CHAPTER 17

Concurrency with Futures

The people bashing threads are typically system programmers which have in mind use cases that the typical application programmer will never encounter in her life. […] In 99% of the use cases an application programmer is likely to run into, the simple pattern of spawning a bunch of independent threads and collecting the results in a queue is everything one needs to know.[1]

— Michele Simionato
Python deep thinker

This chapter focuses on the `concurrent.futures` library introduced in Python 3.2, but also available for Python 2.5 and newer as the `futures` package on PyPI. This library encapsulates the pattern described by Michele Simionato in the preceding quote, making it almost trivial to use.

Here I also introduce the concept of "futures"—objects representing the asynchronous execution of an operation. This powerful idea is the foundation not only of `concurrent.futures` but also of the `asyncio` package, which we'll cover in Chapter 18.

We'll start with a motivating example.

Example: Web Downloads in Three Styles

To handle network I/O efficiently, you need concurrency, as it involves high latency—so instead of wasting CPU cycles waiting, it's better to do something else until a response comes back from the network.

To make this last point with code, I wrote three simple programs to download images of 20 country flags from the Web. The first one, *flags.py*, runs sequentially: it only re-

1. From Michele Simionato's post Threads, processes and concurrency in Python: some thoughts, subtitled "Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency."

quests the next image when the previous one is downloaded and saved to disk. The other two scripts make concurrent downloads: they request all images practically at the same time, and save the files as they arrive. The *flags_threadpool.py* script uses the `concurrent.futures` package, while *flags_asyncio.py* uses `asyncio`.

Example 17-1 shows the result of running the three scripts, three times each. I also posted a 73s video on YouTube so you can watch them running while an OS X Finder window displays the flags as they are saved. The scripts are downloading images from *flupy.org*, which is behind a CDN, so you may see slower results in the first runs. The results in Example 17-1 were obtained after several runs, so the CDN cache was warm.

Example 17-1. Three typical runs of the scripts flags.py, flags_threadpool.py, and flags_asyncio.py

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN  ❶
20 flags downloaded in 7.26s  ❷
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s  ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py  ❹
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR  ❺
20 flags downloaded in 1.42s
```

❶ The output for each run starts with the country codes of the flags as they are downloaded, and ends with a message stating the elapsed time.

❷ It took *flags.py* an average 7.18s to download 20 images.

❸ The average for *flags_threadpool.py* was 1.40s.

❹ For *flags_asyncio.py*, 1.35 was the average time.

❺ Note the order of the country codes: the downloads happened in a different order every time with the concurrent scripts.

The difference in performance between the concurrent scripts is not significant, but they are both more than five times faster than the sequential script—and this is just for a fairly small task. If you scale the task to hundreds of downloads, the concurrent scripts can outpace the sequential one by a factor or 20 or more.

While testing concurrent HTTP clients on the public Web you may inadvertently launch a denial-of-service (DoS) attack, or be suspected of doing so. In the case of Example 17-1, it's OK to do it because those scripts are hardcoded to make only 20 requests. For testing nontrivial HTTP clients, you should set up your own test server. The *17-futures/countries/README.rst* file in the *Fluent Python* code GitHub repository has instructions for setting a local Nginx server.

Now let's study the implementations of two of the scripts tested in Example 17-1: *flags.py* and *flags_threadpool.py*. I will leave the third script, *flags_asyncio.py*, for Chapter 18, but I wanted to demonstrate all three together to make a point: regardless of the concurrency strategy you use—threads or `asyncio`—you'll see vastly improved throughput over sequential code in I/O-bound applications, if you code it properly.

On to the code.

A Sequential Download Script

Example 17-2 is not very interesting, but we'll reuse most of its code and settings to implement the concurrent scripts, so it deserves some attention.

For clarity, there is no error handling in Example 17-2. We will deal with exceptions later, but here we want to focus on the basic structure of the code, to make it easier to contrast this script with the concurrent ones.

Example 17-2. flags.py: sequential download script; some functions will be reused by the other scripts

```
import os
import time
import sys

import requests    ❶
```

```
POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()   ❷

BASE_URL = 'http://flupy.org/data/flags'   ❸

DEST_DIR = 'downloads/'   ❹


def save_flag(img, filename):   ❺
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)


def get_flag(cc):   ❻
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = requests.get(url)
    return resp.content


def show(text):   ❼
    print(text, end=' ')
    sys.stdout.flush()


def download_many(cc_list):   ❽
    for cc in sorted(cc_list):   ❾
        image = get_flag(cc)
        show(cc)
        save_flag(image, cc.lower() + '.gif')

    return len(cc_list)


def main(download_many):   ❿
    t0 = time.time()
    count = download_many(POP20_CC)
    elapsed = time.time() - t0
    msg = '\n{} flags downloaded in {:.2f}s'
    print(msg.format(count, elapsed))


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

❶ Import the `requests` library; it's not part of the standard library, so by convention we import it after the standard library modules `os`, `time`, and `sys`, and separate it from them with a blank line.

❷ List of the ISO 3166 country codes for the 20 most populous countries in order of decreasing population.

❸ The website with the flag images.[2]

❹ Local directory where the images are saved.

❺ Simply save the `img` (a byte sequence) to `filename` in the `DEST_DIR`.

❻ Given a country code, build the URL and download the image, returning the binary contents of the response.

❼ Display a string and flush `sys.stdout` so we can see progress in a one-line display; this is needed because Python normally waits for a line break to flush the `stdout` buffer.

❽ `download_many` is the key function to compare with the concurrent implementations.

❾ Loop over the list of country codes in alphabetical order, to make it clear that the ordering is preserved in the output; return the number of country codes downloaded.

❿ `main` records and reports the elapsed time after running `download_many`.

⓫ `main` must be called with the function that will make the downloads; we pass the `download_many` function as an argument so that `main` can be used as a library function with other implementations of `download_many` in the next examples.

The `requests` library by Kenneth Reitz is available on PyPI and is more powerful and easier to use than the `urllib.request` module from the Python 3 standard library. In fact, `requests` is considered a model Pythonic API. It is also compatible with Python 2.6 and up, while the `urllib2` from Python 2 was moved and renamed in Python 3, so it's more convenient to use `requests` regardless of the Python version you're targeting.

There's really nothing new to *flags.py*. It serves as a baseline for comparing the other scripts and I used it as a library to avoid redundant code when implementing them. Now let's see a reimplementation using `concurrent.futures`.

Downloading with concurrent.futures

The main features of the `concurrent.futures` package are the `ThreadPoolExecutor` and `ProcessPoolExecutor` classes, which implement an interface that allows you to submit callables for execution in different threads or processes, respectively. The classes manage an internal pool of worker threads or processes, and a queue of tasks to be

2. The images are originally from the CIA World Factbook, a public-domain, U.S. government publication. I copied them to my site to avoid the risk of launching a DOS attack on CIA.gov.

executed. But the interface is very high level and we don't need to know about any of those details for a simple use case like our flag downloads.

Example 17-3 shows the easiest way to implement the downloads concurrently, using the `ThreadPoolExecutor.map` method.

Example 17-3. flags_threadpool.py: threaded download script using futures.ThreadPoolExecutor

```
from concurrent import futures

from flags import save_flag, get_flag, show, main  ❶

MAX_WORKERS = 20  ❷


def download_one(cc):  ❸
    image = get_flag(cc)
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))  ❹
    with futures.ThreadPoolExecutor(workers) as executor:  ❺
        res = executor.map(download_one, sorted(cc_list))  ❻

    return len(list(res))  ❼


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

❶ Reuse some functions from the `flags` module (Example 17-2).

❷ Maximum number of threads to be used in the `ThreadPoolExecutor`.

❸ Function to download a single image; this is what each thread will execute.

❹ Set the number of worker threads: use the smaller number between the maximum we want to allow (`MAX_WORKERS`) and the actual items to be processed, so no unnecessary threads are created.

❺ Instantiate the `ThreadPoolExecutor` with that number of worker threads; the `executor.__exit__` method will call `executor.shutdown(wait=True)`, which will block until all threads are done.

❻ The `map` method is similar to the `map` built-in, except that the `download_one` function will be called concurrently from multiple threads; it returns a generator that can be iterated over to retrieve the value returned by each function.

7. Return the number of results obtained; if any of the threaded calls raised an exception, that exception would be raised here as the implicit `next()` call tried to retrieve the corresponding return value from the iterator.
8. Call the `main` function from the `flags` module, passing the enhanced version of `download_many`.

Note that the `download_one` function from Example 17-3 is essentially the body of the `for` loop in the `download_many` function from Example 17-2. This is a common refactoring when writing concurrent code: turning the body of a sequential `for` loop into a function to be called concurrently.

The library is called `concurrency.futures` yet there are no futures to be seen in Example 17-3, so you may be wondering where they are. The next section explains.

Where Are the Futures?

Futures are essential components in the internals of `concurrent.futures` and of `asyncio`, but as users of these libraries we sometimes don't see them. Example 17-3 leverages futures behind the scenes, but the code I wrote does not touch them directly. This section is an overview of futures, with an example that shows them in action.

As of Python 3.4, there are two classes named `Future` in the standard library: `concurrent.futures.Future` and `asyncio.Future`. They serve the same purpose: an instance of either `Future` class represents a deferred computation that may or may not have completed. This is similar to the `Deferred` class in Twisted, the `Future` class in Tornado, and `Promise` objects in various JavaScript libraries.

Futures encapsulate pending operations so that they can be put in queues, their state of completion can be queried, and their results (or exceptions) can be retrieved when available.

An important thing to know about futures in general is that you and I should not create them: they are meant to be instantiated exclusively by the concurrency framework, be it `concurrent.futures` or `asyncio`. It's easy to understand why: a `Future` represents something that will eventually happen, and the only way to be sure that something will happen is to schedule its execution. Therefore, `concurrent.futures.Future` instances are created only as the result of scheduling something for execution with a `concurrent.futures.Executor` subclass. For example, the `Executor.submit()` method takes a callable, schedules it to run, and returns a future.

Client code is not supposed to change the state of a future: the concurrency framework changes the state of a future when the computation it represents is done, and we can't control when that happens.

Both types of Future have a .done() method that is nonblocking and returns a Boolean that tells you whether the callable linked to that future has executed or not. Instead of asking whether a future is done, client code usually asks to be notified. That's why both Future classes have an .add_done_callback() method: you give it a callable, and the callable will be invoked with the future as the single argument when the future is done.

There is also a .result() method, which works the same in both classes when the future is done: it returns the result of the callable, or re-raises whatever exception might have been thrown when the callable was executed. However, when the future is not done, the behavior of the result method is very different between the two flavors of Future. In a concurrency.futures.Future instance, invoking f.result() will block the caller's thread until the result is ready. An optional timeout argument can be passed, and if the future is not done in the specified time, a TimeoutError exception is raised. In "asyncio.Future: Nonblocking by Design" on page 545, we'll see that the asyncio.Future.result method does not support timeout, and the preferred way to get the result of futures in that library is to use yield from—which doesn't work with concurrency.futures.Future instances.

Several functions in both libraries return futures; others use them in their implementation in a way that is transparent to the user. An example of the latter is the Executor.map we saw in Example 17-3: it returns an iterator in which __next__ calls the result method of each future, so what we get are the results of the futures, and not the futures themselves.

To get a practical look at futures, we can rewrite Example 17-3 to use the concurrent.futures.as_completed function, which takes an iterable of futures and returns an iterator that yields futures as they are done.

Using futures.as_completed requires changes to the download_many function only. The higher-level executor.map call is replaced by two for loops: one to create and schedule the futures, the other to retrieve their results. While we are at it, we'll add a few print calls to display each future before and after it's done. Example 17-4 shows the code for a new download_many function. The code for download_many grew from 5 to 17 lines, but now we get to inspect the mysterious futures. The remaining functions are the same as in Example 17-3.

Example 17-4. flags_threadpool_ac.py: replacing executor.map with executor.submit and futures.as_completed in the download_many function

```
def download_many(cc_list):
    cc_list = cc_list[:5]  ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor:  ❷
        to_do = []
        for cc in sorted(cc_list):  ❸
            future = executor.submit(download_one, cc)  ❹
            to_do.append(future)  ❺
            msg = 'Scheduled for {}: {}'
            print(msg.format(cc, future))    ❻

        results = []
        for future in futures.as_completed(to_do):    ❼
            res = future.result()    ❽
            msg = '{} result: {!r}'
            print(msg.format(future, res))    ❾
            results.append(res)

    return len(results)
```

❶ For this demonstration, use only the top five most populous countries.

❷ Hardcode `max_workers` to `3` so we can observe pending futures in the output.

❸ Iterate over country codes alphabetically, to make it clear that results arrive out of order.

❹ `executor.submit` schedules the callable to be executed, and returns a `future` representing this pending operation.

❺ Store each `future` so we can later retrieve them with `as_completed`.

❻ Display a message with the country code and the respective `future`.

❼ `as_completed` yields futures as they are completed.

❽ Get the result of this `future`.

❾ Display the `future` and its result.

Note that the `future.result()` call will never block in this example because the `future` is coming out of `as_completed`. Example 17-5 shows the output of one run of Example 17-4.

Example 17-5. Output of flags_threadpool_ac.py

```
$ python3 flags_threadpool_ac.py
Scheduled for BR: <Future at 0x100791518 state=running>  ❶
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending>  ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN'  ❸
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR'  ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 flags downloaded in 0.70s
```

❶ The futures are scheduled in alphabetical order; the `repr()` of a future shows its state: the first three are `running`, because there are three worker threads.

❷ The last two futures are `pending`, waiting for worker threads.

❸ The first `CN` here is the output of `download_one` in a worker thread; the rest of the line is the output of `download_many`.

❹ Here two threads output codes before `download_many` in the main thread can display the result of the first thread.

If you run *flags_threadpool_ac.py* several times, you'll see the order of the results varying. Increasing the `max_workers` argument to `5` will increase the variation in the order of the results. Decreasing it to `1` will make this code run sequentially, and the order of the results will always be the order of the `submit` calls.

We saw two variants of the download script using `concurrent.futures`: Example 17-3 with `ThreadPoolExecutor.map` and Example 17-4 with `futures.as_completed`. If you are curious about the code for *flags_asyncio.py*, you may peek at Example 18-5 in Chapter 18.

Strictly speaking, none of the concurrent scripts we tested so far can perform downloads in parallel. The `concurrent.futures` examples are limited by the GIL, and the *flags_asyncio.py* is single-threaded.

At this point, you may have questions about the informal benchmarks we just did:

- How can *flags_threadpool.py* perform 5× faster than *flags.py* if Python threads are limited by a Global Interpreter Lock (GIL) that only lets one thread run at any time?
- How can *flags_asyncio.py* perform 5× faster than *flags.py* when both are single threaded?

I will answer the second question in "Running Circling Around Blocking Calls" on page 552.

Read on to understand why the GIL is nearly harmless with I/O-bound processing.

Blocking I/O and the GIL

The CPython interpreter is not thread-safe internally, so it has a Global Interpreter Lock (GIL), which allows only one thread at a time to execute Python bytecodes. That's why a single Python process usually cannot use multiple CPU cores at the same time.[3]

When we write Python code, we have no control over the GIL, but a built-in function or an extension written in C can release the GIL while running time-consuming tasks. In fact, a Python library coded in C can manage the GIL, launch its own OS threads, and take advantage of all available CPU cores. This complicates the code of the library considerably, and most library authors don't do it.

However, all standard library functions that perform blocking I/O release the GIL when waiting for a result from the OS. This means Python programs that are I/O bound can benefit from using threads at the Python level: while one Python thread is waiting for a response from the network, the blocked I/O function releases the GIL so another thread can run.

That's why David Beazley says: "Python threads are great at doing nothing."[4]

Every blocking I/O function in the Python standard library releases the GIL, allowing other threads to run. The `time.sleep()` function also releases the GIL. Therefore, Python threads are perfectly usable in I/O-bound applications, despite the GIL.

Now let's take a brief look at a simple way to work around the GIL for CPU-bound jobs using `concurrent.futures`.

Launching Processes with concurrent.futures

The `concurrent.futures` documentation page is subtitled "Launching parallel tasks". The package does enable truly parallel computations because it supports distributing work among multiple Python processes using the `ProcessPoolExecutor` class—thus bypassing the GIL and leveraging all available CPU cores, if you need to do CPU-bound processing.

Both `ProcessPoolExecutor` and `ThreadPoolExecutor` implement the generic `Executor` interface, so it's very easy to switch from a thread-based to a process-based solution using `concurrent.futures`.

3. This is a limitation of the CPython interpreter, not of the Python language itself. Jython and IronPython are not limited in this way; but Pypy, the fastest Python interpreter available, also has a GIL.

4. Slide 106 of "Generators: The Final Frontier".

There is no advantage in using a ProcessPoolExecutor for the flags download example or any I/O-bound job. It's easy to verify this; just change these lines in Example 17-3:

```
def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))
    with futures.ThreadPoolExecutor(workers) as executor:
```

To this:

```
def download_many(cc_list):
    with futures.ProcessPoolExecutor() as executor:
```

For simple uses, the only notable difference between the two concrete executor classes is that ThreadPoolExecutor.__init__ requires a max_workers argument setting the number of threads in the pool. That is an optional argument in ProcessPoolExecutor, and most of the time we don't use it—the default is the number of CPUs returned by os.cpu_count(). This makes sense: for CPU-bound processing, it makes no sense to ask for more workers than CPUs. On the other hand, for I/O-bound processing, you may use 10, 100, or 1,000 threads in a ThreadPoolExecutor; the best number depends on what you're doing and the available memory, and finding the optimal number will require careful testing.

A few tests revealed that the average time to download the 20 flags increased to 1.8s with a ProcessPoolExecutor—compared to 1.4s in the original ThreadPoolExecutor version. The main reason for this is likely to be the limit of four concurrent downloads on my four-core machine, against 20 workers in the thread pool version.

The value of ProcessPoolExecutor is in CPU-intensive jobs. I did some performance tests with a couple of CPU-bound scripts:

arcfour_futures.py
: Encrypt and decrypt a dozen byte arrays with sizes from 149 KB to 384 KB using a pure-Python implementation of the RC4 algorithm (listing: Example A-7).

sha_futures.py
: Compute the SHA-256 hash of a dozen 1 MB byte arrays with the standard library hashlib package, which uses the OpenSSL library (listing: Example A-9).

Neither of these scripts do I/O except to display summary results. They build and process all their data in memory, so I/O does not interfere with their execution time.

Table 17-1 shows the average timings I got after 64 runs of the RC4 example and 48 runs of the SHA example. The timings include the time to actually spawn the worker processes.

Table 17-1. Time and speedup factor for the RC4 and SHA examples with one to four workers on an Intel Core i7 2.7 GHz quad-core machine, using Python 3.4

Workers	RC4 time	RC4 factor	SHA time	SHA factor
1	11.48s	1.00x	22.66s	1.00x
2	8.65s	1.33x	14.90s	1.52x
3	6.04s	1.90x	11.91s	1.90x
4	5.58s	2.06x	10.89s	2.08x

In summary, for cryptographic algorithms, you can expect to double the performance by spawning four worker processes with a `ProcessPoolExecutor`, if you have four CPU cores.

For the pure-Python RC4 example, you can get results 3.8 times faster if you use PyPy and four workers, compared with CPython and four workers. That's a speedup of 7.8 times in relation to the baseline of one worker with CPython in Table 17-1.

If you are doing CPU-intensive work in Python, you should try PyPy. The *arcfour_futures.py* example ran from 3.8 to 5.1 times faster using PyPy, depending on the number of workers used. I tested with PyPy 2.4.0, which is compatible with Python 3.2.5, so it has `concurrent.futures` in the standard library.

Now let's investigate the behavior of a thread pool with a demonstration program that launches a pool with three workers, running five callables that output timestamped messages.

Experimenting with Executor.map

The simplest way to run several callables concurrently is with the `Executor.map` function we first saw in Example 17-3. Example 17-6 is a script to demonstrate how `Executor.map` works in some detail. Its output appears in Example 17-7.

Example 17-6. demo_executor_map.py: Simple demonstration of the map method of ThreadPoolExecutor

```
from time import sleep, strftime
from concurrent import futures


def display(*args):    ❶
    print(strftime('[%H:%M:%S]'), end=' ')
    print(*args)
```

```
def loiter(n):   ❷
    msg = '{}loiter({}): doing nothing for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n * 10   ❸


def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3)   ❹
    results = executor.map(loiter, range(5))   ❺
    display('results:', results)  # ❻.
    display('Waiting for individual results:')
    for i, result in enumerate(results):   ❼
        display('result {}: {}'.format(i, result))


main()
```

❶ This function simply prints whatever arguments it gets, preceded by a timestamp in the format `[HH:MM:SS]`.

❷ `loiter` does nothing except display a message when it starts, sleep for *n* seconds, then display a message when it ends; tabs are used to indent the messages according to the value of *n*.

❸ `loiter` returns `n * 10` so we can see how to collect results.

❹ Create a `ThreadPoolExecutor` with three threads.

❺ Submit five tasks to the `executor` (because there are only three threads, only three of those tasks will start immediately: the calls `loiter(0)`, `loiter(1)`, and `loiter(2)`); this is a nonblocking call.

❻ Immediately display the `results` of invoking `executor.map`: it's a generator, as the output in Example 17-7 shows.

❼ The `enumerate` call in the `for` loop will implicitly invoke `next(results)`, which in turn will invoke `_f.result()` on the (internal) `_f` future representing the first call, `loiter(0)`. The `result` method will block until the future is done, therefore each iteration in this loop will have to wait for the next result to be ready.

I encourage you to run Example 17-6 and see the display being updated incrementally. While you're at it, play with the `max_workers` argument for the `ThreadPoolExecutor` and with the `range` function that produces the arguments for the `executor.map` call—or replace it with lists of handpicked values to create different delays.

Example 17-7 shows a sample run of Example 17-6.

Example 17-7. Sample run of demo_executor_map.py from Example 17-6

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
[15:56:50] loiter(0): done.
[15:56:50]      loiter(1): doing nothing for 1s... ❸
[15:56:50]              loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ❹
[15:56:50]                      loiter(3): doing nothing for 3s... ❺
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❻
[15:56:51]      loiter(1): done. ❼
[15:56:51]                              loiter(4): doing nothing for 4s...
[15:56:51] result 1: 10 ❽
[15:56:52]              loiter(2): done. ❾
[15:56:52] result 2: 20
[15:56:53]                      loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]                              loiter(4): done. ❿
[15:56:55] result 4: 40
```

❶ This run started at 15:56:50.

❷ The first thread executes `loiter(0)`, so it will sleep for 0s and return even before the second thread has a chance to start, but YMMV.[5]

❸ `loiter(1)` and `loiter(2)` start immediately (because the thread pool has three workers, it can run three functions concurrently).

❹ This shows that the `results` returned by `executor.map` is a generator; nothing so far would block, regardless of the number of tasks and the `max_workers` setting.

❺ Because `loiter(0)` is done, the first worker is now available to start the fourth thread for `loiter(3)`.

❻ This is where execution may block, depending on the parameters given to the `loiter` calls: the `__next__` method of the `results` generator must wait until the first future is complete. In this case, it won't block because the call to `loiter(0)` finished before this loop started. Note that everything up to this point happened within the same second: 15:56:50.

❼ `loiter(1)` is done one second later, at 15:56:51. The thread is freed to start `loiter(4)`.

5. Your mileage may vary: with threads, you never know the exact sequencing of events that should happen practically at the same time; it's possible that, in another machine, you see `loiter(1)` starting before `loiter(0)` finishes, particularly because `sleep` always releases the GIL so Python may switch to another thread even if you sleep for 0s.

8. The result of `loiter(1)` is shown: 10. Now the `for` loop will block waiting for the result of `loiter(2)`.
9. The pattern repeats: `loiter(2)` is done, its result is shown; same with `loiter(3)`.
10. There is a 2s delay until `loiter(4)` is done, because it started at 15:56:51 and did nothing for 4s.

The `Executor.map` function is easy to use but it has a feature that may or may not be helpful, depending on your needs: it returns the results exactly in the same order as the calls are started: if the first call takes 10s to produce a result, and the others take 1s each, your code will block for 10s as it tries to retrieve the first result of the generator returned by `map`. After that, you'll get the remaining results without blocking because they will be done. That's OK when you must have all the results before proceeding, but often it's preferable to get the results as they are ready, regardless of the order they were submitted. To do that, you need a combination of the `Executor.submit` method and the `futures.as_completed` function, as we saw in Example 17-4. We'll come back to this technique in "Using futures.as_completed" on page 527.

> The combination of `executor.submit` and `futures.as_completed` is more flexible than `executor.map` because you can `submit` different callables and arguments, while `executor.map` is designed to run the same callable on the different arguments. In addition, the set of futures you pass to `futures.as_completed` may come from more than one executor—perhaps some were created by a `ThreadPoolExecutor` instance while others are from a `ProcessPoolExecutor`.

In the next section, we will resume the flag download examples with new requirements that will force us to iterate over the results of `futures.as_completed` instead of using `executor.map`.

Downloads with Progress Display and Error Handling

As mentioned, the scripts in "Example: Web Downloads in Three Styles" on page 505 have no error handling to make them easier to read and to contrast the structure of the three approaches: sequential, threaded, and asynchronous.

In order to test the handling of a variety of error conditions, I created the `flags2` examples:

flags2_common.py
: This module contains common functions and settings used by all `flags2` examples, including a `main` function, which takes care of command-line parsing, timing, and

reporting results. This is really support code, not directly relevant to the subject of this chapter, so the source code is in Appendix A, Example A-10.

flags2_sequential.py
: A sequential HTTP client with proper error handling and progress bar display. Its `download_one` function is also used by `flags2_threadpool.py`.

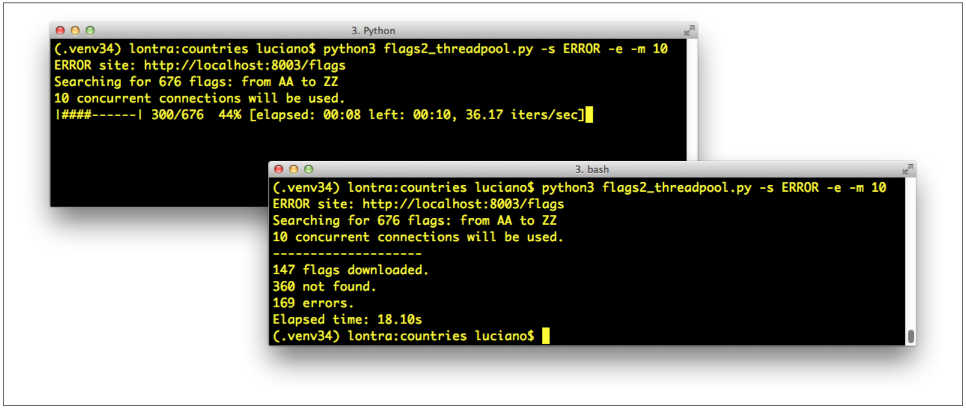
flags2_threadpool.py
: Concurrent HTTP client based on `futures.ThreadPoolExecutor` to demonstrate error handling and integration of the progress bar.

flags2_asyncio.py
: Same functionality as previous example but implemented with `asyncio` and `aiohttp`. This will be covered in "Enhancing the asyncio downloader Script" on page 554, in Chapter 18.

Be Careful When Testing Concurrent Clients

When testing concurrent HTTP clients on public HTTP servers, you may generate many requests per second, and that's how denial-of-service (DoS) attacks are made. We don't want to attack anyone, just learn how to build high-performance clients. Carefully throttle your clients when hitting public servers. For high-concurrency experiments, set up a local HTTP server for testing. Instructions for doing it are in the *README.rst* file in the *17-futures/countries/* directory of the *Fluent Python* code repository.

The most visible feature of the `flags2` examples is that they have an animated, text-mode progress bar implemented with the TQDM package. I posted a 108s video on YouTube to show the progress bar and contrast the speed of the three `flags2` scripts. In the video, I start with the sequential download, but I interrupt it after 32s because it was going to take more than 5 minutes to hit on 676 URLs and get 194 flags; I then run the threaded and `asyncio` scripts three times each, and every time they complete the job in 6s or less (i.e., more than 60 times faster). Figure 17-1 shows two screenshots: during and after running *flags2_threadpool.py*.


```
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
|####------| 300/676  44% [elapsed: 00:08 left: 00:10, 36.17 iters/sec]
```

```
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
--------------------
147 flags downloaded.
360 not found.
169 errors.
Elapsed time: 18.10s
(.venv34) lontra:countries luciano$
```

Figure 17-1. Top-left: flags2_threadpool.py running with live progress bar generated by tqdm; bottom-right: same terminal window after the script is finished.

TQDM is very easy to use, the simplest example appears in an animated *.gif* in the project's *README.md*. If you type the following code in the Python console after installing the `tqdm` package, you'll see an animated progress bar were the comment is:

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> progress bar will appear here <-
```

Besides the neat effect, the `tqdm` function is also interesting conceptually: it consumes any iterable and produces an iterator which, while it's consumed, displays the progress bar and estimates the remaining time to complete all iterations. To compute that estimate, `tqdm` needs to get an iterable that has a `len`, or receive as a second argument the expected number of items. Integrating TQDM with our `flags2` examples provide an opportunity to look deeper into how the concurrent scripts actually work, by forcing us to use the `futures.as_completed` and the `asyncio.as_completed` functions so that `tqdm` can display progress as each future is completed.

The other feature of the `flags2` example is a command-line interface. All three scripts accept the same options, and you can see them by running any of the scripts with the `-h` option. Example 17-8 shows the help text.

Example 17-8. Help screen for the scripts in the flags2 series

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
                            [-v]
                            [CC [CC ...]]
```

```
Download flags for country codes. Default: top 20 countries by population.

positional arguments:
  CC                    country code or 1st letter (eg. B for BA...BZ)

optional arguments:
  -h, --help            show this help message and exit
  -a, --all             get all available flags (AD to ZW)
  -e, --every           get flags for every possible code (AA...ZZ)
  -l N, --limit N       limit to N first codes
  -m CONCURRENT, --max_req CONCURRENT
                        maximum concurrent requests (default=30)
  -s LABEL, --server LABEL
                        Server to hit; one of DELAY, ERROR, LOCAL, REMOTE
                        (default=LOCAL)
  -v, --verbose         output detailed progress info
```

All arguments are optional. The most important arguments are discussed next.

One option you can't ignore is `-s/--server`: it lets you choose which HTTP server and base URL will be used in the test. You can pass one of four strings to determine where the script will look for the flags (the strings are case insensitive):

`LOCAL`
: Use `http://localhost:8001/flags`; this is the default. You should configure a local HTTP server to answer at port 8001. I used Nginx for my tests. The *README.rst* file for this chapter's example code explains how to install and configure it.

`REMOTE`
: Use `http://flupy.org/data/flags`; that is a public website owned by me, hosted on a shared server. Please do not pound it with too many concurrent requests. The `flupy.org` domain is handled by a free account on the Cloudflare CDN so you may notice that the first downloads are slower, but they get faster when the CDN cache warms up.[6]

`DELAY`
: Use `http://localhost:8002/flags`; a proxy delaying HTTP responses should be listening at port 8002. I used a Mozilla Vaurien in front of my local Nginx to introduce delays. The previously mentioned *README.rst* file has instructions for running a Vaurien proxy.

6. Before configuring Cloudflare, I got HTTP 503 errors—Service Temporarily Unavailable—when testing the scripts with a few dozen concurrent requests on my inexpensive shared host account. Now those errors are gone.

`ERROR`
: Use `http://localhost:8003/flags`; a proxy introducing HTTP errors and delaying responses should be installed at port 8003. I used a different Vaurien configuration for this.

> The `LOCAL` option only works if you configure and start a local HTTP server on port 8001. The `DELAY` and `ERROR` options require proxies listening on ports 8002 and 8003. Configuring Nginx and Mozilla Vaurien to enable these options is explained in the *17-futures/countries/README.rst* file in the *Fluent Python* code repository on GitHub.

By default, each `flags2` script will fetch the flags of the 20 most populous countries from the `LOCAL` server (`http://localhost:8001/flags`) using a default number of concurrent connections, which varies from script to script. Example 17-9 shows a sample run of the *flags2_sequential.py* script using all defaults.

Example 17-9. Running flags2_sequential.py with all defaults: LOCAL site, top-20 flags, 1 concurrent connection

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8001/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
--------------------
20 flags downloaded.
Elapsed time: 0.10s
```

You can select which flags will be downloaded in several ways. Example 17-10 shows how to download all flags with country codes starting with the letters A, B, or C.

Example 17-10. Run flags2_threadpool.py to fetch all flags with country codes prefixes A, B, or C from DELAY server

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8002/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
--------------------
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

Regardless of how the country codes are selected, the number of flags to fetch can be limited with the `-l/--limit` option. Example 17-11 demonstrates how to run exactly 100 requests, combining the `-a` option to get all flags with `-l 100`.

Example 17-11. Run flags2_asyncio.py to get 100 flags (-al 100) from the ERROR server, using 100 concurrent requests (-m 100)

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

That's the user interface of the `flags2` examples. Let's see how they are implemented.

Error Handling in the flags2 Examples

The common strategy adopted in all three examples to deal with HTTP errors is that 404 errors (Not Found) are handled by the function in charge of downloading a single file (`download_one`). Any other exception propagates to be handled by the `download_many` function.

Again, we'll start by studying the sequential code, which is easier to follow—and mostly reused by the thread pool script. Example 17-12 shows the functions that perform the actual downloads in the *flags2_sequential.py* and *flags2_threadpool.py* scripts.

Example 17-12. flags2_sequential.py: basic functions in charge of downloading; both are reused in flags2_threadpool.py

```
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
    if resp.status_code != 200:  ❶
        resp.raise_for_status()
    return resp.content


def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:  ❷
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found  ❸
            msg = 'not found'
        else:  ❹
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'
    if verbose:  ❺
        print(cc, msg)

    return Result(status, cc)  ❻
```

❶ get_flag does no error handling, it uses requests.Response.raise_for_status to raise an exception for any HTTP code other than 200.

❷ download_one catches requests.exceptions.HTTPError to handle HTTP code 404 specifically…

❸ …by setting its local status to HTTPStatus.not_found; HTTPStatus is an Enum imported from flags2_common (Example A-10).

❹ Any other HTTPError exception is re-raised; other exceptions will just propagate to the caller.

❺ If the -v/--verbose command-line option is set, the country code and status message will be displayed; this how you'll see progress in the verbose mode.

❻ The Result namedtuple returned by download_one will have a status field with a value of HTTPStatus.not_found or HTTPStatus.ok.

Example 17-13 lists the sequential version of the download_many function. This code is straightforward, but its worth studying to contrast with the concurrent versions coming up. Focus on how it reports progress, handles errors, and tallies downloads.

Example 17-13. flags2_sequential.py: the sequential implementation of download_many

```
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()  ❶
    cc_iter = sorted(cc_list)  ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)  ❸
    for cc in cc_iter:  ❹
        try:
            res = download_one(cc, base_url, verbose)  ❺
        except requests.exceptions.HTTPError as exc:  ❻
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:  ❼
            error_msg = 'Connection error'
        else:  ❽
            error_msg = ''
            status = res.status

        if error_msg:
            status = HTTPStatus.error  ❾
        counter[status] += 1  ❿
        if verbose and error_msg:  ⓫
```

```
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter  ⓬
```

❶ This `Counter` will tally the different download outcomes: `HTTPStatus.ok`, `HTTPStatus.not_found`, or `HTTPStatus.error`.

❷ `cc_iter` holds the list of the country codes received as arguments, ordered alphabetically.

❸ If not running in verbose mode, `cc_iter` is passed to the `tqdm` function, which will return an iterator that yields the items in `cc_iter` while also displaying the animated progress bar.

❹ This `for` loop iterates over `cc_iter` and…

❺ …performs the download by successive calls to `download_one`.

❻ HTTP-related exceptions raised by `get_flag` and not handled by `download_one` are handled here.

❼ Other network-related exceptions are handled here. Any other exception will abort the script, because the `flags2_common.main` function that calls `download_many` has no `try/except`.

❽ If no exception escaped `download_one`, then the `status` is retrieved from the `HTTPStatus` `namedtuple` returned by `download_one`.

❾ If there was an error, set the local `status` accordingly.

❿ Increment the counter by using the value of the `HTTPStatus` `Enum` as key.

⓫ If running in verbose mode, display the error message for the current country code, if any.

⓬ Return the `counter` so that the `main` function can display the numbers in its final report.

We'll now study the refactored thread pool example, *flags2_threadpool.py*.

Using futures.as_completed

In order to integrate the TQDM progress bar and handle errors on each request, the *flags2_threadpool.py* script uses `futures.ThreadPoolExecutor` with the `futures.as_completed` function we've already seen. Example 17-14 is the full listing of *flags2_threadpool.py*. Only the `download_many` function is implemented; the other functions are reused from the `flags2_common` and `flags2_sequential` modules.

Example 17-14. flags2_threadpool.py: full listing

```
import collections
from concurrent import futures
```

```
import requests
import tqdm  ❶

from flags2_common import main, HTTPStatus  ❷
from flags2_sequential import download_one  ❸

DEFAULT_CONCUR_REQ = 30  ❹
MAX_CONCUR_REQ = 1000  ❺


def download_many(cc_list, base_url, verbose, concur_req):
    counter = collections.Counter()
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor:  ❻
        to_do_map = {}  ❼
        for cc in sorted(cc_list):  ❽
            future = executor.submit(download_one,
                            cc, base_url, verbose)  ❾
            to_do_map[future] = cc  ❿
        done_iter = futures.as_completed(to_do_map)  ⓫
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list))  ⓬
        for future in done_iter:  ⓭
            try:
                res = future.result()  ⓮
            except requests.exceptions.HTTPError as exc:  ⓯
                error_msg = 'HTTP {res.status_code} - {res.reason}'
                error_msg = error_msg.format(res=exc.response)
            except requests.exceptions.ConnectionError as exc:
                error_msg = 'Connection error'
            else:
                error_msg = ''
                status = res.status

            if error_msg:
                status = HTTPStatus.error
            counter[status] += 1
            if verbose and error_msg:
                cc = to_do_map[future]  ⓰
                print('*** Error for {}: {}'.format(cc, error_msg))

    return counter


if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ Import the progress-bar display library.

❷ Import one function and one `Enum` from the `flags2_common` module.

❸ Reuse the `donwload_one` from `flags2_sequential` (Example 17-12).

❹ If the `-m/--max_req` command-line option is not given, this will be the maximum number of concurrent requests, implemented as the size of the thread pool; the actual number may be smaller, if the number of flags to download is smaller.

❺ `MAX_CONCUR_REQ` caps the maximum number of concurrent requests regardless of the number of flags to download or the `-m/--max_req` command-line option; it's a safety precaution.

❻ Create the `executor` with `max_workers` set to `concur_req`, computed by the `main` function as the smaller of: `MAX_CONCUR_REQ`, the length of `cc_list`, and the value of the `-m/--max_req` command-line option. This avoids creating more threads than necessary.

❼ This `dict` will map each `Future` instance—representing one download—with the respective country code for error reporting.

❽ Iterate over the list of country codes in alphabetical order. The order of the results will depend on the timing of the HTTP responses more than anything, but if the size of the thread pool (given by `concur_req`) is much smaller than `len(cc_list)`, you may notice the downloads batched alphabetically.

❾ Each call to `executor.submit` schedules the execution of one callable and returns a `Future` instance. The first argument is the callable, the rest are the arguments it will receive.

❿ Store the `future` and the country code in the `dict`.

⓫ `futures.as_completed` returns an iterator that yields futures as they are done.

⓬ If not in verbose mode, wrap the result of `as_completed` with the `tqdm` function to display the progress bar; because `done_iter` has no `len`, we must tell `tqdm` what is the expected number of items as the `total=` argument, so `tqdm` can estimate the work remaining.

⓭ Iterate over the futures as they are completed.

⓮ Calling the `result` method on a future either returns the value returned by the callable, or raises whatever exception was caught when the callable was executed. This method may block waiting for a resolution, but not in this example because `as_completed` only returns futures that are done.

⓯ Handle the potential exceptions; the rest of this function is identical to the sequential version of `download_many` (Example 17-13), except for the next callout.

⓰ To provide context for the error message, retrieve the country code from the `to_do_map` using the current `future` as key. This was not necessary in the sequential version because we were iterating over the list of country codes, so we had the current `cc`; here we are iterating over the futures.

Example 17-14 uses an idiom that's very useful with `futures.as_completed`: building a `dict` to map each future to other data that may be useful when the future is completed. Here the `to_do_map` maps each future to the country code assigned to it. This makes it easy to do follow-up processing with the result of the futures, despite the fact that they are produced out of order.

Python threads are well suited for I/O-intensive applications, and the `concurrent.futures` package makes them trivially simple to use for certain use cases. This concludes our basic introduction to `concurrent.futures`. Let's now discuss alternatives for when `ThreadPoolExecutor` or `ProcessPoolExecutor` are not suitable.

Threading and Multiprocessing Alternatives

Python has supported threads since its release 0.9.8 (1993); `concurrent.futures` is just the latest way of using them. In Python 3, the original `thread` module was deprecated in favor of the higher-level `threading` module.[7] If `futures.ThreadPoolExecutor` is not flexible enough for a certain job, you may need to build your own solution out of basic `threading` components such as `Thread`, `Lock`, `Semaphore`, etc.—possibly using the thread-safe queues of the `queue` module for passing data between threads. Those moving parts are encapsulated by `futures.ThreadPoolExecutor`.

For CPU-bound work, you need to sidestep the GIL by launching multiple processes. The `futures.ProcessPoolExecutor` is the easiest way to do it. But again, if your use case is complex, you'll need more advanced tools. The `multiprocessing` package emulates the `threading` API but delegates jobs to multiple processes. For simple programs, `multiprocessing` can replace `threading` with few changes. But `multiprocessing` also offers facilities to solve the biggest challenge faced by collaborating processes: how to pass around data.

Chapter Summary

We started the chapter by comparing two concurrent HTTP clients with a sequential one, demonstrating significant performance gains over the sequential script.

After studying the first example based on `concurrent.futures`, we took a closer look at future objects, either instances of `concurrent.futures.Future`, or `asyncio.Future`, emphasizing what these classes have in common (their differences will be emphasized in Chapter 18). We saw how to create futures by calling `Executor.sub`

7. The `threading` module has been available since Python 1.5.1 (1998), yet some insist on using the old `thread` module. In Python 3, it was renamed to `_thread` to highlight the fact that it's just a low-level implementation detail, and shouldn't be used in application code.

mit(…), and iterate over completed futures with concurrent.futures.as_comple ted(…).

Next, we saw why Python threads are well suited for I/O-bound applications, despite the GIL: every standard library I/O function written in C releases the GIL, so while a given thread is waiting for I/O, the Python scheduler can switch to another thread. We then discussed the use of multiple processes with the concurrent.futures.Proces sPoolExecutor class, to go around the GIL and use multiple CPU cores to run cryptographic algorithms, achieving speedups of more than 100% when using four workers.

In the following section, we took a close look at how the concurrent.futures.Thread PoolExecutor works, with a didactic example launching tasks that did nothing for a few seconds, except displaying their status with a timestamp.

Next we went back to the flag downloading examples. Enhancing them with a progress bar and proper error handling prompted further exploration of the future.as_comple ted generator function showing a common pattern: storing futures in a dict to link further information to them when submitting, so that we can use that information when the future comes out of the as_completed iterator.

We concluded the coverage of concurrency with threads and processes with a brief reminder of the lower-level, but more flexible threading and multiprocessing modules, which represent the traditional way of leveraging threads and processes in Python.

Further Reading

The concurrent.futures package was contributed by Brian Quinlan, who presented it in a great talk titled "The Future Is Soon!" at PyCon Australia 2010. Quinlan's talk has no slides; he shows what the library does by typing code directly in the Python console. As a motivating example, the presentation features a short video with XKCD cartoonist/programmer Randall Munroe making an unintended DOS attack on Google Maps to build a colored map of driving times around his city. The formal introduction to the library is PEP 3148 - futures - execute computations asynchronously. In the PEP, Quinlan wrote that the concurrent.futures library was "heavily influenced by the Java java.util.concurrent package."

Parallel Programming with Python (Packt), by Jan Palach, covers several tools for concurrent programming, including the concurrent.futures, threading, and multiproc essing modules. It goes beyond the standard library to discuss Celery, a task queue used to distribute work across threads and processes, even on different machines. In the Django community, Celery is probably the most widely used system to offload heavy tasks such as PDF generation to other processes, thus avoiding delays in producing an HTTP response.

In the Beazley and Jones *Python Cookbook, 3E* (O'Reilly) there are recipes using `concurrent.futures` starting with "Recipe 11.12. Understanding Event-Driven I/O." "Recipe 12.7. Creating a Thread Pool" shows a simple TCP echo server, and "Recipe 12.8. Performing Simple Parallel Programming" offers a very practical example: analyzing a whole directory of `gzip` compressed Apache logfiles with the help of a `ProcessPoolExecutor`. For more about threads, the entire Chapter 12 of Beazley and Jones is great, with special mention to "Recipe 12.10. Defining an Actor Task," which demonstrates the Actor model: a proven way of coordinating threads through message passing.

Brett Slatkin's *Effective Python* (Addison-Wesley) has a multitopic chapter about concurrency, including coverage of coroutines, `concurrent.futures` with threads and processes, and the use of locks and queues for thread programming without the `ThreadPoolExecutor`.

High Performance Python (O'Reilly) by Micha Gorelick and Ian Ozsvald and *The Python Standard Library by Example* (Addison-Wesley), by Doug Hellmann, also cover threads and processes.

For a modern take on concurrency without threads or callbacks, *Seven Concurrency Models in Seven Weeks*, by Paul Butcher (Pragmatic Bookshelf) is an excellent read. I love its subtitle: "When Threads Unravel." In that book, threads and locks are covered in Chapter 1, and the remaining six chapters are devoted to modern alternatives to concurrent programming, as supported by different languages. Python, Ruby, and JavaScript are not among them.

If you are intrigued about the GIL, start with the *Python Library and Extension FAQ* ("Can't we get rid of the Global Interpreter Lock?"). Also worth reading are posts by Guido van Rossum and Jesse Noller (contributor of the `multiprocessing` package): "It isn't Easy to Remove the GIL" and "Python Threads and the Global Interpreter Lock." Finally, David Beazley has a detailed exploration on the inner workings of the GIL: "Understanding the Python GIL."[8] In slide #54 of the presentation, Beazley reports some alarming results, including a 20× increase in processing time for a particular benchmark with the new GIL algorithm introduced in Python 3.2. However, Beazley apparently used an empty `while True: pass` to simulate CPU-bound work, and that is not realistic. The issue is not significant with real workloads, according to a comment by Antoine Pitrou—who implemented the new GIL algorithm—in the bug report submitted by Beazley.

While the GIL is real problem and is not likely to go away soon, Jesse Noller and Richard Oudkerk contributed a library to make it easier to work around it in CPU-bound applications: the `multiprocessing` package, which emulates the `threading` API across processes, along with supporting infrastructure of locks, queues, pipes, shared memory,

8. Thanks to Lucas Brunialti for sending me a link to this talk.

etc. The package was introduced in PEP 371 — Addition of the multiprocessing package to the standard library. The official documentation for the package is a 93 KB *.rst* file—that's about 63 pages—making it one of the longest chapters in the Python standard library. Multiprocessing is the basis for the `concurrent.futures.ProcessPoolExecutor`.

For CPU- and data-intensive parallel processing, a new option with a lot of momentum in the big data community is the Apache Spark distributed computing engine, offering a friendly Python API and support for Python objects as data, as shown in their examples page.

Two elegant and super easy libraries for parallelizing tasks over processes are `lelo` by João S. O. Bueno and `python-parallelize` by Nat Pryce. The `lelo` package defines a `@parallel` decorator that you can apply to any function to magically make it unblocking: when you call the decorated function, its execution is started in another process. Nat Pryce's `python-parallelize` package provides a `parallelize` generator that you can use to distribute the execution of a `for` loop over multiple CPUs. Both packages use the `multiprocessing` module under the covers.

Soapbox

Thread Avoidance

Concurrency: one of the most difficult topics in computer science (usually best avoided).[9]

— David Beazley
Python coach and mad scientist

I agree with the apparently contradictory quotes by David Beazley, above, and Michele Simionato at the start of this chapter. After attending a concurrency course at the university—in which "concurrent programming" was equated to managing threads and locks—I came to the conclusion that I don't want to manage threads and locks myself, any more than I want to manage memory allocation and deallocation. Those jobs are best carried out by the systems programmers who have the know-how, the inclination, and the time to get them right—hopefully.

That's why I think the `concurrent.futures` package is exciting: it treats threads, processes, and queues as infrastructure at your service, not something you have to deal with directly. Of course, it's designed with simple jobs in mind, the so-called "embarrassingly parallel" problems. But that's a large slice of the concurrency problems we face when writing applications—as opposed to operating systems or database servers, as Simionato points out in that quote.

9. Slide #9 from "A Curious Course on Coroutines and Concurrency," tutorial presented at PyCon 2009.

For “nonembarrassing” concurrency problems, threads and locks are not the answer either. Threads will never disappear at the OS level, but every programming language I’ve found exciting in the last several years provides better, higher-level, concurrency abstractions, as the *Seven Concurrency Models* book demonstrates. Go, Elixir, and Clojure are among them. Erlang—the implementation language of Elixir—is a prime example of a language designed from the ground up with concurrency in mind. It doesn’t excite me for a simple reason: I find its syntax ugly. Python spoiled me that way.

José Valim, well-known as a Ruby on Rails core contributor, designed Elixir with a pleasant, modern syntax. Like Lisp and Clojure, Elixir implements syntactic macros. That’s a double-edged sword. Syntactic macros enable powerful DSLs, but the proliferation of sublanguages can lead to incompatible codebases and community fragmentation. Lisp drowned in a flood of macros, with each Lisp shop using its own arcane dialect. Standardizing around Common Lisp resulted in a bloated language. I hope José Valim can inspire the Elixir community to avoid a similar outcome.

Like Elixir, Go is a modern language with fresh ideas. But, in some regards, it’s a conservative language, compared to Elixir. Go doesn’t have macros, and its syntax is simpler than Python’s. Go doesn’t support inheritance or operator overloading, and it offers fewer opportunities for metaprogramming than Python. These limitations are considered features. They lead to more predictable behavior and performance. That’s a big plus in the highly concurrent, mission-critical settings where Go aims to replace C++, Java, and Python.

While Elixir and Go are direct competitors in the high-concurrency space, their design philosophies appeal to different crowds. Both are likely to thrive. But in the history of programming languages, the conservative ones tend to attract more coders. I’d like to become fluent in Go and Elixir.

About the GIL

The GIL simplifies the implementation of the CPython interpreter and of extensions written in C, so we can thank the GIL for the vast number of extensions in C available for Python—and that is certainly one of the key reasons why Python is so popular today.

For many years, I was under the impression that the GIL made Python threads nearly useless beyond toy applications. It was not until I discovered that *every* blocking I/O call in the standard library releases the GIL that I realized Python threads are excellent for I/O-bound systems—the kind of applications customers usually pay me to develop, given my professional experience.

Concurrency in the Competition

MRI—the reference implementation of Ruby—also has a GIL, so its threads are under the same limitations as Python's. Meanwhile, JavaScript interpreters don't support user-level threads at all; asynchronous programming with callbacks is their only path to concurrency. I mention this because Ruby and JavaScript are the closest direct competitors to Python as general-purpose, dynamic programming languages.

Looking at the concurrency-savvy new crop of languages, Go and Elixir are probably the ones best positioned to eat Python's lunch. But now we have `asyncio`. If hordes of people believe Node.js with raw callbacks is a viable platform for concurrent programming, how hard can it be to win them over to Python when the `asyncio` ecosystem matures? But that's a topic for the next "Soapbox" on page 580.