ Invoke the primality check and enqueue `PrimeResult`.
- ❺ Send back a `PrimeResult(0, False, 0.0)` to let the main loop know that this worker is done.
- ❻ `procs` is the number of processes that will compute the prime checks in parallel.
- ❼ Enqueue the numbers to be checked in `jobs`.
- ❽ Fork a child process for each worker. Each child will run the loop inside its own instance of the `worker` function, until it fetches a 0 from the `jobs` queue.
- ❾ Start each child process.
- ❿ Enqueue one 0 for each process, to terminate them.

Loops, Sentinels, and Poison Pills

The worker function in [Example 19-13](#) follows a common pattern in concurrent programming: looping indefinitely while taking items from a queue and processing each with a function that does the actual work. The loop ends when the queue produces a sentinel value. In this pattern, the sentinel that shuts down the worker is often called a “poison pill.”

`None` is often used as a sentinel value, but it may be unsuitable if it can occur in the data stream. Calling `object()` is a common way to get a unique value to use as sentinel. However, that does not work across processes because Python objects must be serialized for inter-process communication, and when you `pickle.dump` and `pickle.load` an instance of `object`, the unpickled instance is distinct from the original: it doesn’t compare equal. A good alternative to `None` is the `Ellipsis` built-in object (a.k.a. `...`), which survives serialization without losing its identity.¹⁵

Python’s standard library uses [lots of different values](#) as sentinels. [PEP 661—Sentinel Values](#) proposes a standard sentinel type. As of September 2021, it’s only a draft.

¹⁴ In this example, 0 is a convenient sentinel. `None` is also commonly used for that. Using 0 simplifies the type hint for `PrimeResult` and the code for `worker`.

¹⁵ Surviving serialization without losing our identity is a pretty good life goal.

Now let's study the main function of *procs.py* in [Example 19-14](#).

Example 19-14. procs.py: multiprocessing primality check; main function

```
def main() -> None:
    if len(sys.argv) < 2: ❶
        procs = cpu_count()
    else:
        procs = int(sys.argv[1])

    print(f'Checking {len(NUMBERS)} numbers with {procs} processes:')
    t0 = perf_counter()
    jobs: JobQueue = SimpleQueue() ❷
    results: ResultQueue = SimpleQueue()
    start_jobs(procs, jobs, results) ❸
    checked = report(procs, results) ❹
    elapsed = perf_counter() - t0
    print(f'{checked} checks in {elapsed:.2f}s') ❺

def report(procs: int, results: ResultQueue) -> int: ❻
    checked = 0
    procs_done = 0
    while procs_done < procs: ❼
        n, prime, elapsed = results.get() ❽
        if n == 0: ❾
            procs_done += 1
        else:
            checked += 1 ❿
            label = 'P' if prime else ' '
            print(f'{n:16} {label} {elapsed:9.6f}s')
    return checked

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

- ❶ If no command-line argument is given, set the number of processes to the number of CPU cores; otherwise, create as many processes as given in the first argument.
- ❷ jobs and results are the queues described in [Example 19-13](#).
- ❸ Start proc processes to consume jobs and post results.
- ❹ Retrieve the results and display them; report is defined in ❻.
- ❺ Display how many numbers were checked and the total elapsed time.
- ❻ The arguments are the number of procs and the queue to post the results.

- ⑦ Loop until all processes are done.
- ⑧ Get one `PrimeResult`. Calling `.get()` on a queue block until there is an item in the queue. It's also possible to make this nonblocking, or set a timeout. See the [SimpleQueue.get](#) documentation for details.
- ⑨ If `n` is zero, then one process exited; increment the `procs_done` count.
- ⑩ Otherwise, increment the checked count (to keep track of the numbers checked) and display the results.

The results will not come back in the same order the jobs were submitted. That's why I had to put `n` in each `PrimeResult` tuple. Otherwise, I'd have no way to know which result belonged to each number.

If the main process exits before all subprocesses are done, you may see confusing tracebacks on `FileNotFoundError` exceptions caused by an internal lock in `multi` processing. Debugging concurrent code is always hard, and debugging `multiproc` essing is even harder because of all the complexity behind the thread-like façade. Fortunately, the `ProcessPoolExecutor` we'll meet in [Chapter 20](#) is easier to use and more robust.



Thanks to reader Michael Albert who noticed the code I published during the early release had a *race condition* in [Example 19-14](#). A race condition is a bug that may or may not occur depending on the order of actions performed by concurrent execution units. If “A” happens before “B,” all is fine; but if “B” happens first, something goes wrong. That's the race.

If you are curious, this diff shows the bug and how I fixed it: [example-code-2e/commit/2c123057](#)—but note that I later refactored the example to delegate parts of `main` to the `start_jobs` and `report` functions. There's a [README.md](#) file in the same directory explaining the problem and the solution.

Experimenting with More or Fewer Processes

You may want try running `procs.py`, passing arguments to set the number of worker processes. For example, this command...

```
$ python3 procs.py 2
```

...will launch two worker processes, producing results almost twice as fast as *sequential.py*—if your machine has at least two cores and is not too busy running other programs.

I ran *procs.py* 12 times with 1 to 20 processes, totaling 240 runs. Then I computed the median time for all runs with the same number of processes, and plotted [Figure 19-2](#).

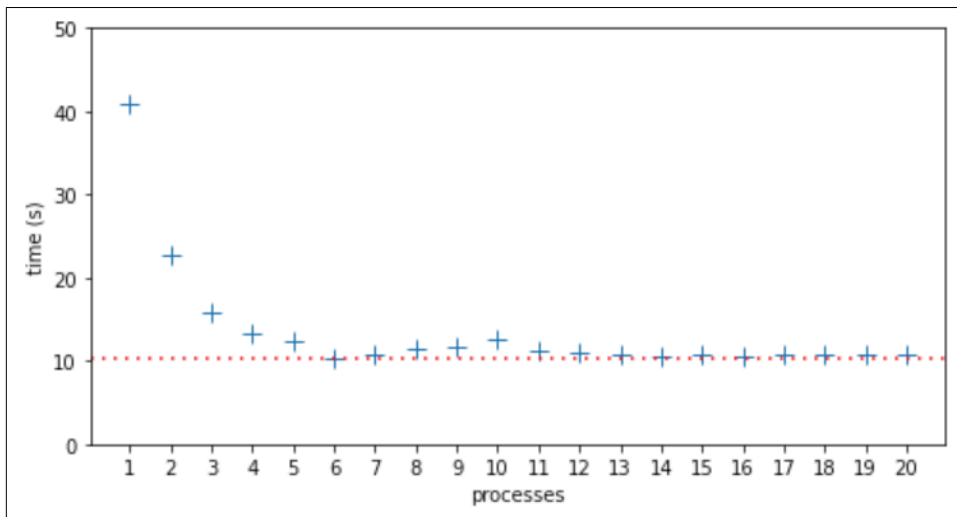


Figure 19-2. Median run times for each number of processes from 1 to 20. Highest median time was 40.81s, with 1 process. Lowest median time was 10.39s, with 6 processes, indicated by the dotted line.

In this 6-core laptop, the lowest median time was with 6 processes: 10.39s—marked by the dotted line in [Figure 19-2](#). I expected the run time to increase after 6 processes due to CPU contention, and it reached a local maximum of 12.51s at 10 processes. I did not expect and I can’t explain why the performance improved at 11 processes and remained almost flat from 13 to 20 processes, with median times only slightly higher than the lowest median time at 6 processes.

Thread-Based Nonsolution

I also wrote *threads.py*, a version of *procs.py* using threading instead of multiprocessing. The code is very similar—as is usually the case when converting simple examples between these two APIs.¹⁶ Due to the GIL and the compute-intensive nature of *is_prime*, the threaded version is slower than the sequential code in [Example 19-12](#), and it gets slower as the number of threads increase, because of CPU contention and the cost of context switching. To switch to a new thread, the OS needs to save CPU registers and update the program counter and stack pointer,

¹⁶ See [19-concurrency/primes/threads.py](#) in the *Fluent Python* code repository.

triggering expensive side effects like invalidating CPU caches and possibly even swapping memory pages.¹⁷

The next two chapters will cover more about concurrent programming in Python, using the high-level *concurrent.futures* library to manage threads and processes ([Chapter 20](#)) and the *asyncio* library for asynchronous programming ([Chapter 21](#)).

The remaining sections in this chapter aim to answer the question:

Given the limitations discussed so far, how is Python thriving in a multicore world?

Python in the Multicore World

Consider this citation from the widely quoted article “[The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software](#)” by Herb Sutter:

The major processor manufacturers and architectures, from Intel and AMD to Sparc and PowerPC, have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds and straight-line instruction throughput ever higher, they are instead turning en masse to hyper-threading and multicore architectures. March 2005. [Available online].

What Sutter calls the “free lunch” was the trend of software getting faster with no additional developer effort because CPUs were executing sequential code faster, year after year. Since 2004, that is no longer true: clock speeds and execution optimizations reached a plateau, and now any significant increase in performance must come from leveraging multiple cores or hyperthreading, advances that only benefit code that is written for concurrent execution.

Python’s story started in the early 1990s, when CPUs were still getting exponentially faster at sequential code execution. There was no talk about multicore CPUs except in supercomputers back then. At the time, the decision to have a GIL was a no-brainer. The GIL makes the interpreter faster when running on a single core, and its implementation simpler.¹⁸ The GIL also makes it easier to write simple extensions through the Python/C API.

¹⁷ To learn more, see “[Context switch](#)” in the English Wikipedia.

¹⁸ These are probably the same reasons that prompted the creator of the Ruby language, Yukihiro Matsumoto, to use a GIL in his interpreter as well.



I just wrote “simple extensions” because an extension does not need to deal with the GIL at all. A function written in C or Fortran may be hundreds of times faster than the equivalent in Python.¹⁹ Therefore the added complexity of releasing the GIL to leverage multicore CPUs may not be needed in many cases. So we can thank the GIL for many extensions available for Python—and that is certainly one of the key reasons why the language is so popular today.

Despite the GIL, Python is thriving in applications that require concurrent or parallel execution, thanks to libraries and software architectures that work around the limitations of CPython.

Now let’s discuss how Python is used in system administration, data science, and server-side application development in the multicore, distributed computing world of 2021.

System Administration

Python is widely used to manage large fleets of servers, routers, load balancers, and network-attached storage (NAS). It’s also a leading option in software-defined networking (SDN) and ethical hacking. Major cloud service providers support Python through libraries and tutorials authored by the providers themselves or by their large communities of Python users.

In this domain, Python scripts automate configuration tasks by issuing commands to be carried out by the remote machines, so rarely there are CPU-bound operations to be done. Threads or coroutines are well suited for such jobs. In particular, the concurrent.futures package we’ll see in [Chapter 20](#) can be used to perform the same operations on many remote machines at the same time without a lot of complexity.

Beyond the standard library, there are popular Python-based projects to manage server clusters: tools like *Ansible* and *Salt*, as well as libraries like *Fabric*.

There is also a growing number of libraries for system administration supporting coroutines and `asyncio`. In 2016, Facebook’s [Production Engineering team reported](#): “We are increasingly relying on `AsyncIO`, which was introduced in Python 3.4, and seeing huge performance gains as we move codebases away from Python 2.”

¹⁹ As an exercise in college, I had to implement the LZW compression algorithm in C. But first I wrote it in Python, to check my understanding of the spec. The C version was about 900× faster.

Data Science

Data science—including artificial intelligence—and scientific computing are very well served by Python. Applications in these fields are compute-intensive, but Python users benefit from a vast ecosystem of numeric computing libraries written in C, C++, Fortran, Cython, etc.—many of which are able to leverage multicore machines, GPUs, and/or distributed parallel computing in heterogeneous clusters.

As of 2021, Python’s data science ecosystem includes impressive tools such as:

Project Jupyter

Two browser-based interfaces—Jupyter Notebook and JupyterLab—that allow users to run and document analytics code potentially running across the network on remote machines. Both are hybrid Python/JavaScript applications, supporting computing kernels written in different languages, all integrated via ZeroMQ—an asynchronous messaging library for distributed applications. The name *Jupyter* actually comes from Julia, Python, and R, the first three languages supported by the Notebook. The rich ecosystem built on top of the Jupyter tools include **Bokeh**, a powerful interactive visualization library that lets users navigate and interact with large datasets or continuously updated streaming data, thanks to the performance of modern JavaScript engines and browsers.

TensorFlow and PyTorch

These are the top two deep learning frameworks, according to **O’Reilly’s January 2021 report** on usage of their learning resources during 2020. Both projects are written in C++, and are able to leverage multiple cores, GPUs, and clusters. They support other languages as well, but Python is their main focus and is used by the majority of their users. TensorFlow was created and is used internally by Google; PyTorch by Facebook.

Dask

A parallel computing library that can farm out work to local processes or clusters of machines, “tested on some of the largest supercomputers in the world”—as their **home page** states. Dask offers APIs that closely emulate NumPy, pandas, and scikit-learn—the most popular libraries in data science and machine learning today. Dask can be used from JupyterLab or Jupyter Notebook, and leverages Bokeh not only for data visualization but also for an interactive dashboard showing the flow of data and computations across the processes/machines in near real time. Dask is so impressive that I recommend watching a video such as this **15-minute demo** in which Matthew Rocklin—a maintainer of the project—shows Dask crunching data on 64 cores distributed in 8 EC2 machines on AWS.

These are only some examples to illustrate how the data science community is creating solutions that leverage the best of Python and overcome the limitations of the CPython runtime.

Server-Side Web/Mobile Development

Python is widely used in web applications and for the backend APIs supporting mobile applications. How is it that Google, YouTube, Dropbox, Instagram, Quora, and Reddit—among others—managed to build Python server-side applications serving hundreds of millions of users 24x7? Again, the answer goes way beyond what Python provides “out of the box.”

Before we discuss tools to support Python at scale, I must quote an admonition from the Thoughtworks *Technology Radar*:

High performance envy/web scale envy

We see many teams run into trouble because they have chosen complex tools, frameworks or architectures because they “might need to scale.” Companies such as Twitter and Netflix need to support extreme loads and so need these architectures, but they also have extremely skilled development teams able to handle the complexity. Most situations do not require these kinds of engineering feats; teams should keep their *web scale envy* in check in favor of simpler solutions that still get the job done.²⁰

At *web scale*, the key is an architecture that allows horizontal scaling. At that point, all systems are distributed systems, and no single programming language is likely to be the right choice for every part of the solution.

Distributed systems is a field of academic research, but fortunately some practitioners have written accessible books anchored on solid research and practical experience. One of them is Martin Kleppmann, the author of *Designing Data-Intensive Applications* (O’Reilly).

Consider [Figure 19-3](#), the first of many architecture diagrams in Kleppmann’s book. Here are some components I’ve seen in Python engagements that I worked on or have firsthand knowledge of:

- Application caches:²¹ *memcached*, *Redis*, *Varnish*
- Relational databases: *PostgreSQL*, *MySQL*
- Document databases: *Apache CouchDB*, *MongoDB*
- Full-text indexes: *Elasticsearch*, *Apache Solr*
- Message queues: *RabbitMQ*, *Redis*

20 Source: Thoughtworks Technology Advisory Board, *Technology Radar—November 2015*.

21 Contrast application caches—used directly by your application code—with HTTP caches, which would be placed on the top edge of [Figure 19-3](#) to serve static assets like images, CSS, and JS files. Content Delivery Networks (CDNs) offer another type of HTTP cache, deployed in data centers closer to the end users of your application.

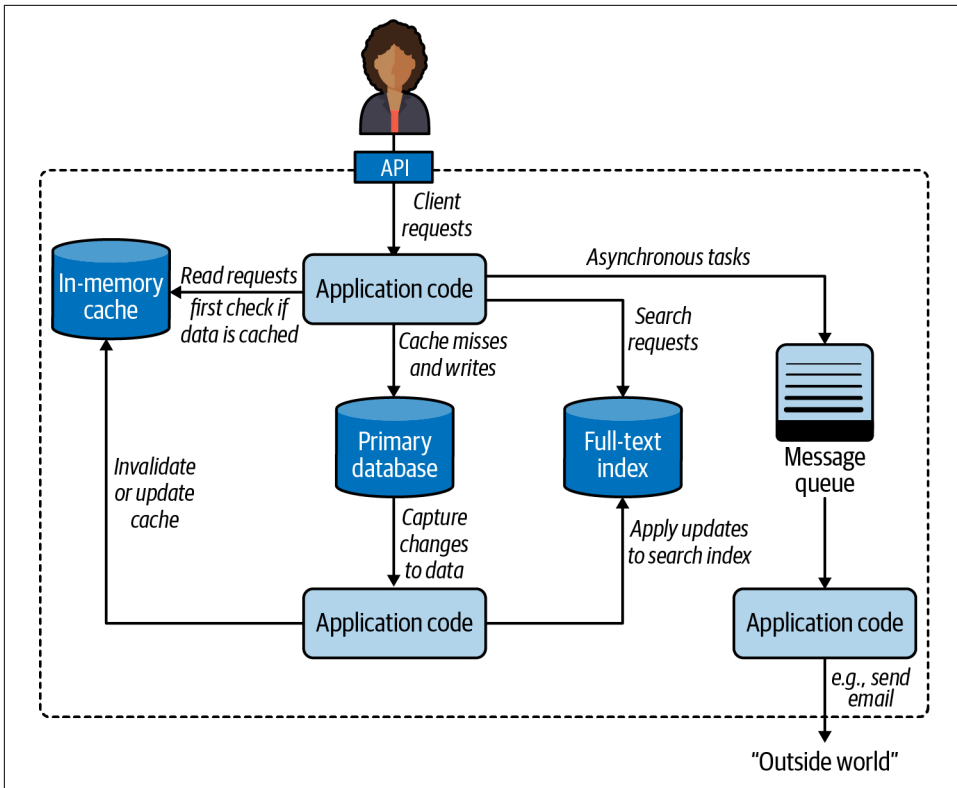


Figure 19-3. One possible architecture for a system combining several components.²²

There are other industrial-strength open source products in each of those categories. Major cloud providers also offer their own proprietary alternatives.

Kleppmann’s diagram is general and language independent—as is his book. For Python server-side applications, two specific components are often deployed:

- An application server to distribute the load among several instances of the Python application. The application server would appear near the top in Figure 19-3, handling client requests before they reached the application code.
- A task queue built around the message queue on the righthand side of Figure 19-3, providing a higher-level, easier-to-use API to distribute tasks to processes running on other machines.

²² Diagram adapted from Figure 1-1, *Designing Data-Intensive Applications* by Martin Kleppmann (O’Reilly).

The next two sections explore these components that are recommended best practices in Python server-side deployments.

WSGI Application Servers

WSGI—the **Web Server Gateway Interface**—is a standard API for a Python framework or application to receive requests from an HTTP server and send responses to it.²³ WSGI application servers manage one or more processes running your application, maximizing the use of the available CPUs.

Figure 19-4 illustrates a typical WSGI deployment.



If we wanted to merge the previous pair of diagrams, the content of the dashed rectangle in Figure 19-4 would replace the solid “Application code” rectangle at the top of Figure 19-3.

The best-known application servers in Python web projects are:

- *mod_wsgi*
- *uWSGI*²⁴
- *Gunicorn*
- *NGINX Unit*

For users of the Apache HTTP server, *mod_wsgi* is the best option. It’s as old as WSGI itself, but is actively maintained, and now provides a command-line launcher called *mod_wsgi-express* that makes it easier to configure and more suitable for use in Docker containers.

²³ Some speakers spell out the WSGI acronym, while others pronounce it as one word rhyming with “whisky.”

²⁴ *uWSGI* is spelled with a lowercase “u,” but that is pronounced as the Greek letter “μ,” so the whole name sounds like “micro-whisky” with a “g” instead of the “k.”

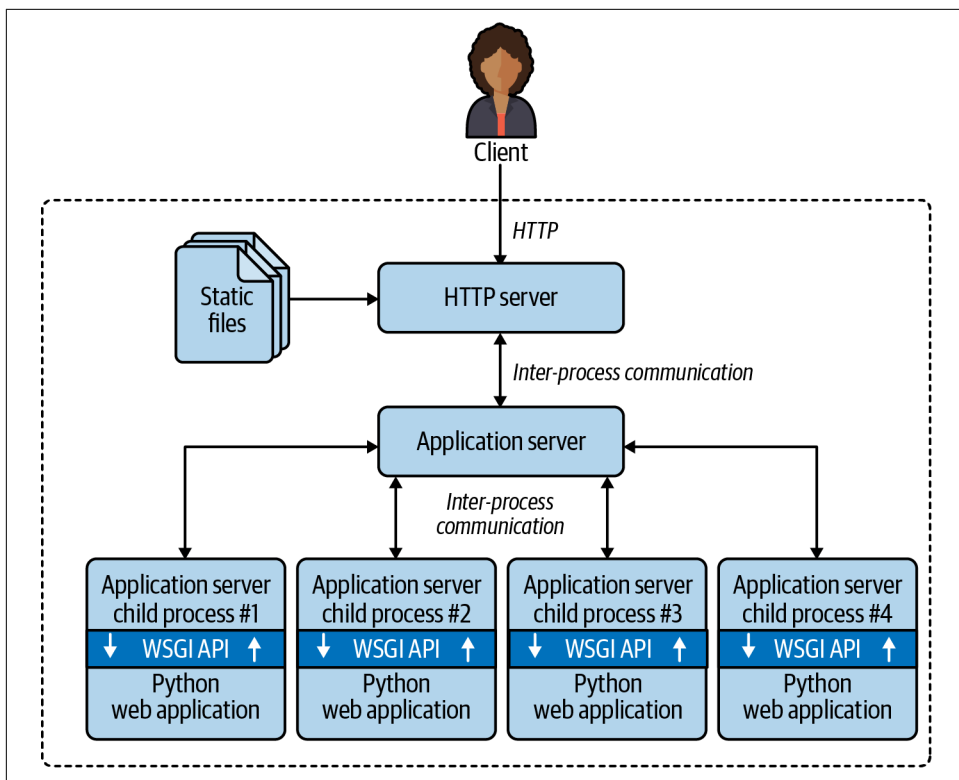


Figure 19-4. Clients connect to an HTTP server that delivers static files and routes other requests to the application server, which forks child processes to run the application code, leveraging multiple CPU cores. The WSGI API is the glue between the application server and the Python application code.

uWSGI and *Gunicorn* are the top choices in recent projects I know about. Both are often used with the *NGINX* HTTP server. *uWSGI* offers a lot of extra functionality, including an application cache, a task queue, cron-like periodic tasks, and many other features. On the flip side, *uWSGI* is much harder to configure properly than *Gunicorn*.²⁵

Released in 2018, *NGINX Unit* is a new product from the makers of the well-known *NGINX* HTTP server and reverse proxy.

²⁵ Bloomberg engineers Peter Sperl and Ben Green wrote “[Configuring uWSGI for Production Deployment](#)”, explaining how many of the default settings in *uWSGI* are not suitable for many common deployment scenarios. Sperl presented a summary of their recommendations at [EuroPython 2019](#). Highly recommended for users of *uWSGI*.

mod_wsgi and *Gunicorn* support Python web apps only, while *uWSGI* and *NGINX Unit* work with other languages as well. Please browse their docs to learn more.

The main point: all of these application servers can potentially use all CPU cores on the server by forking multiple Python processes to run traditional web apps written in good old sequential code in *Django*, *Flask*, *Pyramid*, etc. This explains why it's been possible to earn a living as a Python web developer without ever studying the threading, multiprocessing, or *asyncio* modules: the application server handles concurrency transparently.



ASGI—Asynchronous Server Gateway Interface

WSGI is a synchronous API. It doesn't support coroutines with *async/await*—the most efficient way to implement WebSockets or HTTP long polling in Python. The **ASGI specification** is a successor to WSGI, designed for asynchronous Python web frameworks such as *aiohttp*, *Sanic*, *FastAPI*, etc., as well as *Django* and *Flask*, which are gradually adding asynchronous functionality.

Now let's turn to another way of bypassing the GIL to achieve higher performance with server-side Python applications.

Distributed Task Queues

When the application server delivers a request to one of the Python processes running your code, your app needs to respond quickly: you want the process to be available to handle the next request as soon as possible. However, some requests demand actions that may take longer—for example, sending email or generating a PDF. That's the problem that distributed task queues are designed to solve.

Celery and *RQ* are the best known open source task queues with Python APIs. Cloud providers also offer their own proprietary task queues.

These products wrap a message queue and offer a high-level API for delegating tasks to workers, possibly running on different machines.



In the context of task queues, the words *producer* and *consumer* are used instead of traditional client/server terminology. For example, a *Django* view handler *produces* job requests, which are put in the queue to be *consumed* by one or more PDF rendering processes.

Quoting directly from *Celery*'s **FAQ**, here are some typical use cases:

- Running something in the background. For example, to finish the web request as soon as possible, then update the users page incrementally. This gives the user the

impression of good performance and “snappiness,” even though the real work might actually take some time.

- Running something after the web request has finished.
- Making sure something is done, by executing it asynchronously and using retries.
- Scheduling periodic work.

Besides solving these immediate problems, task queues support horizontal scalability. Producers and consumers are decoupled: a producer doesn’t call a consumer, it puts a request in a queue. Consumers don’t need to know anything about the producers (but the request may include information about the producer, if an acknowledgment is required). Crucially, you can easily add more workers to consume tasks as demand grows. That’s why *Celery* and *RQ* are called distributed task queues.

Recall that our simple *procs.py* (Example 19-13) used two queues: one for job requests, the other for collecting results. The distributed architecture of *Celery* and *RQ* uses a similar pattern. Both support using the *Redis* NoSQL database as a message queue and result storage. *Celery* also supports other message queues like *RabbitMQ* or *Amazon SQS*, as well other databases for result storage.

This wraps up our introduction to concurrency in Python. The next two chapters will continue this theme, focusing on the `concurrent.futures` and `asyncio` packages of the standard library.

Chapter Summary

After a bit of theory, this chapter presented the spinner scripts implemented in each of Python’s three native concurrency programming models:

- Threads, with the `threading` package
- Processes, with `multiprocessing`
- Asynchronous coroutines with `asyncio`

We then explored the real impact of the GIL with an experiment: changing the spinner examples to compute the primality of a large integer and observe the resulting behavior. This demonstrated graphically that CPU-intensive functions must be avoided in `asyncio`, as they block the event loop. The threaded version of the experiment worked—despite the GIL—because Python periodically interrupts threads, and the example used only two threads: one doing compute-intensive work, and the other driving the animation only 10 times per second. The `multiprocessing` variant worked around the GIL, starting a new process just for the animation, while the main process did the primality check.