JULIEN DANJOU

THE HACKER'S GUIDE TO SCALING PYTHON

ISBN 978-1-387-37932-3

Contents

List of Figures

List of Examples

About this Book

Version 1.0 released in 2017.

When I released The Hacker's Guide to Python in 2014, I had no idea that I would be writing a new book so soon. Having worked on OpenStack for a few more years, I saw how it is easy to struggle with other aspects of Python, even after being on board for a while.

Nowadays, even if computers are super-fast, no server is fast enough to handle millions of request per second, which is a typical workload we want to use them for. Back in the day, when your application was slow, you just had to optimize it or upgrade your hardware – whichever was cheaper. However, in a world where you may already have done both, you need to be able to scale your application horizontally, i.e., you have to make it run on multiple computers in parallel.

That is usually the start of a long journey, filled with concurrency problems and disaster scenarios.

Developers often dismiss Python when they want to write performance enhancing, and distributed applications. They tend to consider the language to be slow and not suited to that task. Sure, Python is not Erlang, but there's also no need to ditch it for Go because of everyone *saying* it is faster.

I would like to make you aware, dear reader, that a language is never slow. You would not say that English or French is slow, right? The same applies for programming languages. The only thing that can be slow is the implementation of the language – in Python's case, its reference implementation is CPython.

Indeed CPython can be quite sluggish, and it has its share of problems. Every

implementation of a programming language has its downside. However, I think that the ecosystem of Python can make up for that defect.

Python and everything that evolves around it offer a large set of possibilities to extend your application, so it can manage thousands of requests simultaneously, compensating for its lack of distributed design or, sometimes its "slowness".

Moreover, if you need proof, you can ask companies such as Dropbox, PayPal or Google as they all use Python on a large scale. Instagram has 400 million active users every day and their whole stack is served using Python and Django.

In this book, we will discuss how one can push Python further and build applications that can scale horizontally, perform well and remain fast while being distributed. I hope it makes you more productive at Python and allows you to write better applications that are also faster!

Most code in this book targets Python 3. Some snippets might work on Python 2 without much change, but there is no guarantee.

CHAPTER 1

Scaling?

> Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.
>
> — Wikipedia

When we talk about scaling Python, what we mean is making Python application scalable. However, what is scalability?

According to Wikipedia, scalability is "the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth". This definition makes scalability difficult to define as an absolute since no definition applies to all applications.

This book concentrates on methods, technologies, and practice that allow one to make applications fast and able to grow in order to handle more jobs – all of

that using the Python programming language and its major implementation, named CPython.

We are all aware that processors are not becoming faster and faster at a rate where a single threaded application could, one day, be *fast enough* to handle any size workload. That means you need to think about using more than just one processor. Building scalable applications implies that you distribute the workload across multiple workers using multiple processing units.

Dividing up the tasks at hand, those workers run across several processors, and in some cases, across several computers.

That is a *distributed application*.

There are fundamental properties to understand about distributed systems before digging into how to build them in Python – or any other language.

We can lay out the following options when writing an application:

- Write a single-threaded application. This should be your first pick, and indeed it implies no distribution. They are the simplest of all applications. They are easy to understand and therefore easier to maintain. However, they are limited by the power of using a single processor.

- Write a multi-threaded application. Most computers – even your smartphone – are now equipped with multiple processing units. If an application can overload an entire CPUs, it needs to spread its workload over other processors by spawning new threads (or new processes). Multi-threading applications are more error-prone than single-threaded applications, but they offer fewer failure scenarios than multi-nodes applications, as no network is involved.

- Write network distributed applications. This is your last resort when your application needs to scale significantly, and not even one big computer with plenty of CPUs is enough. Those are the most complicated applications to write as they use a network. It means they should handle a lot of scenarios, such as a total or partial

failure of a node or the network, high latency, messages being lost, and any other terrible property related to the unreliability of networks.

The properties of distribution vary widely depending on the type you pick. Operations on a single processor can be regarded as fast, with low latency while being reliable, and ordered, whereas operations across several nodes should be considered, slow, with high latency. They are often unreliable and unordered.

Consider each architecture choice or change carefully. As seen throughout this book, there are various tools and methods in Python available for dealing with any of those choices. They help to build distributed systems, and therefore scalable applications.

1.1 Across CPUs

Scaling across processors is usually done using multithreading. Multithreading is the ability to run code in parallel using *threads*. Threads are usually provided by the operating system and are contained in a single process. The operating system is responsible to schedule their execution.

Since they run in parallel, that means they can be executed on separate processors even if they are contained in a single process. However, if only one CPU is available, the code is split up and run sequentially.

Therefore, when writing a multithreaded application, the code always runs concurrently but runs in parallel only if there is more than one CPU available.

This means that multithreading looks like a good way to scale and parallelize your application on one computer. When you want to spread the workload, you start a new thread for each new request instead of handling them one at a time.

However, this does have several drawbacks in Python. If you have been in the Python world for a long time, you have probably encountered the word *GIL*, and

know how hated it is. The GIL is the Python *global interpreter lock*, a lock that must be acquired each time *CPython* needs to execute byte-code. Unfortunately, this means that if you try to scale your application by making it run multiple threads, this global lock always limits the performance of your code, as there are many conflicting demands. All your threads try to grab it as soon as they need to execute Python instructions.

The reason that the *GIL* is required in the first place is that it makes sure that some basic Python objects are thread-safe. For example, the code in Example 1.1 would not be thread-safe without the global Python lock.

Example 1.1 Thread-unsafe code without the GIL

```
import threading

x = []

def append_two(l):
    l.append(2)

threading.Thread(target=append_two, args=(x,)).start()

x.append(1)
print(x)
```

That code prints either `[2, 1]` or `[1, 2]`, no matter what. While there is no way to know which thread appends 1 or 2 before the other, there is an assumption built into Python that each `list.append` operation is **atomic**. If it was not atomic, a memory corruption might arise and the list could simply contain `[1]` or `[2]`.

This phenomenon happens because only one thread is allowed to execute a **bytecode** instruction at a time. That also means that if your threads run a lot of bytecodes, there are many contentions to acquire the GIL, and therefore your program cannot be faster than a single-threaded version – or it could even be slower.

The easiest to way to know if an operation is thread-safe is to know if it translates

to a single bytecode instruction [1] or if it uses a basic type whose operations are atomic [2].

So while using threads seems like an ideal solution at first glance, most applications I have seen running using multiple threads struggle to attain 150% CPU usage – that is to say, 1.5 cores used. With computing nodes nowadays usually not having less than four or eight cores, it is a shame. Blame the GIL.

There is currently an effort underway (named gilectomy) to remove the GIL in *CPython*. Whether this effort will pay off is still unknown, but it is exciting to follow and see how far it will go.

However, *CPython* is just one – although the most common – of the available Python implementations. Jython, for example, doesn't have a global interpreter lock, which means that it can run multiple threads in parallel efficiently. Unfortunately, these projects by their very natures lag behind *CPython*, and so they are not useful targets.

Multithreading involves several traps, and one of them is that all the pieces of code running concurrently are sharing the same global environment and variables. Reading or writing global variables should be done exclusive by using techniques such as locking, which complicates your code; moreover, it is an infinite source of human errors.

Getting multi-threaded applications right is hard. The level of complexity means that it is a large source of bugs – and considering the little to be gained in general, it is better not to waste too much effort on it.

So are we back to our initial use cases, with no real solutions on offer? Not true – there's another solution you can use: using multiple processes. Doing this is going to be more efficient and easier as we will see in Chapter 2. It is also a first step before spreading across a network.

[1] Details about disassembling code and bytecode instruction are provided in Section 13.4.
[2] The list is provided in the Python FAQ.

1.2 Distributed Systems

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> — Lamport *(1987)*

When an application uses all the CPU power of a node, and you cannot add more processors to your server or switch to a bigger server, you need a plan B.

The next step usually involves multiple servers, linked together via a network of some sort. That means the application starts to be *distributed*: running not only on one node but on multiple, connected, nodes. Spreading the workload over different hosts introduces several advantages, such as:

- Horizontal scalability, the ability to add more nodes as more traffic comes in
- Fault tolerance, as if a node goes down, another one can pick up the traffic of the dysfunctioning one

While this sounds awesome, it also introduces major drawbacks:

- As with multithreading, concurrency and parallelism come into play and complicate the workflow (e.g., locking usage)
- What can fail will fail, such as a random node in the middle of an operation or a laggy network, so tolerance for failure must be built-in

All of this means that an application which is going the distributed route expand its complexity while potentially increasing its throughput. Making this kind of architectural decision requires great wisdom.

Python does not offer so many tools for building a distributed system, but its ecosystem has a few good options as seen throughout this book. For example, it can be pretty easy to distribute jobs across several nodes as covered in Chapter 5. Bigger problems such as coordination and synchronization with sibling nodes also have a few solutions, as discussed in Chapter 7 and Chapter 8.

Finally, a great approach to writing distributed systems is to make them purely functional, i.e., without any shared state. That means such applications should not have even a single shared, global variable, across all of its distributed processes. Stateless systems are the easiest ones to distribute and scale, and therefore systems should be designed as such when possible. Chapter 4 talks about functional programming and the mindset behind writing such programs.

1.3 Service-Oriented Architecture

If you've never heard of it, service-oriented architecture is an architectural style where a software design is made up of several independent components communicating over a network. Each service is a discrete unit of functionality that can work autonomously. That means that the problem should be divided up into interacting logical pieces. If we refer back to the different application styles defined in Chapter 1, SOA refers to network distributed applications.

This kind of architecture is not a perfect or magical solution. It has many drawbacks, but it also has many advantages that make it valuable… and so popular these days for building distributed applications.

The service-oriented architecture is not a first-class citizen in Python, though it makes it easy to use and implement – the language being generic and the ecosystem

rich enough.

Services built for this kind of architecture should follow a few principles [3] among them being **stateless**. That means services must either modify and return the requested value (or an error) while separating their functioning from the state of the data. This is an essential property, as it makes it easier to scale the services horizontally.

Statelessness is a property that is also shared with the functional programming paradigm, as discussed in Chapter 4. Both of these are relevant topics and principles to know about when designing scalable applications.

How to split your application into different services might deserve a book on its own, but there are mainly two categories:

- Object-oriented approach: each *noun* is a service, e.g., catalog service, phone service, queue service, etc. Such service types are a good way to represent data types.
- Functional approach: each *verb* is a service, e.g., search service, authentication service, crawl service, etc. Such service types are a good way to represent transformations.

Having too many services has a cost, as they come with some overheard. Think of all of the costs associated, such as maintenance and deployment, and not only development time. Splitting an application should always be a well-thought out decision.

If you know that some services need to scale independently, you should probably split them. However, if they are latency sensitive and should work together very closely, involving a lot of communication, that might be where the line is drawn.

Software production is both a technical and social artifact: there might be some services that come naturally to mind due to the social organization of your project.

[3] Wikipedia offers a great list of those principles.

For example, some teams might be responsible for the user database, so that might be their job to create and maintain an independent user service that other components can use to get information and authenticate it. This is also important to take into consideration when choosing where to set the boundaries of your different services.

Once this is all set [4], the technical aspect of the implementation comes to mind. Nowadays, the most common type of services that are encountered are Web services, base on the well-known and ubiquitous HTTP protocol. This is what will be largely discussed in Chapter 9.

[4] Of course no architecture is written in stone, and everything can evolve, likewise social groups change and services might come and go.

CHAPTER 2

CPU Scaling

As CPUs are not getting infinitely faster, using multiple CPUs is the best path towards scalability. That means introducing concurrency and parallelism into your program, and that is not an easy task. However, once correctly done, it really does increase the total throughput.

Python offers two options to spread your workload across multiple local CPUs: threads or processes. They both come with challenges; some are not specifically tied to Python, while some are only relevant to its main implementation, i.e., *CPython*.

2.1 Using Threads

Threads in Python are a good way to run a function concurrently other functions. If your system does not support multiple processors, the threads will be executed one after another as scheduled by the operating system. However, if multiple CPUs are available, threads could be scheduled on multiple processing units, once again as determined by the operating system.

By default, there is only one thread − the main thread − and it is the thread

that runs your Python application. To start another thread, Python provides the `threading` module.

Example 2.1 Starting a new thread

```
import threading


def print_something(something):
    print(something)


t = threading.Thread(target=print_something, args=("hello",))
t.start()
print("thread started")
t.join()
```

If you run the program in Example 2.1 multiple times, you will notice that the output might be different each time. On my laptop, doing this gives the following:

```
$ python examples/chapter2-cpu-scaling/threading-start.py
hellothread started
$ python examples/chapter2-cpu-scaling/threading-start.py
hello
 thread started
$ python examples/chapter2-cpu-scaling/threading-start.py
hello
thread started
```

If you specifically expected any one of the outputs each time, then you forgot that there is no guarantee regarding the order of execution for the threads.

Once started, the threads join: the main thread waits for the second thread to complete by calling its `join` method. Using `join` is handy in terms of not leaving any threads behind.

If you do not join all your threads and wait for them to finish, it is possible that the main thread finishes and exits before the other threads. If this happens, your

program will appear to be blocked and will not respond to even a simple `KeyboardInterrupt` signal.

To avoid this, and because your program might not be in a position to wait for the threads, you can configure threads as *daemons*. When a thread is a daemon, it is considered as a background thread by Python and is terminated as soon as the main thread exists.

Example 2.2 Starting a new thread in daemon mode

```
import threading


def print_something(something):
    print(something)


t = threading.Thread(target=print_something, args=("hello",))
t.daemon = True
t.start()
print("thread started")
```

In Example 2.2 , there is no longer a need to use the `join` method since the thread is set to be a `daemon`.

The program below is a simple example, which sums one million random integers eight times, spread across eight threads at the same time.

Example 2.3 Workers using multithreading

```
import random
import threading

results = []


def compute():
```

```
    results.append(sum(
        [random.randint(1, 100) for i in range(1000000)]))


workers = [threading.Thread(target=compute) for x in range(8)]
for worker in workers:
    worker.start()
for worker in workers:
    worker.join()
print("Results: %s" % results)
```

Running Example 2.3 program returns the following:

```
$ time python multithreading-worker.py
Results: [50505811, 50471217, 50531481, 50460206, 50462903, 50533718,  ↩
    50500182, 50480848]
python examples/multithreading-worker.py 19.84s user 6.32s system 116% ↩
     cpu 22.501 total
```

The program ran on an idle quad cores CPU, which means that Python could have used up to 400% CPU power. However, it was unable to do that, even with eight threads running in parallel – it stuck at 116%, which is just 29% of the hardware's capabilities.

The graph in Figure 2.1 illustrates that bottleneck: to access all of the system's CPU, you need to go through CPython's GIL.

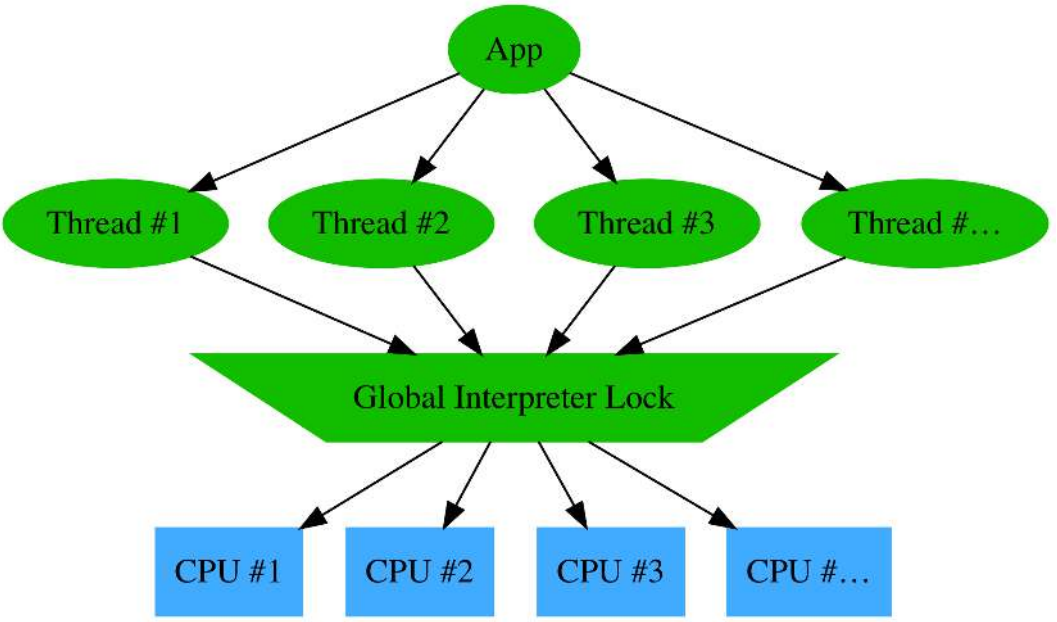


Figure 2.1: Using threads with CPython

Again, as discussed in Section 1.1, the GIL limits the performance of *CPython* when executing multiple threads. Threads are therefore useful when doing parallel computing or input/output on slow networks or files: those tasks can run in parallel without blocking the main thread.

To achieve a greater throughput using multiple CPUs, using processes is an interesting alternative discussed in Section 2.2.

2.2 Using Processes

Since multithreading is not a perfect scalability solution because of the *GIL*, using processes instead of threads is a good alternative. Python obviously exposes the `os`.

`fork` system call to create new processes. However, this approach is a little bit too low-level to be interesting in most cases.

Instead, the **multiprocessing** package is a good higher-level alternative. It provides an interface that starts new processes, whatever your operating system might be.

We can rewrite Example 2.3 using processes thanks to the *multiprocessing* library, as shown in Example 2.4.

Example 2.4 `multiprocessing.Process` usage

```
import random
import multiprocessing


def compute(results):
    results.append(sum(
        [random.randint(1, 100) for i in range(1000000)]))


with multiprocessing.Manager() as manager:
    results = manager.list()
    workers = [multiprocessing.Process(target=compute, args=(results,)
        )
              for x in range(8)]
    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()
    print("Results: %s" % results)
```

The example is a bit trickier to write as there is no data shared available between different processes. Since each process is a new independent Python, the data is **copied** and each process has its own independent global state. The `multiprocessing.Manager` class provides a way to create shared data structures that are safe for concurrent accesses.

Running this program gives the following result:

Example 2.5 Result of `time python multiprocessing-workers.py`

```
$ time python multiprocessing-workers.py
Results: [50505465, 50524237, 50492168, 50482321, 50503634, 50543646, ↩
    50533775, 50521610]
python examples/multiprocessing-workers.py 32.00s user 0.50s system ↩
    332% cpu 9.764 total
```

Compared to Example 2.3, using multiple processes reduces the execution times by 60%. This time, the processes have been able to consume up to 332% of the CPU power, which is more than 80% of the computer's CPU capacity, or close to three times more than multithreading.

The graph in Figure 2.2 tries to lay out the differences in terms of how scheduling processes work and why it is more efficient than using threads, as shown previously.

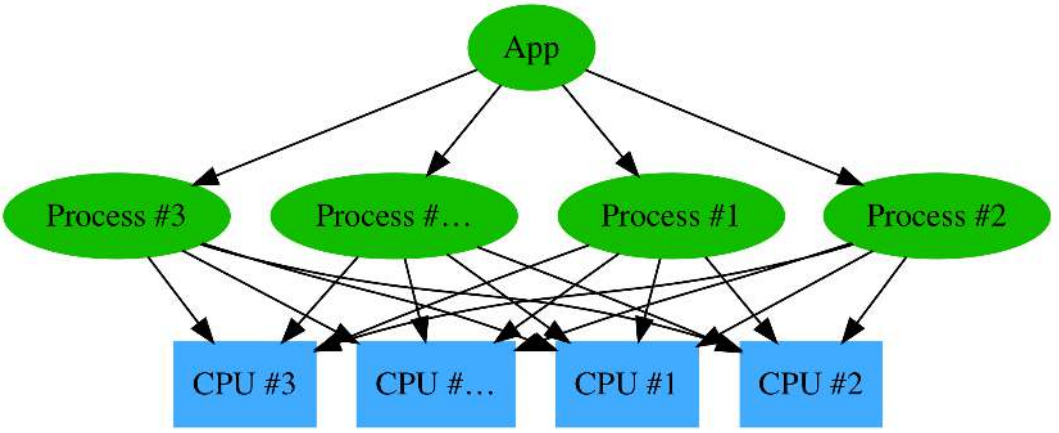


Figure 2.2: Using processes with CPython

Each time some work can be **parallelized** for a certain amount of time, it's

much better to rely on `multiprocessing` and to fork jobs, thiu spreading the workload among several CPU cores, rather than using the `threading` module.

The multiprocessing library also provides a *pool* mechanism that is useful to rewrite the code from Example 2.4 in a more *functional* manner; an example is provided as Example 2.6.

Example 2.6 Worker using multiprocessing

```
import multiprocessing
import random


def compute(n):
    return sum(
        [random.randint(1, 100) for i in range(1000000)])


# Start 8 workers
pool = multiprocessing.Pool(processes=8)
print("Results: %s" % pool.map(compute, range(8)))
```

Using `multiprocessing.Pool`, there is no need to manage the processes "manually". The pool starts processes on-demand and takes care of reaping them when done. They are also reusable, which avoids calling the `fork` syscall too often – which is quite costly. It is a convenient design pattern that is also leveraged in futures, as discussed in Section 2.3.

2.3 Using Futures

Python 3.2 introduced the `concurrent.futures` module, which provides an easy way to schedule asynchronous tasks. The module is also available in Python 2 as it has been back-ported – it can easily be installed it by running `pip install futures`.

The `concurrent.futures` module is pretty straightforward to use. First, one needs to pick an *executor*. An executor is responsible for scheduling and running asynchronous tasks. It can be seen as a type of engine for execution. The module currently provides two kinds of executors: `concurrent.futures.ThreadPoolExecutor` and `concurrent.futures.ProcessPoolExecutor`. As one might guess, the first one is based on threads and the second one on processes.

As outlined in Section 1.1, the process based executor is going to be much more efficient for long-running tasks that benefit from having an entire CPU available. The threading executor suffers from the same limitation of the `threading` module, which was covered earlier.

So what is interesting with the `concurrent.futures` module is that it provides an easier to use abstraction layer on top of the `threading` and `multiprocessing` modules. It allows one to run and parallelize code in a straightforward way, providing an abstract data structure called a `concurrent.futures.Future` object.

Each time a program schedules some tasks to execute in threads or processes, the `concurrent.futures` module returns a `Future` object for each of the task scheduled. This `Future` object owns the promise of the work to be completed. Once that work is achieved, the result is available in that `Future` object – so in the end, it does represent the future and the promise of a task to be performed. That is why it is called `Future` in Python, and sometimes *promise* in other languages.

Example 2.7 Worker using `concurrent.futures.ThreadPoolExecutor`

```
from concurrent import futures
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(1000000)])


```

```
with futures.ThreadPoolExecutor(max_workers=8) as executor:
    futures = [executor.submit(compute) for _ in range(8)]

results = [f.result() for f in futures]

print("Results: %s" % results)
```

Compared to the `threading` based example script, you might notice that this one is more functional. I changed the `compute` function to *return* the result rather than changing a shared object. It is then easy to manipulate and transfer the `Future` object and collect the result as desired when it is needed. Functional programming is a perfect paradigm to embrace when trying to spread workload across distributed workers – it is covered more in Chapter 4.

The code just schedules the jobs to be fulfilled and collects the results from the `Future` objects using the `result` method – which also supports a `timeout` parameter in case the program cannot hang for too long. `Future` objects offer some more interesting methods:

- `done()`: This returns `True` if the call was successfully canceled or terminated correctly.
- `add_done_callback(fn)`: This attaches a callable to the future which is called with the future as its only argument; it is done as soon as the future is canceled or terminates correctly.

Example 2.8 Time and output of futures-threads-worker

```
$ time python futures-threads-worker.py
Results: [50532744, 50524277, 50507195, 50501211, 50537292, 50490570, ↩
    50484569, 50515144]
python futures-threads-worker.py 14.50s user 6.91s system 126% cpu ↩
    16.893 total
```

The execution time is in the same low range as the example using the `threading` technique: indeed, the underlying engine is based on the `threading` module.

Keep in mind that `concurrent.futures` allows you to easily switch from threads to processes by using the `concurrent.futures.ProcessPoolExecutor`:

Example 2.9 Worker using `concurrent.futures.ProcessPoolExecutor`

```
from concurrent import futures
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(1000000)])


with futures.ProcessPoolExecutor() as executor:
    futures = [executor.submit(compute) for _ in range(8)]

results = [f.result() for f in futures]

print("Results: %s" % results)
```

There is no need to set the number of `max_workers`: as by default `concurrent.futures` calls the `multiprocessing.cpu_count` function to set the number of workers to use, which is equal to the number of CPUs the system can use – as is shown in Example 2.10

Example 2.10 Extract of `concurrent.futures.process`

```
class ProcessPoolExecutor(_base.Executor):
    def __init__(self, max_workers=None):
        # [...]
        if max_workers is None:
            self._max_workers = multiprocessing.cpu_count()
        else:
```

```
            self._max_workers = max_workers
```

As expected, using processes is much faster than the threading based executor:

Example 2.11 Time and output of `futures-threads-worker`

```
$ time python futures-processes-worker.py
Results: [50485099, 50461662, 50553224, 50458097, 50520276, 50510314, ↩
    50510035, 50525335]
python futures-processes-worker.py 19.48s user 0.30s system 330% cpu ↩
    5.991 total
```

Warning

One important thing to notice with both of the pool based executors is the way they manage the processes and threads they spawn. There are several policies that the authors could have implemented. The one selected is that for each job submitted, a new worker is spawned to do the work, and the work is put in a queue shared across all the existing workers. That means that if the caller sets `max_workers` to 20, then 20 workers will exist as soon as 20 jobs are submitted. None of those processes will ever be destroyed. This is different than, for example, Apache httpd workers that exit after being idle for a while. You can see that this is marked as a TODO in Python source code as shown in Example 2.12

Example 2.12 Extract of `concurrent.futures.thread`

```
class ThreadPoolExecutor(_base.Executor):
    def submit(self, fn, *args, **kwargs):
        [...]
        self._adjust_thread_count()

```

```
    def _adjust_thread_count(self):
        [...]
        # TODO(bquinlan): Should avoid creating new threads if there are more
        # idle threads than items in the work queue.
        if len(self._threads) < self._max_workers:
            t = threading.Thread(target=_worker,
                                 args=(weakref.ref(self, weakref_cb),
                                       self._work_queue))
            t.daemon = True
            t.start()
            self._threads.add(t)
            _threads_queues[t] = self._work_queue
```

2.4 Advanced Futures Usage

As we have seen in Section 2.3, `Future` objects are an easy way to parallelize tasks in your application. The *futurist* library has been built on top of *concurrent.futures* and offers a few bonuses that I would like to introduce here. It is (almost) a transparent replacement for *concurrent.futures*, so any code should be straightforward in terms of adapting to this library, which is itself entirely based on *concurrent.futures*.

Example 2.13 Workers using `futurist.ThreadPoolExecutor`

```
import futurist
from futurist import waiters
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(10000)])


```

```
with futurist.ThreadPoolExecutor(max_workers=8) as executor:
    futures = [executor.submit(compute) for _ in range(8)]
    print(executor.statistics)

results = waiters.wait_for_all(futures)
print(executor.statistics)

print("Results: %s" % [r.result() for r in results.done])
```

Example 2.14 Output of `futures-threads-worker`

```
 $ python examples/futurist-threads-worker.py
<ExecutorStatistics object at 0x10b95b820 (failures=0, executed=0, runtime=0.00, cancelled=0)>
<ExecutorStatistics object at 0x10b95b820 (failures=0, executed=8, runtime=143.76, cancelled=0)>
Results: [50458683, 50479504, 50517520, 50510116, 50450298, 50510857, 50530137, 50511422]
```

First, *futurist* allows any application to access statistics about the executor it uses. That view is valuable for tracking the current status of your tasks and to report information on how the code runs.

futurist also allows passing a function and possibly denying any new job to be submitted by using the `check_and_reject` argument. This argument allows controlling the maximum size of the queue in order to avoid any memory overflow.

Example 2.15 Using `check_and_reject` to limit queue size

```
import futurist
from futurist import rejection
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(1000000)])
```

```
9
10
11 with futurist.ThreadPoolExecutor(
12        max_workers=8,
13        check_and_reject=rejection.reject_when_reached(2)) as executor:
14     futures = [executor.submit(compute) for _ in range(20)]
15     print(executor.statistics)
16
17 results = [f.result() for f in futures]
18 print(executor.statistics)
19
20 print("Results: %s" % results)
```

Depending on the speed of your computer, it is likely that Example 2.15 raises a `futurist.RejectedSubmission` exception because the executor is not fast enough to absorb the backlog, the size of which is limited to two. This example does not catch the exception – obviously, any decent program should handle that exception and either retry later, or raise a different exception to the caller.

futurist addresses a widespread use case with the `futurist.periodics.PeriodicWorker` class. It allows scheduling functions to run regularly, based on the system clock.

Example 2.16 Using `futurist.periodics`

```
1  import time
2
3  from futurist import periodics
4
5
6  @periodics.periodic(1)
7  def every_one(started_at):
8      print("1: %s" % (time.time() - started_at))
9
10
11 w = periodics.PeriodicWorker([
12     (every_one, (time.time(),), {}),
```

```
])


@periodics.periodic(4)
def print_stats():
    print("stats: %s" % list(w.iter_watchers()))


w.add(print_stats)
w.start()
```

Example 2.17 Output of `futurist-periodics.py`

```
$ python examples/futurist-periodics.py
1: 1.00364780426
1: 2.00827693939
1: 3.00964093208
stats: [<Watcher object at 0x1104fc790 (runs=3, successes=3, failures=0, elapsed=0.00, elapsed_waiting=0.00)>,
        <Watcher object at 0x1104fc810 (runs=0, successes=0, failures=0, elapsed=0.00, elapsed_waiting=0.00)>]
1: 4.00993490219
1: 5.01245594025
1: 6.01481294632
1: 7.0150718689
stats: [<Watcher object at 0x1104fc790 (runs=7, successes=7, failures=0, elapsed=0.00, elapsed_waiting=0.00)>,
        <Watcher object at 0x1104fc810 (runs=1, successes=1, failures=0, elapsed=0.00, elapsed_waiting=0.00)>]
1: 8.01587891579
1: 9.02099585533
[...]
```

Example 2.16 implements two tasks. One runs every second and prints the time elapsed since the start of the task. The second task runs every four seconds and prints statistics about the running of those tasks. Again here, *futurist* offers internal access to its statistics, which is very handy for reporting the status of the application.

While not necessary to depend on, *futurist* is a great improvement over `concurrent.futures` if you need fine grained control over the execution of your threads or processes.

2.5 Daemon Processes

Being aware of the difference between multithreading and multiprocessing in Python, it becomes more clear that using multiple processes to schedule different jobs is efficient. A widespread use case is to run long-running, background processes (often called daemons) that are responsible for scheduling some tasks regularly or processing jobs from a queue.

It could be possible to leverage `concurrent.futures` and a `ProcessPoolExecutor` to do that as discussed in Section 2.3. However, the pool does not provide any control regarding how it dispatches jobs. The same goes for using the `multiprocessing` module. They both make it hard to efficiently control the running of background tasks. Think of it as the "pets vs. cattle" analogy for processes.

In this section, I would like to introduce you to *Cotyledon*, a Python library designed to build long-running processes.

Example 2.18 Daemon using *Cotyledon*

```
import threading
import time

import cotyledon

```

```
6
7  class PrinterService(cotyledon.Service):
8      name = "printer"
9
10     def __init__(self, worker_id):
11         super(PrinterService, self).__init__(worker_id)
12         self._shutdown = threading.Event()
13
14     def run(self):
15         while not self._shutdown.is_set():
16             print("Doing stuff")
17             time.sleep(1)
18
19     def terminate(self):
20         self._shutdown.set()
21
22
23 # Create a manager
24 manager = cotyledon.ServiceManager()
25 # Add 2 PrinterService to run
26 manager.add(PrinterService, 2)
27 # Run all of that
28 manager.run()
```

Example 2.18 is a simple implementation of a daemon using *Cotyledon*. It creates a class named `PrinterService` that implements the needed method for `cotyledon.Service`: `run` which contains the main loop, and `terminate`, which is called by another thread when it terminates the service.

Cotyledon uses several threads internally (at least to handle signals), which is why the `threading.Event` object is used to synchronize the `run` and `terminate` methods.

This service does not do much; it simply prints the message `Doing stuff` every second. The service is started twice by passing two as the number of services to start to `manager.add`. That means *Cotyledon* starts two processes, each of them launching the `PrinterService.run` method.

When launching this program, you can run the `ps` command on your system – on Unix at least – to see what is running:

```
74476 ttys004  0:00.09 cotyledon-simple.py: master process [examples/ ↩
    cotyledon-simple.py]
74478 ttys004  0:00.00 cotyledon-simple.py: printer worker(0)
74479 ttys004  0:00.00 cotyledon-simple.py: printer worker(1)
```

Cotyledon runs a master process that is responsible for handling all of its children. It then starts the two instances of `PrinterService` as it was requested to launch. It also gives them nice shiny process names, making them easier to track in the long list of processes. If one of the processes gets killed or crashes, it is automatically relaunched by *Cotyledon*. The library does a lot behind the scenes, e.g., doing the `os.fork` calls and setting up the right modes for daemons.

Cotyledon also supports all operating systems supported by Python itself, avoiding the developer needing to have to think about operating system portability – which can be quite complex.

Example 2.18 is a simple scenario for independent workers – they can execute a job on their own, and they do not need to communicate with each other. This scenario is rare, as most services need to exchange between one another.

Example 2.19 shows an implementation of the common producer/consumer pattern. In this pattern, a service fills a queue (the producer) and other services (the consumers) consume the jobs to execute them.

Example 2.19 Producer/consumer using *Cotyledon*

```
import multiprocessing
import time

import cotyledon


```

```
class Manager(cotyledon.ServiceManager):
    def __init__(self):
        super(Manager, self).__init__()
        queue = multiprocessing.Manager().Queue()
        self.add(ProducerService, args=(queue,))
        self.add(PrinterService, args=(queue,), workers=2)


class ProducerService(cotyledon.Service):
    def __init__(self, worker_id, queue):
        super(ProducerService, self).__init__(worker_id)
        self.queue = queue

    def run(self):
        i = 0
        while True:
            self.queue.put(i)
            i += 1
            time.sleep(1)


class PrinterService(cotyledon.Service):
    name = "printer"

    def __init__(self, worker_id, queue):
        super(PrinterService, self).__init__(worker_id)
        self.queue = queue

    def run(self):
        while True:
            job = self.queue.get(block=True)
            print("I am Worker: %d PID: %d and I print %s"
                  % (self.worker_id, self.pid, job))


Manager().run()
```

The program in Example 2.19 implements a custom `cotyledon.ServiceManager` that is in charge for creating the queue object. This queue object is passed to all the services. The `ProducerService` uses that queue and fills it with an incremented integer every second, whereas the `PrinterService` instances consume from that queue and print its content.

When run, the program outputs the following:

```
I am Worker: 0 PID: 24727 and I print 0
I am Worker: 0 PID: 24727 and I print 1
I am Worker: 1 PID: 24728 and I print 2
I am Worker: 0 PID: 24727 and I print 3
```

The `multiprocessing.queues.Queue` object eases the communication between different processes. It is safe to use across threads and processes, as it leverages locks internally to guarantee data safety.

Note

If you are familiar with the *Go* programming language, this is the basic pattern that is used to implement the **Go routines** and their channels. That common and efficient pattern made the Go language very popular. In Go, forking new processes and passing messages between them is provided as a built-in element of the language. Providing syntactic sugar makes it quicker to write programs with this pattern. However, in the end, you can achieve the same thing in Python, though, with maybe, a little more effort.

Last, but not least, *Cotyledon* also offers a few more features, such as reloading the program configuration or changing the number of workers for a class dynamically.

Example 2.20 Reconfiguring the number of processes with *Cotyledon*

```
import multiprocessing
import time

import cotyledon


class Manager(cotyledon.ServiceManager):
    def __init__(self):
        super(Manager, self).__init__()
        queue = multiprocessing.Manager().Queue()
        self.add(ProducerService, args=(queue,))
        self.printer = self.add(PrinterService, args=(queue,), workers =2)
        self.register_hooks(on_reload=self.reload)

    def reload(self):
        print("Reloading")
        self.reconfigure(self.printer, 5)


class ProducerService(cotyledon.Service):
    def __init__(self, worker_id, queue):
        super(ProducerService, self).__init__(worker_id)
        self.queue = queue

    def run(self):
        i = 0
        while True:
            self.queue.put(i)
            i += 1
            time.sleep(1)


class PrinterService(cotyledon.Service):
    name = "printer"

```

```
    def __init__(self, worker_id, queue):
        super(PrinterService, self).__init__(worker_id)
        self.queue = queue

    def run(self):
        while True:
            job = self.queue.get(block=True)
            print("I am Worker: %d PID: %d and I print %s"
                  % (self.worker_id, self.pid, job))


Manager().run()
```

In Example 2.20, only two processes for `PrinterService` are started. As soon as `SIGHUP` is sent to the master process, *Cotyledon* calls the `Manager.reload` method that reconfigure the printer service to now have five processes. This is easy to check:

```
$ ps ax | grep cotyledon
55530 s002 S+    0:00.12 cotyledon-reconfigure.py: master process [ ↩
    examples/cotyledon-reconfigure.py]
55531 s002 S+    0:00.02 cotyledon-reconfigure.py: master process [ ↩
    examples/cotyledon-reconfigure.py]
55532 s002 S+    0:00.01 cotyledon-reconfigure.py: ProducerService  ↩
    worker(0)
55533 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(0)
55534 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(1)
$ kill -HUP 55530
$ ps ax | grep cotyledon
55530 s002 S+    0:00.27 cotyledon-reconfigure.py: master process [ ↩
    examples/cotyledon-reconfigure.py]
55531 s002 S+    0:00.03 cotyledon-reconfigure.py: master process [ ↩
    examples/cotyledon-reconfigure.py]
55551 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(2)
55553 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(3)
55554 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(4)
55555 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(1)
55557 s002 S+    0:00.01 cotyledon-reconfigure.py: ProducerService  ↩
```

```
    worker(0)
55558 s002 S+    0:00.01 cotyledon-reconfigure.py: printer worker(0)
```

Cotyledon is an excellent library for managing long-running processes. I encourage everyone to leverage it to build long-running, background, job workers.

2.6 Mehdi Abaakouk on CPU Scaling

Hey Mehdi! Could you start by introducing yourself and explaining how you came to Python?

Hi! I am Mehdi Abaakouk, I live in Toulouse (France), and I have been using Linux for almost twenty years.

My current job is Senior Software Engineer for Redhat. My main interests in computer sciences are open-source software and how the Internet works under the hood, and I like hacking both of them.

At the beginning of my using Linux, I was frustrated with the music players available at that time, so I started to write one. I looked at the code of many media players and wanted to use GTK/GStreamer toolkits. I first tried it in C by reusing some code from Rhythmbox, but I quickly abandoned that because of the slow progress I made

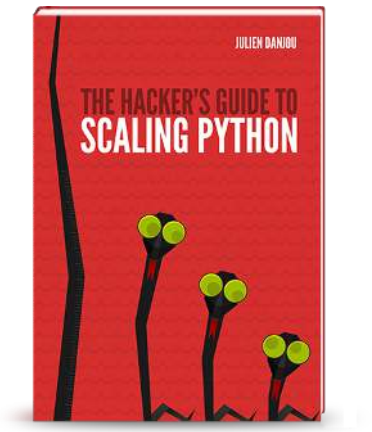
Hey, this was only a sample chapter!

I hope that you did like the sample! It includes the complete table of contents and a full chapter with its examples.

The full version of The Hacker's Guide to Scaling Python includes:

- 13 chapters
- 7 interviews
- 300 pages
- 80 code snippets
- Practical examples
- Available in PDF, HTML, EPUB and MOBI formats
- Available in a professionally printed paperback format
- And a few more bonuses such as Docker images!

Buy the Book!



Now that you've read the sample, you might be interested in buying the whole book. It's available online at scaling-python.com in different formats and packages. Go check it out!