فقط کتاب

مرجع معتبر دانلود کتاب های تخصصی

Faghatketab.ir

JULIEN DANJOU

THE HACKER'S GUIDE TO
SCALING PYTHON

The Hacker's Guide to Scaling Python

Julien Danjou

Julien Danjou
`<julien@danjou.info>`

Copyright © 2016-2017 Julien Danjou

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, "Thread-unsafe code without the GIL"](#) would not be thread-safe without the global Python lock.

Example 1.1. Thread-unsafe code without the GIL

```python
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 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, *CPU Scaling*](#). It is also a first step before spreading across a network.

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, Queue-Based Distribution*. Bigger problems such as coordination and synchronization with sibling nodes also have a few solutions, as discussed in *Chapter 7, Lock Management* and *Chapter 8, Group membership*.

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, Functional Programming* 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, *Scaling?*, 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, *Functional Programming*. 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. 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, Building REST API.

---

[1] Details about disassembling code and bytecode instruction are provided in Section 13.4, "Disassembling Code".

[2] The list is provided in the Python FAQ.

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

[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=("h
t.start()
```

```
print("thread started")
t.join()
```

If you run the program in <u>Example 2.1, "Starting a new thread"</u> 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-star
hellothread started
$ python examples/chapter2-cpu-scaling/threading-star
hello
 thread started
$ python examples/chapter2-cpu-scaling/threading-star
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=("h
t.daemon = True
t.start()
print("thread started")
```

In [Example 2.2, "Starting a new thread in daemon mode"](#) , 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(100000

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

Running [Example 2.3, "Workers using multithreading"](#) program returns the following:

```
$ time python multithreading-worker.py
Results: [50505811, 50471217, 50531481, 50460206, 50
python examples/multithreading-worker.py   19.84s use
```

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, "Using threads with CPython"](#) 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
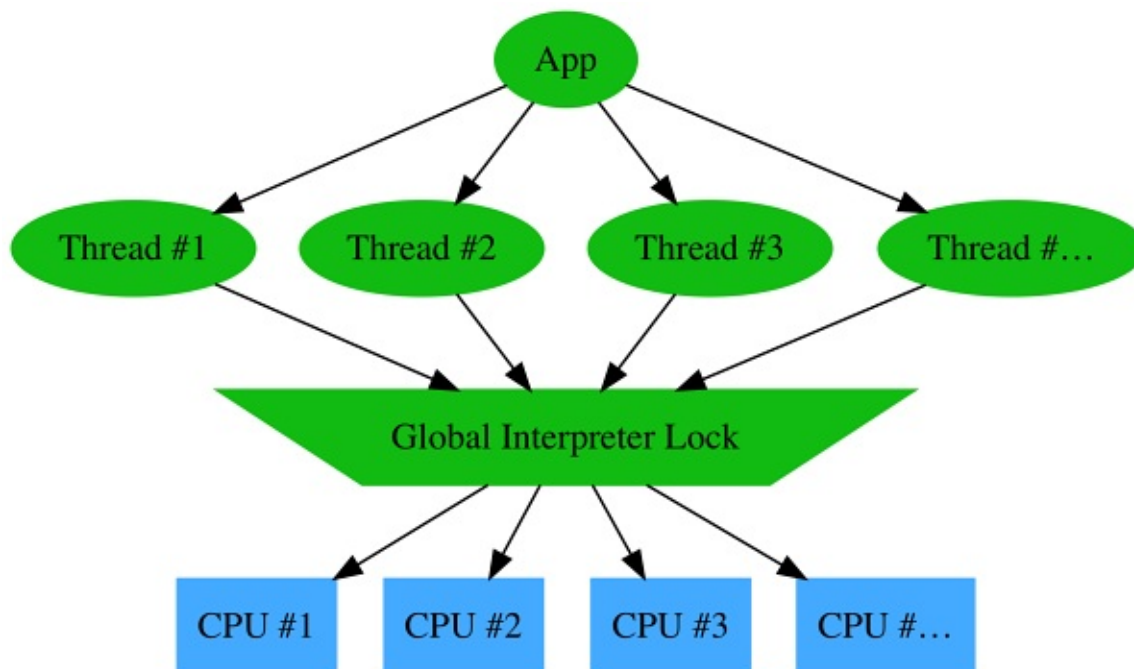
Again, as discussed in [Section 1.1, "Across CPUs"](#), the GIL limits the performance of *CPython* when executing multiple threads. Threads are therefore useful when doing parallel computing or input/output on slow

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, "Using Processes"](#).

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, "Workers using multithreading"](#) using processes thanks to the *multiprocessing* library, as shown in [Example 2.4, "`multiprocessing.Process` usage"](#).

Example 2.4. `multiprocessing.Process` usage

```
import random
import multiprocessing


def compute(results):
    results.append(sum(
        [random.randint(1, 100) for i in range(100000

with multiprocessing.Manager() as manager:
    results = manager.list()
    workers = [multiprocessing.Process(target=compute
               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, 505
python examples/multiprocessing-workers.py  32.00s us
```

Compared to <u>Example 2.3, "Workers using multithreading"</u>, 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 <u>Figure 2.2, "Using processes with CPython"</u> 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
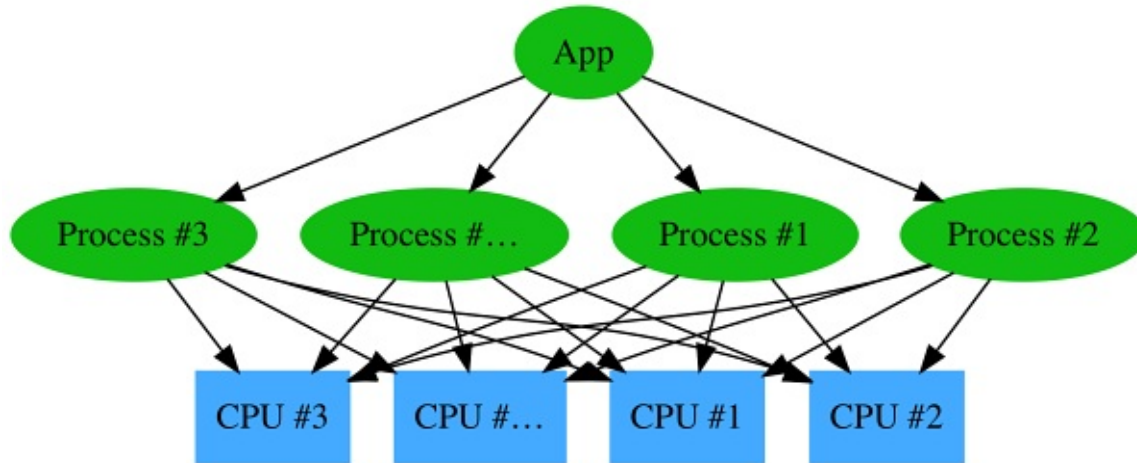
Each time some work can be **parallelized** for a certain amount of time, it's much better to rely on `multiprocessing` and to fork jobs, thiu spreading the workload among several CPU cores, rather than using the `threading` module.

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

Example 2.6. Worker using multiprocessing

```
import multiprocessing
import random


def compute(n):
    return sum(
        [random.randint(1, 100) for i in range(100000

# 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, "Using Futures"](#).

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, "Across CPUs"](#), 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`

```python
from concurrent import futures
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(100000

with futures.ThreadPoolExecutor(max_workers=8) as exe
    futs = [executor.submit(compute) for _ in range(8

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

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 <u>Chapter 4, *Functional Programming*</u>.

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, 505
python futures-threads-worker.py  14.50s user 6.91s s
```

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(100000


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

results = [f.result() for f in futs]
```

```
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,
"Extract of `concurrent.futures.process`"](#)

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_c
        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, 505
python futures-processes-worker.py  19.48s user 0.30s
```

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, "Extract of `concurrent.futures.thread`"

Example 2.12. Extract of `concurrent.futures.thread`

```python
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 t
        # idle threads than items in the work queue.
        if len(self._threads) < self._max_workers:
            t = threading.Thread(target=_worker,
                                 args=(weakref.ref(se
                                 self._work_que
            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, "Using Futures", 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`

```python
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 e>
    futs = [executor.submit(compute) for _ in range(8
    print(executor.statistics)

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

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

Example 2.14. Output of `futures-threads-worker`

```
 $ python examples/futurist-threads-worker.py
<ExecutorStatistics object at 0x10b95b820 (failures=(
<ExecutorStatistics object at 0x10b95b820 (failures=(
 Results: [50458683, 50479504, 50517520, 50510116, 50-
```

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

```python
import futurist
from futurist import rejection
import random


def compute():
    return sum(
        [random.randint(1, 100) for i in range(100000

with futurist.ThreadPoolExecutor(
        max_workers=8,
        check_and_reject=rejection.reject_when_reache
    futs = [executor.submit(compute) for _ in range(2
    print(executor.statistics)

results = [f.result() for f in futs]
print(executor.statistics)

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

Depending on the speed of your computer, it is likely that Example 2.15, "Using `check_and_reject` to limit queue size" 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`

```
import time

from futurist import periodics


@periodics.periodic(1)
def every_one(started_at):
    print("1: %s" % (time.time() - started_at))


w = periodics.PeriodicWorker([
    (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, succe
        <Watcher object at 0x1104fc810 (runs=0, succe
1: 4.00993490219
1: 5.01245594025
1: 6.01481294632
1: 7.0150718689
stats: [<Watcher object at 0x1104fc790 (runs=7, succe
        <Watcher object at 0x1104fc810 (runs=1, succe
```

```
1: 8.01587891579
1: 9.02099585533
[...]
```

[Example 2.16, "Using `futurist.periodics`"](#) 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, "Using Futures"](#). 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


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

    def __init__(self, worker_id):
        super(PrinterService, self).__init__(worker_
        self._shutdown = threading.Event()

    def run(self):
        while not self._shutdown.is_set():
            print("Doing stuff")
            time.sleep(1)

    def terminate(self):
        self._shutdown.set()


# Create a manager
manager = cotyledon.ServiceManager()
# Add 2 PrinterService to run
manager.add(PrinterService, 2)
# Run all of that
manager.run()
```

Example 2.18, "Daemon using *Cotyledon*" 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
74478 ttys004    0:00.00 cotyledon-simple.py: printer
74479 ttys004    0:00.00 cotyledon-simple.py: printer
```

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, "Daemon using *Cotyledon*" 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, "Producer/consumer using *Cotyledon*" 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,), worke


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

    def run(self):
        while True:
            job = self.queue.get(block=True)
```

```
            print("I am Worker: %d PID: %d and I prir
                % (self.worker_id, self.pid, job))



    Manager().run()
```

The program in [Example 2.19, "Producer/consumer using *Cotyledon*"](#) 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=
        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_
        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_:
        self.queue = queue

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


Manager().run()
```

In [Example 2.20, "Reconfiguring the number of processes with Cotyledon"](#), 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:
55531 s002   S+      0:00.02 cotyledon-reconfigure.py:
55532 s002   S+      0:00.01 cotyledon-reconfigure.py:
55533 s002   S+      0:00.01 cotyledon-reconfigure.py:
55534 s002   S+      0:00.01 cotyledon-reconfigure.py:
$ kill -HUP 55530
$ ps ax | grep cotyledon
55530 s002   S+      0:00.27 cotyledon-reconfigure.py:
55531 s002   S+      0:00.03 cotyledon-reconfigure.py:
55551 s002   S+      0:00.01 cotyledon-reconfigure.py:
55553 s002   S+      0:00.01 cotyledon-reconfigure.py:
```

```
55554 s002   S+      0:00.01 cotyledon-reconfigure.py:
55555 s002   S+      0:00.01 cotyledon-reconfigure.py:
55557 s002   S+      0:00.01 cotyledon-reconfigure.py:
55558 s002   S+      0:00.01 cotyledon-reconfigure.py:
```

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

each coding session done during my free time. At my day job,, I was more focusing on PHP and Java languages.

Then I discovered a new media player named Quodlibet that was completely written in Python with some awesome code to parse the metadata of media files. However, the UI, the playlists and songs manager were not up to my standards. So I tried to rewrite it, with my user interface ideas, but in Python this time. I was surprised by the quick progress I made. It was the first time I used Python, but it was so easy to learn! The online documentation and examples were so rich compared to other languages. The Python bindings for GTK/GStreamer were complete and easy to use too. I was able to build something workable very quickly and to focus on the features I wanted. What did surprised me with Python was how the language hid complicated computer things and how it was so concise. As a young developer, it helped me a lot. That is how the *Listen* media player was born. The project and its small community was alive for six years.

That was my first Python and open-source project; it grew very quickly. I learned many things about computers, open-source and Python itself because of this project.

After that, I wrote some small tools in Python every time I could for my new "System and Network Engineer" job to automate the team's work instead of doing the same tasks over and over again. Thanks to this experience and my Python background, I finally was employed to participate in the OpenStack project, a set of open-source cloud computing software. Now I can finally participate in Python projects every day with a good understanding of how computers and networks work, and not just how the language works.

Considering you worked on a cloud computing platform and created *Cotyledon*, I'm sure you have good advice about scaling using threads and processes. What would that be? What would be the mistakes and trap to not fall into?

Typical applications do either I/Os (reading and writing data) or do a calculation using the CPU:

- When an application does I/Os, it is constantly waiting for the operating system to process those streams and to return the result. During that time, the application is stuck.

- When an application does a calculation using the CPU, it can only use one of those CPUs: others CPUs are idle.

To improve this situation, you need to use threads and processes to execute these tasks in parallel.

My advice is this: do not use threads to manipulate objects or compute things with Python code. In that case, because of the GIL, it is indeed slow. However, for I/Os operations, you can use threads. Their biggest advantage is that they share the memory address space of the program, requiring less work for the CPUs to switch from one thread to another. They also help to share network connections between workers, so your program does not have to establish multiple connections to a server.

For CPU intensive tasks, use a Python library written in C or in [Cython](#) which allows manipulating your objects without being locked by the GIL (a perfect example is [NumPy](#)).

Using threads for concurrency forces the developer to pay attention

to shared resources. You should not write in the same socket or modify the same objects at the same time with two different threads. You can use locks to protect accesses to the resource. However, using locks has to be avoided as it just prevents threads from running in parallel efficiently, because they have to wait for the lock to be freed up.

You can also use processes to run many tasks at once. At first glance, that sounds faster because there's no GIL involved. However, processes do not share any memory space. That means your different processes must work stateless. That also means the CPUs will have more work to do to switch from one process to another. An application should not use too many processes. A rule of thumb, is good to have the number of processes equal to the number of CPUs available so that each process can stick to one CPU. That avoids doing context switching on the CPUs to run other processes, which is slow.

Another downside of processes is that there's a good chance they will multiply the number of open network connections. Sharing them across processes is highly complicated and error prone.

Threads cannot be spread across several computers, whereas process could be. That's especially true if the processes are stateless.

Also, having many processes doing the same thing in parallel means that you do not have to deal with their life-cycle. If one crashes, restart it. You can easily stop or restart all of them at once or propagate the signal handling between them. A good way to handle that is to create a process that only takes care of the life of the others and does nothing else (a scheduler).

With both solutions, you may have to deal with signals, and they are not concurrency friendly. Signals can be received at any time, freezing the code of a random thread of your process and running the signal callback within instead. If your signal callback does I/Os or a blocking operation, the process will switch to another thread, without unfreezing the code executed previously!

You could also receive a second signal in the meantime, or another thread could end the thread that is in charge of processing the signal. That is a nightmare to handle. It leads to race conditions that are hard to debug and fix.

If I had only one piece of advice to solve of all this: register a signal callback that does nothing. Instead, use the main thread of your application to create a file descriptor with `signal.set_wakeup_fd` and wait for CPython to write in the last signal number received. That removes all the complexity of signal handling.

Cotyledon is a library I wrote to easily manage the processes of workers. Just create a worker class to do the work and tell *Cotyledon* how many of them you want to run in parallel. It takes care of their life cycle, it also takes care of the signal handling and ensure the application callbacks are always run in the main thread without interrupting the code violently, as I described.

Finally, none of these solutions are exclusive. You can use threads to do I/Os concurrently and spawn many processes to maximize the CPU utilization. Any mix is possible. Generate some metrics with your application to see which part of your code waits for I/Os completion and where it uses CPU. That should help you to create

threads and processes at the correct places.

It sounds like running multiple processes is a good first step for scaling out Python horizontally on multiple nodes. So when one wants to do that, what's your favorite design pattern to write applications that can leverage that? What's your go-to library, solution, or technology to help in writing application for that?

I always start with the same scheme. I write one component to talk to the user, often a REST API within a WSGI application. Then I write other components to do the real job, often called workers, executing background jobs. Interactions between them must be asynchronous.

Next, I pick a technology to make the component interact with each other. Depending of the nature of the jobs that can be a database, message queues, or even both.

In case of multiple processes working on multiples nodes, it often makes sense to use a pub/sub mechanism based on a message queues system. I like to use Redis most of the times because it is easy to use and deploy with the python-redis or python-rq libraries. On a bigger scale, when the number of messages waiting to be processed is too high, or when you have many different kinds of producers/consumers, I would switch to Kafka, to partition the workload across different servers with confluent-kakfa-python or python-kafka libraries.

On the REST API side, I avoid waiting for the workers. The REST API just publishes messages to the workers and returns the fact that the jobs have been accepted. I prefer the client to poll the REST API

to get the status of the jobs regularly, instead of having a request waiting on another part of the infrastructure. This avoids long lived requests that add timeout everywhere. Moreover, since each part works independently, it simplifies a lot of investigation during root cause analysis.

Even better, I like to produce an event stream that can be consumed by the client. For example, when the client is a browser, the Javascript *EventSource* API allows subscribing to an HTTP event stream easily. On the server side, using Flask, a stream is just a Python generator that yields a JSON, with a response mime type set to `text/event-stream`. To consume this stream from the REST application, I use Redis again: the workers publish the jobs statuses on queues, if an HTTP stream request has subscribed to one of them, it is yielded by the stream generator and sent back to the browser via the EventSource API. No polling at all: just a bunch of HTTP connections, idle most of the time.

I would also recommend writing a dedicated WSGI application for the event stream REST endpoint in such cases – it often needs some specific tuning to allow many idle open connections in parallel that must be kept open for a long time. The REST API should not allow wasting resources.

To finish, if you like all these events based workflows, the new asyncio Python API fits perfectly in the event stream use case. Instead of writing a blocking WSGI application with flask/python-redis, you can switch to aiohttp and aioredis just for this REST endpoint.

Thank you Mehdi!

You're welcome!

Chapter 3. Event Loops

If event loops do not ring a bell, maybe you've heard of them under another name, such as *message dispatching*. An event loop is a central control flow of a program where messages are pushed into a queue, and this queue is consumed by the event loop, dispatching them to appropriate functions.

A very simplistic form of event loop would be something like this:

```
while True: message = get_message() if message == qui
    process_message(message)
```

Each time a message (which can also be called an event) is received, it is processed – until a final message is received to make the program quit.

This design is quite efficacious as it allows the program to wait and do nothing (as long as `get_message` is blocking) and waking up as soon as a message is received.

As long as there is only one source of message, this program can be quite trivial to write. Consuming from a single queue is precisely what distributed programs would do based on queues, as described in [Chapter 5, *Queue-Based Distribution*](#).

However, most programs do not have a single source of messages and events. Moreover, dealing with any event might generate new sources of events, of different types. That means that the program has to handle different source of events.

3.1. Basic Pattern

The most used source of events is I/O readiness. Most read and write operations are blocking in nature by default, slowing down the program execution speed. If the program has to wait several seconds for a `read` to

be completed, it cannot do anything else during that time. `read` is a synchronous call and when performed on a file, socket, etc., that has no data ready to be read, it blocks the program.

The solution to that problem is to expect an event when the socket, for example, is ready to be read. While this is not the case, the program can deal with any other event that might happen.

Example 3.1, "A blocking socket" is a simple program that sends an HTTP request to `http://httpbin.org/delay/5`. The URL returns a JSON content after five seconds of delay.

Example 3.1. A blocking socket

```
import socket

s = socket.create_connection(("httpbin.org", 80))
s.send(b"GET /delay/5 HTTP/1.1\r\nHost: httpbin.org\r
buf = s.recv(1024)
print(buf)
```

As expected, when this program runs, it takes at least five seconds to complete: the `socket.recv` call hangs until the remote Web server sends the reply.

This is the kind of situation that should be avoided: waiting for an input or output to complete before going on, as the program could be doing something else rather than waiting.

The solution here is to put the socket in *asynchronous mode*. This can be done using the `setblocking` method.

Example 3.2. A non-blocking socket

```
import socket

s = socket.create_connection(("httpbin.org", 80))
s.setblocking(False)
```

```
s.send(b"GET /delay/5 HTTP/1.1\r\nHost: httpbin.org\i
buf = s.recv(1024)
print(buf)
```

Running the program in [Example 3.2, "A non-blocking socket"](#) will fail with an interesting error:

```
Traceback (most recent call last):
  File "examples/socket-non-blocking.py", line 6, in
    buf = s.recv(1024)
BlockingIOError: [Errno 35] Resource temporarily unav
```

As the socket does not have any data to be read, rather than blocking until it has, Python raises a `BlockingIOError`, asking the caller to retry at a later time.

At this point, you can see where this is going. If the program can get a message as soon as the socket is ready to be handled, the code can do something else rather than actively waiting.

The simplest mechanism to do that in Python is to use the `select` module. This module provides `select.select(rlist, wlist, xlist)` function that takes any number of sockets (or file descriptors) as input and returns the ones that are ready to read, write or have errors.

Example 3.3. Using `select.select` with sockets

```
import select
import socket

s = socket.create_connection(("httpbin.org", 80))
s.setblocking(False)
s.send(b"GET /delay/1 HTTP/1.1\r\nHost: httpbin.org\i
while True:
    ready_to_read, ready_to_write, in_error = select.
        [s], [], [])
```

```
    if s in ready_to_read:
        buf = s.recv(1024)
        print(buf)
        break
```

In Example 3.3, "Using `select.select` with sockets", the socket is passed as an argument in the list of descriptors we want to watch for read-readiness. As soon as the socket has data available to read, the program can read them without blocking.

If you combine multiple sources of events in a `select` call, it is easy to see how your program can become *event-driven*. The `select` loop becomes the main control flow of the program, and everything revolves around it. As soon as some file descriptor or socket is available for reading or writing, it is possible to continue operating on it.

This kind of mechanism is at the heart of any program that wants to handle, for example, thousands of connections at once. It is the base technology leveraged by tools such as really fast HTTP servers like NGINX or Node.js.

`select` is an old but generic system call, and it is not the most well performing out there. Different operating systems implement various alternative and optimizations, such as `epoll` in Linux or `kqueue` in FreeBSD. As Python is a high-level language, it implements and provides an abstraction layer known as *asyncio*.

3.2. Using Asyncio

Now that you have a good idea of what an event loop is and of what it provides, it is a good time to dig into the use of a state-of-the-art event loop. Asyncio is new in Python 3, and it requires Python 3.5 or later to use it as described in this section.

Asyncio is centered on the concept of event loops, which work in the same way as the `select` module described in Section 3.1, "Basic Pattern".

Once asyncio has created an event loop, an application registers the functions to call back when a specific event happens: as time passes, a file descriptor is ready to be read, or a socket is ready to be written.

That type of function is called a *coroutine*. It is a particular type of function that can give back control to the caller so that the event loop can continue running. It works in the same manner than a generator would, giving back the control to a caller using the `yield` statement.

Example 3.4. Hello world asyncio coroutine

```
import asyncio


async def hello_world():
    print("hello world!")
    return 42

hello_world_coroutine = hello_world()
print(hello_world_coroutine)

event_loop = asyncio.get_event_loop()
try:
    print("entering event loop")
    result = event_loop.run_until_complete(hello_worl
    print(result)
finally:
    event_loop.close()
```

The example in [Example 3.4, "Hello world asyncio coroutine"](#) shows a very straightforward implementation of an event loop using a coroutine. The coroutine `hello_world` is defined as a function, except that the keyword to start its definition is `async def` rather than just `def`. This coroutine just prints a message and return a result.

The event loop runs this coroutine and is terminated as soon as the coroutine returns, ending the program. Coroutines can return values, and

in this case, the value 42 is returned by the coroutine, which is then returned by the event loop itself.

Example 3.5. Output of `asyncio-basic.py`

```
$ python3 examples/asyncio-basic.py
<coroutine object hello_world at 0x107dad518>
entering event loop
hello world!
42
```

Coroutines can cooperate – this is why they are named coroutines after all. You can, therefore, call a coroutine from a coroutine.

Example 3.6. Coroutine awaiting on coroutine

```
import asyncio


async def add_42(number):
    print("Adding 42")
    return 42 + number


async def hello_world():
    print("hello world!")
    result = await add_42(23)
    return result

event_loop = asyncio.get_event_loop()
try:
    result = event_loop.run_until_complete(hello_worl
    print(result)
finally:
    event_loop.close()
```

The `await` keyword is used to run the coroutine cooperatively. `await`

gives the control back to the event loop, registering the coroutine `add_42(23)` into it. The event loop can, therefore, schedule whatever it needs to run. In this case, only `add_42(23)` is ready and waiting to be executed: therefore it is one that gets executed. Once finished, the execution of the `hello_world` coroutine can be resumed by the event loop scheduler.

Example 3.7. Output of `asyncio-coroutines.py`

```
$ python3 examples/asyncio-coroutines.py
hello world!
Adding 42
65
```

The examples so far are pretty straightforward, and it is easy to guess in which order the various coroutines are executed. Those programs were barely leveraging event loop scheduling.

The example in [Example 3.8, "Coroutine using `asyncio.sleep` and `asyncio.gather.`](#) introduces two new functions:

- `asyncio.sleep` is the asynchronous implementation of `time.sleep`. It is a coroutine that sleeps some number of seconds. Since it is a coroutine and not a function, it can be used to yield back the control to the event loop.

- `asyncio.gather` allows to wait for several coroutines at once using a single `await` keyword. Rather than using several sequential `await` keywords, this allows explicitly stating to the scheduler that all the results of those operations are needed to continue the execution of the program. It makes sure the event loop executes those coroutines concurrently.

Example 3.8. Coroutine using `asyncio.sleep` and `asyncio.gather`.

```
import asyncio
```

```
async def hello_world():
    print("hello world!")


async def hello_python():
    print("hello Python!")
    await asyncio.sleep(0.1)


event_loop = asyncio.get_event_loop()
try:
    result = event_loop.run_until_complete(asyncio.ga
        hello_world(),
        hello_python(),
    ))
    print(result)
finally:
    event_loop.close()
```

In Example 3.8, "Coroutine using `asyncio.sleep` and `asyncio.gather`.", both the `hello_world` and `hello_python` coroutines are executed concurrently. If the event loop scheduler starts with `hello_world`, it will be able to continue only afterwards with `hello_python`. `hello_python` will then inform the scheduler that it needs to wait for the `asyncio.sleep(0.1)` coroutine to complete. The scheduler will execute this coroutine, which makes take into account a 0.1 second delay before giving back the execution to `hello_python`. Once it does that, the coroutine is finished, terminating the event loop.

Contrary to the classic `time.sleep` function, which makes Python sleep synchronously and blocks it while it waits, `asyncio.sleep` can be handled asynchronously, so Python can do something else while it waits for the specified delay to pass.

In the case where `hello_python` is the first coroutine to run, the execution of `hello_world` only starts after that the `await asyncio.sleep(0.1)` coroutine is yielded back to the event loop. Then the rest of the event loop continues its execution and terminates.

The `aiohttp` library provides an asynchronous HTTP – it is also covered in [Section 9.5, "Fast HTTP Client"](). The example in [Example 3.9, "Using aiohttp"]() is a simple example of leveraging *asyncio* for concurrency.

Example 3.9. Using *aiohttp*

```
import aiohttp
import asyncio


async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return response


loop = asyncio.get_event_loop()

coroutines = [get("http://example.com") for _ in rang

results = loop.run_until_complete(asyncio.gather(*co

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

This example creates several coroutines, one for each call to `get`. Those coroutines are then gathered, and so they are executed concurrently by the event loop. If the remote Web server is far away and needs a long delay to reply, the event loop switches to the next coroutine that is ready to be resumed, making sure the connections that are ready to be read are read.

Note

The `async with` keyword used in Example 3.9, "Using *aiohttp*" is equivalent to the `await` keyword, but is specific to context managers that use `await` in their `__enter__` and `__exit__` methods.

Asyncio also provides a way to call functions at a later time. Rather than building a waiting loop with `asyncio.sleep`, the methods `call_later` and `call_at` can be used to call functions at a relative or absolute future time respectively.

Example 3.10. Using *loop.call_later*

```
import asyncio


def hello_world():
    print("Hello world!")


loop = asyncio.get_event_loop()
loop.call_later(1, hello_world)
loop.run_forever()
```

The example in Example 3.10, "Using *loop.call_later*" prints "Hello world!" one second after starting and then blocks forever as the loop has nothing else to do.

Example 3.11. Using *loop.call_later* repeatedly

```
import asyncio


loop = asyncio.get_event_loop()


def hello_world():
    loop.call_later(1, hello_world)
```

```
    print("Hello world!")


loop = asyncio.get_event_loop()
loop.call_later(1, hello_world)
loop.run_forever()
```

The example in Example 3.11, "Using *loop.call_later* repeatedly" provides a `hello_world` function that leverages the `call_later` method to reschedule itself every second.

Asyncio is excellent at handling network related tasks. Its event loop can handle thousands of concurrent sockets, and therefore connections, switching to the one that is ready to be processed as soon as possible – as long as your program regularly yields back control the event loop using the `await` keyword. It obviously means programs need to use code and library that are *asyncio* compatible , which are not always widely available for all sorts of things.

3.2.1. Network Server

Since *asyncio* is exceptional at handling thousands of network connections, it provides a useful framework for implementing network servers. The following example implements a simplistic TCP server, but it is up to you to build any network server you might like.

Note

It is possible to build an *asyncio* based Web server using *aiohttp*. However, it is not that useful in most cases because it will often be slower than a fast, optimized, native WSGI server like *uwsgi* or *gunicorn*. As Python Web applications are always using WSGI, it is easy to switch out the WSGI server for a fast and asynchronous one.

Example 3.12, "asyncio TCP server" is a simple example of a TCP server. This server listens for strings terminated by `\n` and returns them in

upper case.

Example 3.12. asyncio TCP server

```
import asyncio

SERVER_ADDRESS = ('0.0.0.0', 1234)


class YellEchoServer(asyncio.Protocol):
    def connection_made(self, transport):
        self.transport = transport
        print("Connection received from:",
                transport.get_extra_info('peername'))

    def data_received(self, data):
        self.transport.write(data.upper())

    def connection_lost(self, exc):
        print("Client disconnected")


event_loop = asyncio.get_event_loop()

factory = event_loop.create_server(YellEchoServer, *S
server = event_loop.run_until_complete(factory)

try:
    event_loop.run_forever()
finally:
    server.close()
    event_loop.run_until_complete(server.wait_closed
    event_loop.close()
```

To implement a server, the first step is to define a class that inherits from
`asyncio.Protocol`. It is not strictly necessary to inherit from this
class, but it is a good idea to get all the basic methods defined – even if

they do nothing.

The `connection_made(transport)` method is called as soon as a connection is established by a client. An `asyncio.BaseTransport` object is passed as the argument, which represents the underlying socket and stream with the client. This object offers several methods such as `get_extra_info` to get more information on the client, or `close`, to close the transport. The `connection_lost` is the other end of the connection handling code: it is called when the connection is terminated. Both `connection_made` and `connection_lost` are called once per connection.

The `data_received` method is called each time some data is received. It might never be called if no data is ever received. The `eof_received` method might be called once if the client sent an `EOF` signal.

Figure 3.1, "`asyncio.Protocol` state machine" describes the state machine and the usual workflow that `asyncio.Protocol` provides.

Figure 3.1. `asyncio.Protocol` state machine
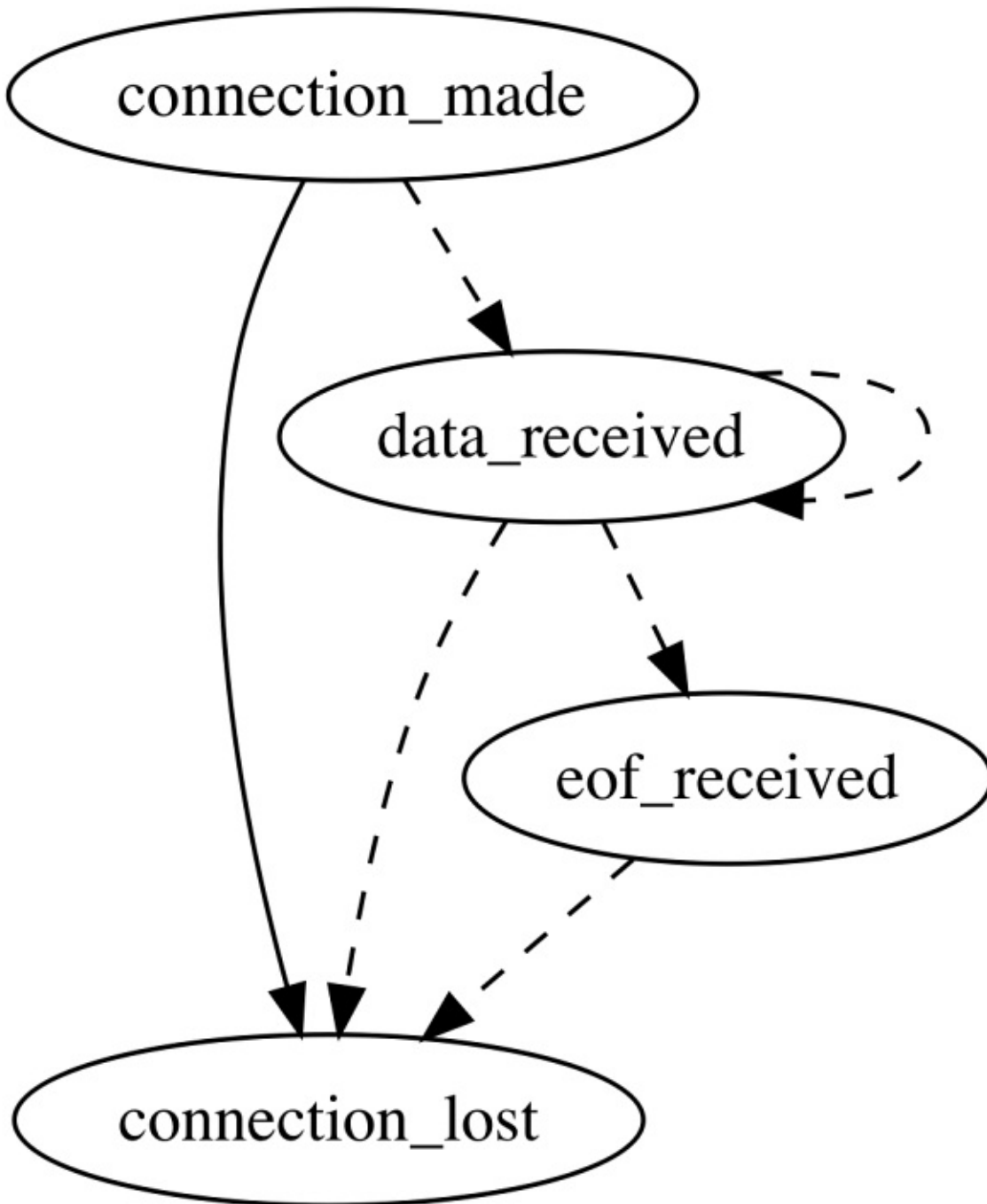
The simplest way to test Example 3.12, "asyncio TCP server" is to use the *netcat* program, available as the `nc` command on most versions of Unix.

Example 3.13. Using `nc` to connect to the `YellEchoServer`

```
$ nc localhost 1234
hello world!
HELLO WORLD!
^D
$
```

Typing any text followed by `\n` returns it in upper case. Sending `EOF` by pressing `Control+d` closes the connection.

We have been selling *asyncio* as extremely fast, so it is time to prove that point. The same `asyncio.Protocol` class can be used to implement a client.

Example 3.14. asyncio TCP client

```
import asyncio

SERVER_ADDRESS = ('0.0.0.0', 1234)


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def talk(self):
        self.transport.write(self.message)

    def connection_made(self, transport):
        self.transport = transport
        self.talk()

    def data_received(self, data):
        self.talk()

    def connection_lost(self, exc):
```

```
        self.loop.stop()


loop = asyncio.get_event_loop()
loop.run_until_complete(loop.create_connection(
    lambda: EchoClientProtocol(b'Hello World!', loop)
    *SERVER_ADDRESS))
try:
    loop.run_forever()
finally:
    loop.close()
```

The client in Example 3.14, "asyncio TCP client" connects to the
YellEchoServer. Once connected, `connection_made` is called and the
client sends its message via its `talk` method. The server replies back with
that text in upper case, and the `data_received` is called. In this case,
the client talks again to the server, creating an infinite loop of interaction
between the two.

This is useful to test the performance of the server. By modifying some
bits of Example 3.12, "asyncio TCP server" and adding some statistics, we
can have a rough idea of the amount our server can handle.

Example 3.15. asyncio TCP server with statistics

```
import asyncio
import time

SERVER_ADDRESS = ('0.0.0.0', 1234)


class YellEchoServer(asyncio.Protocol):
    def __init__(self, stats):
        self.stats = stats
        self.stats['started at'] = time.time()

    def connection_made(self, transport):
```

```
        self.transport = transport
        self.stats['connections'] += 1

    def data_received(self, data):
        self.transport.write(data.upper())
        self.stats['messages sent'] += 1


event_loop = asyncio.get_event_loop()

stats = {
    "connections": 0,
    "messages sent": 0,
}

factory = event_loop.create_server(
    lambda: YellEchoServer(stats), *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)

try:
    event_loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    server.close()
    event_loop.run_until_complete(server.wait_closed
    event_loop.close()

    ran_for = time.time() - stats['started at']
    print("Server ran for: %.2f seconds" % ran_for)
    print("Connections: %d" % stats['connections'])
    print("Messages sent: %d" % stats['messages sent'
    print("Messages sent per second: %.2f"
          % (stats['messages sent'] / ran_for))
```

In [Example 3.15, "asyncio TCP server with statistics"](), basic statistics are stored, computed, and printed at the end of the program.

Example 3.16. Testing the asyncio TCP echo server

```
$ python3 examples/asyncio-server-stats.py
^C
Server ran for: 71.48 seconds
Connections: 5
Messages sent: 1656928
Messages sent per second: 23178.68
```

Run on my laptop with five clients at the same time, the asyncio server is able to handle more than 23,000 messages per second. Obviously, this server is not doing much work – upper-casing a string is not that impressive – but this is still a pretty decent result. Keep in mind that this server is not using any thread or any extra processes, so it is only using **one single CPU**.

Asyncio is a transcendent solution to write asynchronous network clients and servers. The protocol implementation is straightforward, and the ability to mix all kinds of asynchronous workload makes the framework powerful.

3.3. Naoki Inada on asyncio

Hey Naoki! Could you introduce yourself and explain how you came to Python?

Hi! I have been a core Python developer since 2016. I implemented the ["compact ordered dict" feature in Python 3.6](#), which makes the ordered dictionary smaller in memory. I work on real-time online

game servers at [KLab Inc](). Thanks to my job, I can use most of my paid-time to work on open-source software, including CPython.

When I was a student, I did not like scripting languages like Python. I was way more into C++. I studied Perl and bash, but features like implicit type conversion between string and integer drew me away.

After graduation, my first job was to develop the firmware of digital still camera. C was the primary language used there, though using C made me spend way too much time on very simple tasks.

For example, analyzing and visualizing log data written in a custom format would require writing tons of long and tedious C code. Therefore I looked into scripting languages to improve my productivity. I stumbled upon IPython and found it very good for analytical tasks. And writing Python is way better than writing Windows `.bat` files!

I have read that you worked on *asyncio*. Can you brief us on the state of the project right now?

Well, back in 2016, you could have said that Tornado + PyPy2 was faster than asyncio. However, since then, asyncio increased its performances due to various improvement that have been made:

- It uses the new "yield from" keyword.
- Yury Selivanov implemented [uvloop]() and [asyncpg](). [Cython]() also provides more performance for the asyncio stack.
- The `async` and `await` keyword in CPython 3.6 provides more readability for asyncio applications.
- C implementation of `Future` and `Task` in CPython 3.6 increase performances of asyncio based applications (I implemented that

with Yury).

- Tornado supports asyncio correctly. Using the asyncio library from Tornado is now possible.

- PyPy3.5 was released recently, and while it is still in a beta stage, asyncio can run on top of PyPy now. The performance gains from using PyPy are high.

Therefore I think asyncio is now the practical approach for writing high performance async applications in Python.

Personally, I use golang when building high-performance network applications. However, I hope that asyncio will become the new standard for writing network libraries. An asyncio-based library (e.g. http2 client) can be used from (a) asyncio applications, (b) Tornado applications, and (c) normal synchronous applications.

At this point, there's no more reason not to embrace it.

What are the reasons that make you prefer Go rather than Python to write extremely high performance applications? Do you see Python being able to catch up with Go on those points?

First of all, Go is a different language and has a different culture (e.g., they use tabulations to indent). However, it also has the same Zen. I think Go is a language that is easy to learn and to maintain, similar to Python.

However, Go is usually faster than Python because it is a static language. Cython might be the final weapon for CPython, but I feel that Go code is more maintainable than large Cython codebases. Also, Go performance is more predictable than PyPy performance.

Another reason is that Go supports parallelism. Go can use multiple cores in one process. It makes it possible to use more efficient techniques than multi-processes architecture, as it is done in Python usually. For example, in-process caching is going to be more efficient in single-process architecture.

My last reason is that Go applications are easier to maintain than their asynchronous equivalent in Python.

For example, there are many "unlikely blocking" syscalls. They are not blocking when testing on your laptop, but they will sometimes block in production environments. Debugging such issues is a very tough job. The natural solution is to move all the possibly blocking functions to a dedicated thread. However, that makes the application more complicated and harder to maintain. To solve that problem, other languages such as C# and node.js also have an implementation of asynchronicity using a thread pool.

Goroutines are unique here. Goroutines run in threads. When a goroutine makes a call using a syscall and is blocked, Go runtime starts a new thread and moves the waiting goroutines from the old thread to this new thread. Therefore, blocking syscalls do not block the entire application. You can write an entire application with a single programming model, instead of mixing the asynchronous model and threaded model like you would have to do in Python.

The Go profiling tool, pprof, is also helpful. You can use it for the memory, CPU usage, and blocking profile. It makes it easy to maintain applications written in Go.

Do you see Python being able to catch up with Go on those

points?

I do not think Python can catch up 100% in terms of Go's strong points.

Of course, it is not a reason to stop optimizing both Python and asyncio. There are many areas where Python is more productive than static languages.

For example, you can write large web applications with Python, and use Go to write some middleware or microservices that will be used by the application behind. Asyncio can be used to call such microservices from a Python application.

So combining both languages can also create original and great solutions!

Chapter 4. Functional Programming

Functional programming might not be the first thing you think of when you think of Python, but the support is there, and it is quite extensive. Nonetheless, many Python developers do not seem to realize this, which is a shame: with few exceptions, functional programming allows you to write more concise and efficient code.

When you write code using a functional style, your functions are designed to not have side effects: they take an input and produce an output without maintaining state or modifying anything not reflected in the return value. Functions that adhere to this ideal are referred to as *purely functional*:

A non-pure function.

```python
def remove_last_item(mylist):
    """Removes the last item from a list."""
    mylist.pop(-1)  # This modifies mylist
```

A pure function.

```python
def butlast(mylist):
    """Like butlast in Lisp; returns the list without
    return mylist[:-1]  # This returns a copy of myl:
```

Python is not strictly functional, but it allows you to use that approach as long as you are rigorous. Apparently, the object-oriented model is not very compatible with such a functional approach as changing an object state is the opposite of being functional, where immutability is the rule.

The functional approach is even more important when talking about scalability. It has a significant property that helps a in terms of lot building large-scale applications: it has no shared state.

As discussed in [Section 1.2, "Distributed Systems"](#), not having a shared-state allows entire parts of your program to be distributed, as they are designed as black boxes, taking data in and outputting new data. Highly functional and distributed programming languages such as [Erlang](#) rely on this property as a core of their language to provide scalability as a first-class citizen in their programming experience.

As was also discussed in [Section 2.5, "Daemon Processes"](#), this approach likewise enables you to use other distributed patterns, such as queue and message passing, in the same manner, that the [Go programming languages](#) leverages channels to load balance tasks to execute.

There are other practical advantages of functional programming, such as:

- **Formal provability** Admittedly, this is a pure theoretical advantage as nobody is going to mathematically prove a Python program.
- **Modularity** Writing functionally forces a certain degree of separation in solving your problems and eases reuse in other contexts.
- **Brevity** Functional programming is often less verbose than other paradigms.
- **Concurrency** Purely functional functions are thread-safe and can run concurrently. While it is not yet the case in Python, some functional languages do this automatically, which can be a big help if you ever need to scale your application.
- **Testability** It is a simple matter to test a functional program, in that, all you need is a set of inputs and an expected set of outputs. They are idempotent.

Tip

If you want to get serious about functional programming, take my advice: take a break from Python and learn a more functional language such as Lisp. I know it might sound strange to talk about Lisp in a Python book, but playing with Lisp for several years is what taught me how to "think functional." You simply won't develop the thought processes necessary to make full use of

functional programming if all your experience comes from imperative and object-oriented programming. Lisp is not *purely* functional itself, but there's more focus on functional programming than you will find in Python.

4.1. The Functional Toolkit

While writing functional code will be your responsibility, Python also includes a number of tools for functional programming. These built-in functions cover the basics, and they should help you write better code:

- `map(function, iterable)` applies `function` to each item in `iterable` and returns either a list in Python 2 or an iterable `map` object in Python 3:

```
>>> map(lambda x: x + "bzz!", ["I think", "I'm good"]
<map object at 0x7fe7101abdd0>
>>> list(map(lambda x: x + "bzz!", ["I think", "I'm g
['I thinkbzz!', "I'm goodbzz!"]
```

- `filter(function or None, iterable)` filters the items in `iterable` based on the result returned by `function`, and returns either a list in Python 2 or better, an iterable `filter` object in Python 3:

```
>>> filter(lambda x: x.startswith("I "), ["I think",
<filter object at 0x7f9a0d636dd0>
>>> list(filter(lambda x: x.startswith("I "), ["I thi
['I think']
```

Tip

You can write a function equivalent to `filter` or `map` using generators and list comprehension:

Equivalent of `map` using list comprehension.

```
>>> (x + "bzz!" for x in ["I think", "I'm good"])
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x + "bzz!" for x in ["I think", "I'm good"]]
['I thinkbzz!', "I'm goodbzz!"]
```

Equivalent of `filter` using list comprehension.

```
>>> (x for x in ["I think", "I'm good"] if x.startsw:
<generator object <genexpr> at 0x7f9a0d697dc0>
>>> [x for x in ["I think", "I'm good"] if x.startsw:
['I think']
```

Using generators like this in Python 2 gives you an iterable object rather than a list, just like the `map` and `filter` functions in Python 3.

- `enumerate(iterable[, start])` returns an iterable `enumerate` object that yields a sequence of tuples, each consisting of an integer index (starting with `start`, if provided) and the corresponding item in `iterable`. It is useful when you need to write code that refers to array indexes. For example, instead of writing this:

```
i = 0
while i < len(mylist):
    print("Item %d: %s" % (i, mylist[i]))
    i += 1
```

You could write this:

```
for i, item in enumerate(mylist):
    print("Item %d: %s" % (i, item))
```

- `sorted(iterable, key=None, reverse=False)` returns a sorted version of `iterable`. The `key` argument allows you to provide a function that returns the value to sort on.

- `any(iterable)` and `all(iterable)` both return a Boolean depending on the values returned by `iterable`. These functions are equivalent to:

```python
def all(iterable):
    for x in iterable:
        if not x:
            return False
    return True
```

```python
def any(iterable):
    for x in iterable:
        if x:
            return True
    return False
```

These functions are useful for checking whether any or all of the values in an iterable satisfy a given condition:

```python
mylist = [0, 1, 3, -1]
if all(map(lambda x: x > 0, mylist)):
    print("All items are greater than 0")
if any(map(lambda x: x > 0, mylist)):
    print("At least one item is greater than 0")
```

- `zip(iter1 [,iter2 [...]])` takes multiple sequences and combines them into tuples. It is useful when you need to combine a list of keys and a list of values into a dictionary. Like the other functions described above, it returns a list in Python 2 and an iterable in Python 3:

```
>>> keys = ["foobar", "barzz", "ba!"]
>>> map(len, keys)
<map object at 0x7fc1686100d0>
>>> zip(keys, map(len, keys))
<zip object at 0x7fc16860d440>
>>> list(zip(keys, map(len, keys)))
[('foobar', 6), ('barzz', 5), ('ba!', 3)]
>>> dict(zip(keys, map(len, keys)))
{'foobar': 6, 'barzz': 5, 'ba!': 3}
```

Tip

You might have noticed that the return types differ between Python 2 and Python 3. Most of Python's purely functional built-in functions return a list rather than an iterable in Python 2, making them less memory-efficient than their Python 3.x equivalents. If you are planning to write code using these functions, keep in mind that you will get the most benefit out of them in Python 3. If you are stuck with Python 2, don't despair yet: the `itertools` module from the standard library provides an iterator-based version of many of these functions (`itertools.izip`, `itertools.imap`, `itertools.ifilter`, etc.).

There's still one essential tool missing from this list, however. One common task when working with lists is finding the first item that satisfies a specific condition. This is usually accomplished with a function like this:

```
def first_positive_number(numbers):
    for n in numbers:
        if n > 0:
            return n
```

We can also write this in functional style:

```
def first(predicate, items):
    for item in items:
        if predicate(item):
            return item

first(lambda x: x > 0, [-1, 0, 1, 2])
```

Or more concisely:

```
# Less efficient
list(filter(lambda x: x > 0, [-1, 0, 1, 2]))[0] (1)
# Efficient but for Python 3
next(filter(lambda x: x > 0, [-1, 0, 1, 2]))
# Efficient but for Python 2
next(itertools.ifilter(lambda x: x > 0, [-1, 0, 1, 2]
```

(1) Note that this may elicit an `IndexError` if no items satisfy the condition, causing `list(filter())` to return an empty list.

For simple case you can also rely on `next`:

```
>>> a = range(10)
>>> next(x for x in a if x > 3)
4
```

This will raise `StopIteration` if a condition can never be satisfied, so in that case the second argument of `next` can be used:

```
>>> a = range(10)
>>> next((x for x in a if x > 10), 'default')
'default'
```

Instead of writing this same function in every program you make, you can include the small but very useful Python package **first**:

Example 4.1. Using `first`

```
>>> from first import first
>>> first([0, False, None, [], (), 42])
42
>>> first([-1, 0, 1, 2])
-1
>>> first([-1, 0, 1, 2], key=lambda x: x > 0)
1
```

The `key` argument can be used to provide a function which receives each item as an argument and returns a Boolean indicating whether it satisfies the condition.

You will notice that we used `lambda` in a good number of the examples so far in this chapter. In the first place, `lambda` was added to Python to facilitate functional programming functions such as `map` and `filter`, which otherwise would have required writing an entirely new function every time you wanted to check a different condition:

```
import operator
from first import first

def greater_than_zero(number):
    return number > 0

first([-1, 0, 1, 2], key=greater_than_zero)
```

This code works identically to the previous example, but it is a good deal more cumbersome: if we wanted to get the first number in the sequence that is greater than, say, 42, then we would need to `def` an appropriate function rather than defining it in-line with our call to `first`.

However, despite its usefulness in helping us avoid situations like this, `lambda` still has its problems. First and most obviously, we cannot pass a `key` function using `lambda` if it would require more than a single line of

code. In this event, we are back to the cumbersome pattern of writing new function definitions for each `key` we need... or are we?

Our first step towards replacing `lambda` with a more flexible alternative is `functools.partial`. It allows us to create a wrapper function with a twist: rather than changing the behavior of a function, it instead changes the *arguments* it receives:

```python
from functools import partial
from first import first


def greater_than(number, min=0):
    return number > min

first([-1, 0, 1, 2], key=partial(greater_than, min=42
```

Our new `greater_than` function works just like the old `greater_than_zero` by default, but now we can specify the value we want to compare our numbers to. In this case, we pass `functools.partial` our function and the value we want for `min`, and we get back a new function that has `min` set to 42, just like we want. In other words, we can write a function and use `functools.partial` to customize what it does to our needs in any given situation.

This is still a couple of lines more than we strictly need in this case, though. All we are doing in this example is comparing two numbers; what if Python had built-in functions for these kinds of comparisons? As it turns out, the **operator** module has just what we are looking for:

```python
import operator
from functools import partial
from first import first

first([-1, 0, 1, 2], key=partial(operator.le, 0))
```

Here we see that `functools.partial` also works with positional arguments. In this case, `operator.le(a, b)` takes two numbers and returns whether the first is less than or equal to the second: the 0 we pass to `functools.partial` gets sent to `a`, and the argument passed to the function returned by `functools.partial` gets sent to `b`. So this works identically to our initial example, without using `lambda` or defining any additional functions.

Note

`functools.partial` is typically useful as a replacement of `lambda` and it should be considered as a superior alternative. `lambda` is to be considered an anomaly in Python language [5], due to its limited body size of a single expression that is one line long. On the other hand, `functools.partial` is built as a nice wrapper around the original function.

The **itertools** module in the Python standard library also provides a bunch of useful functions that you will want to keep in mind. I've seen too many programmers end up writing their own versions of these functions even though Python itself provides them out-of-the-box:

- `accumulate(iterable[, func])` returns a series of accumulation of items from iterables, or whatever is mapped to the `+` operator.

- `chain(*iterables)` iterates over multiple iterables, one after another without building an intermediate list of all items.

- `combinations(iterable, r)` generates all combination of length `r` from the given `iterable`.

- `compress(data, selectors)` applies a Boolean mask from `selectors` to `data` and returns only the values from `data` where the corresponding element of `selectors` is true.

- `count(start, step)` generates an endless sequence of values, starting from `start` and incrementing a `step` at a time with each call.

- `cycle(iterable)` loops repeatedly over the values in `iterable`.

- `repeat(elem[, n])` repeats an element `n` times.

- `dropwhile(predicate, iterable)` filters elements of an iterable starting from the beginning until `predicate` is false.

- `groupby(iterable, keyfunc)` creates an iterator grouping items by the result returned by the *keyfunc* function.

- `permutations(iterable[, r])` returns successive r-length permutations of the items in `iterable`.

- `product(*iterables)` returns an iterable of the Cartesian product of `iterables` without using a nested `for` loop.

- `takewhile(predicate, iterable)` returns elements of an iterable starting from the beginning until `predicate` is false.

These functions are particularly useful in conjunction with the *operator* module. When used together, *itertools* and *operator* can handle most situations that programmers typically rely on `lambda` for:

Example 4.2. Using the `operator` module with `itertools.groupby`

```
>>> import itertools
>>> a = [{'foo': 'bar'}, {'foo': 'bar', 'x': 42}, {'f
>>> import operator
>>> list(itertools.groupby(a, operator.itemgetter('fc
[('bar', <itertools._grouper object at 0xb000d0>), ('
>>> [(key, list(group)) for key, group in itertools.g
[('bar', [{'foo': 'bar'}, {'x': 42, 'foo': 'bar'}]),
```

In this case, we could have also written `lambda x: x['foo']`, but using `operator` lets us avoid having to use `lambda` at all.

As previously explained, all of the code and functions presented in this section are purely functional. That means they have no side effects and they moreover have no dependency on any global, shared data. Writing

code using that style of programming is a key to scalability as it makes it easy to execute those function in parallel, or even to spread their execution on different systems as will be discussed in [Chapter 5, *Queue-Based Distribution*](#).

[5] `lambda` was was once even planned for removal in Python 3, but in the end it escaped this fate.

Chapter 5. Queue-Based Distribution

As described, for example, in Section 9.4, "Asynchronous HTTP API", it is not always possible or the best approach to synchronously reply to a client request. Sometimes, the best response is to accept a job and promise to execute it later.

Implementing distributed system using queues is a trade-off, improving throughput with the drawback of increased latency.

Looking at basic architecture, it is composed of two elements: a queue that stores jobs and workers that consume the jobs from a queue, process them, and send the result to another destination.

Figure 5.1. Queue architecture
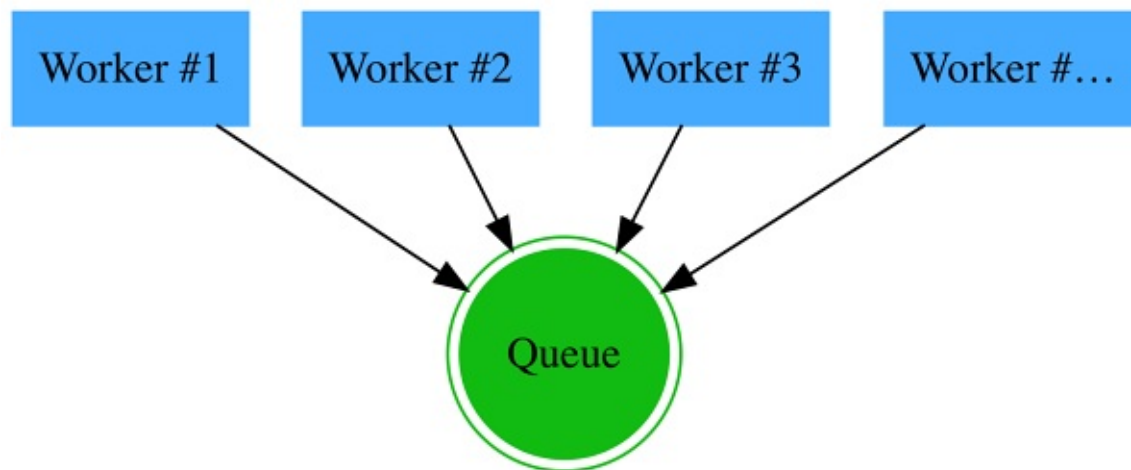
When an application makes such a promise, a job distribution mechanism is needed. The upside is that if correctly implemented, it allows the application to scale its workload horizontally pretty easily.

Indeed, as more entry points add more jobs to the queue, more workers (job processors) can be spawned as needed. In dynamic environments nowadays, it is not outside the realm of possibility that those workers

appear and disappear dynamically, controlled by some external systems monitoring the health of the processed queue.

As no nodes have infinite memory or storage, it is likely best that queues should be bounded and limited. The limit is often far from being known in advance, so experimentation and experience play a role here. Looking at it from a different angle, if your workload is bigger than your processing power, the queue will never ever be big enough.

There are multiple queue processing systems available in Python, from built-in in the standard library to external packages that rely on other pieces to build a complete stack. Some of them are described in this chapter.

The other upside of queue-based architecture is that it makes the testing and debugging of the application easier. A *worker* (a piece of code treating the jobs from the queue and pushing the results somewhere else) can be treated as a black box that takes an input and outputs a result. If it can be properly written in a purely-functional manner (i.e. without any side effects) then that makes it much easier to scale, replace, test, debug or even improve.

5.1. RQ

The *RQ library* provides a simple and direct approach to implementing queuing in any Python program. The letters *R* and *Q* are used to designate *Redis* and *queue*; this accurately summarizes what this library is trying to achieve.

If you don't know Redis, it is an open-source in-memory database project implementing a networked, in-memory key-value store with optional durability. It supports different kinds of abstract data structures, among

them strings, lists, maps, sets and sorted sets. That makes it a nice fit for implementing a simple queue mechanism.

Note

All the examples in this section assume that you have a Redis server running on `localhost`.

By using *RQ*, an application can push jobs into a Redis database and have workers executing these jobs asynchronously. To push a job into a queue, the `enqueue` method of that queue must be used, as demonstrated in [Example 5.1, "Pushing a rq job in a queue"](#).

Example 5.1. Pushing a rq job in a queue

```python
import time

from rq import Queue
from redis import Redis

q = Queue(connection=Redis())

job = q.enqueue(sum, [42, 43])
# Wait until the result is ready
while job.result is None:
    time.sleep(1)

print(job.result)
```

When starting the program in [Example 5.1, "Pushing a rq job in a queue"](#), a job whose actual work is to execute `sum([42, 43])` is pushed into a queue. However, for now, nothing is processing the queue, so the program waits forever, sleeping one second.

To start consuming those events, *RQ* provides an `rq` command line tool that is responsible for initiating a worker. Once run, it outputs the following:

```
$ rq worker
RQ worker 'rq:worker:abydos.87255' started, version (
Cleaning registries for queue: default

*** Listening on default...
default: builtins.sum([42, 43]) (631eeb5f-66f2-44f8-8
default: Job OK (631eeb5f-66f2-44f8-817b-2d0ef285f8e3
Result is kept for 500 seconds (3)

*** Listening on default...
```

(1) This is the function to execute and the UUID of the job

(2) The job has been correctly executed

(3) The result must be retrieved by that time

As `rq` indicates above, in this case the result of the job is put back into the queue for 500 seconds. Redis supports setting values with a specific lifespan, or time-to-live, meaning that after 500 seconds Redis will destroy the result.

Warning

To pass jobs around, rq serializes the jobs using `pickle`, a Python-specific serialization format. That implies several things:

- The Python version must be the same on both producers and workers.

- The code version must be the same on both producers and workers, and both must have the same version of the code so they can, import the same module, for example.

- It is close to impossible to have producers and workers using any language other than Python.

As with all queuing systems, the called function must not have any dependency on any global variable or context. As you can see, having

purely-functional functions implemented is a mandatory element of building distributed systems like this one.

RQ provides several nifty features that can help tweaking your workloads, such as specifying the time-to-live of the job itself (`ttl` option) or the time-to-live of the result (`ttl_result` option). The queue name can also be specified, making it easy to dispatch jobs to different queues.

Example 5.2. Pushing a rq job in a queue

```python
import time

from rq import Queue
from redis import Redis
import requests

q = Queue(name="http", connection=Redis())

job = q.enqueue(requests.get, "http://httpbin.org/del
                ttl=60, result_ttl=300)
# While the URL is fetched, it's possible to do anyth
# Wait until the result is ready.
while job.result is None:
    time.sleep(1)

print(job.result)
```

Running the worker with a custom queue name is possible by passing the name as an argument via the command line:

```
$ rq worker http
RQ worker 'rq:worker:abydos.95246' started, version (
Cleaning registries for queue: http

*** Listening on http...
http: requests.api.get('http://httpbin.org/delay/1')
http: Job OK (8cb8ad47-e873-4b78-8cfd-e94f39170343)
```

```
Result is kept for 300 seconds

*** Listening on http...
```

The *rq* command line tool reveals information about the state of the queue:

```
$ rq info
http          |███   3
failed        | 0
default       | 0
4 queues, 3 jobs total

0 workers, 4 queues

Updated: 2017-09-27 21:37:38.030541
```

The cherry on the cake, the *rq-dashboard* package ([available on PyPI](#)) allows for visual feedback and gives some control over the queues as you can see in [Figure 5.2, "rq-dashboard example"](#). Once installed, simply run it by running the *rq-dashboard* , using arguments to point to the right Redis instance as needed. It is then available on port 9181 by default.

Figure 5.2. rq-dashboard example

Queues

This list below contains all the registered queues with the number of jobs currently in the queue. Select a queue from above to view all jobs currently pending on the queue.

Queue	Jobs
default	0
failed	0
http	1
low	0

Workers

This list below contains all the registered workers.

State	Worker	Queues
No workers.		

Jobs on **http**

This list below contains all the registered jobs on queue **http**, sorted by age (oldest on top).

[Requeue All] [Compact] [Empty]

Name	Age	Actions
requests.api.get('http://httpbin.org/delay/1') 57f4befb-782c-4743-9706-16b58fda550c	13 seconds ago	✖ Cancel

« 1 »

As promised, *RQ* is very simple to install, deploy and administer. Obviously, it has its limitation, but it makes it implementing an asynchronous and distributed workflow in your application straightforward.

5.2. Celery
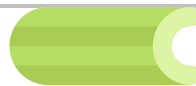
Celery is another queue management system. In contrast to *RQ*, it is broker agnostic and can use various software as broker, such as Redis, RabbitMQ or Amazon SQS. Moreover if you are brave enough, it is possible to write your own driver.

Celery also needs a **backend** for storing the results of the job. It supports a bunch of solution, such as Redis, MongoDB, SQL databases, ElasticSearch, files, etc. Just like for brokers, you can also write your own.

Note

Celery implements its own serialization format for its jobs. However, this format is not specific to Python. That means it is possible to implement job creators or consumers in different languages; there are already clients in PHP and Javascript.

In *Celery*, tasks are functions that can be called asynchronously. When called, *Celery* puts them in the broker queue for execution. Remote workers then execute the tasks, putting the task results into the backend.

When called, a task returns a `celery.result.AsyncResult` object. This object offers a principal method, `get`, which returns the result as soon as it is available, blocking the program in the meantime.

Example 5.3. A simple *Celery* task

```python
import celery


app = celery.Celery('celery-task',
                    broker='redis://localhost',
                    backend='redis://localhost')


@app.task
def add(x, y):
    return x + y


if __name__ == '__main__':
    result = add.delay(4, 4)
    print("Task state: %s" % result.state)
    print("Result: %s" % result.get())
    print("Task state: %s" % result.state)
```

Example 5.3, "A simple *Celery* task" is a simple implementation of a

Celery task. The *Celery* application is created with the main module name as its first argument, and then the URL to access the broker and backends.

The `app.task` function decorator registers the `add` task so it can be used asynchronously in the application, leveraging *Celery* workers for execution.

Once run, this program prints the following:

```
$ python celery-task.py
python examples/celery-task.py
Task state: PENDING
```

The program is blocked and then waits forever. There is no worker processing the job queue yet, therefore calling the `result.get` method blocks the program until the result is ready — which is not going to happen until a worker starts.

The `celery` command line tool provides a broad set of commands to manipulate and inspect the jobs queue and the workers. It is in charge of starting the actual workers:

```
$ celery worker --app celery-task

celery@abydos v4.1.0 (latentcall)

Darwin-17.0.0-x86_64-i386-64bit 2017-10-21 13:05:46

[config]
.> app:         celery-task:0x102d4e3d0
.> transport:   redis://localhost:6379//
.> results:     redis://localhost/
.> concurrency: 4 (prefork)
.> task events: OFF (enable -E to monitor tasks in th
```

```
[queues]
.> celery                 exchange=celery(direct) key=celei
```

As soon as it starts, the worker processes queued tasks and puts results in the backend. That unblocks the program `celery-task` started earlier:

```
$ python celery-task.py
Task state: PENDING
Result: 8
Task state: SUCCESS
```

This program can be started over and over again, and it won't hang for too long as long as a worker is running and processing the tasks.

5.2.1. Handling Failures

Task executions might fail, and in this case it is crucial to handle that properly. It is common for tasks to depend on external services, such as a remote database or a REST API. Connection failure might be transient; it is therefore better to deal with defeat and retry later.

Example 5.4. A *Celery* task with retry support

```
import celery


app = celery.Celery('celery-task-retry',
                    broker='redis://localhost',
                    backend='redis://localhost')


@app.task(bind=True, retry_backoff=True,
          retry_kwargs={'max_retries': 5})
def add(self, x, y):
    try:
        return x + y
```

```
    except OverflowError as exc:
        self.retry(exc=exc)


if __name__ == '__main__':
    result = add.delay(4, 4)
    print("Task state: %s" % result.state)
    print("Result: %s" % result.get())
    print("Task state: %s" % result.state)
```

Example 5.4, "A *Celery* task with retry support" implements a simple retry logic if an `OverflowError` occurs. The `retry_backoff` argument makes sure that *Celery* retries using an exponential back-off algorithm between delays (as described in Section 6.2, "Retrying with *Tenacity*"), while the `max_retries` argument makes sure it does not retry more than five times. Limiting the number of retries is important, as you never want to have jobs stuck forever in your queue because of a permanent error or a bug.

Retrying a task calls the same function with the same argument. That means that the best design for those tasks is, as is usually the case, to be entirely idempotent. If a task has side effects and fails in the middle of its execution, it might be more complicated to handle the task repeat execution later on. Imagine the following piece of code:

```
@app.task(autoretry_for=(DatabaseError,))
def record_visit(user_id):
    database.increment_visitor_counter()
    remote_api.record_last_visit_time(user_id)
```

If an error occurs while calling `remote_api.record_last_visit_time`, the visitor counter would already be incremented. When the task is retried, the counter will be incremented again, counting the visitor twice. Such a task should be re-written in a way where if executed multiple times, it produces the same result in the final system.

By default, *Celery* stores the results in the specified backend. However, it is sometimes the case that a task has no interesting return value. In that case, pass the `ignore_result=True` to the `app.task` decorator to make sure the result are ignored.

5.2.2. Chaining Tasks

Celery supports chaining tasks, which allows you to build more complex workflows.

Example 5.5. A *Celery* task chain

```python
import celery

app = celery.Celery('celery-chain',
                    broker='redis://localhost',
                    backend='redis://localhost')


@app.task
def add(x, y):
    return x + y


@app.task
def multiply(x, y):
    return x * y


if __name__ == '__main__':
    chain = celery.chain(add.s(4, 6), multiply.s(10))
    print("Chain: %s" % chain)
    result = chain()
    print("Task state: %s" % result.state)
    print("Result: %s" % result.get())
    print("Task state: %s" % result.state)
```

Example 5.4, "A *Celery* task with retry support" shows how to chain two tasks. First, the numbers 4 and 6 are summed using the `add` function. Then the result of this function is passed to `multiply` with 10 as the second argument. This outputs:

```
$ python celery-chain.py
Chain: celery-chain.add(4, 6) | multiply(10)
Task state: PENDING
Result: 100
Task state: SUCCESS
```

Building your program with multiple idempotent functions that can be chained together is something very natural in functional programming. Again, this kind of design makes it very easy to parallelize job execution and therefore make it possible to increase the throughput of your program and scale its execution horizontally.

5.2.3. Multiple Queues

By default, *Celery* uses a single queue named `celery`. However, it is possible to use multiple queues to spread the distribution of the tasks. This feature makes it possible to have a more finely grained control over the distribution of the jobs to execute.

For example, it is common to have a queue dedicated to low-priority jobs, where only a few workers are available.

The queue for a task can be specified at call time as shown in Example 5.6, "A *Celery* task called with a specific queue".

Example 5.6. A *Celery* task called with a specific queue

```
import celery


app = celery.Celery('celery-task-queue',
                    broker='redis://localhost',
```

```
                         backend='redis://localhost')


@app.task
def add(x, y):
    return x + y



if __name__ == '__main__':
    result = add.apply_async(args=[4, 6], queue='low-
    print("Task state: %s" % result.state)
    print("Result: %s" % result.get())
    print("Task state: %s" % result.state)
```

To treat queues other than just the default one, there's the `--queues` option:

```
$ celery worker --app celery-task-queue --queues cele
celery@abydos v4.1.0 (latentcall)

Darwin-17.0.0-x86_64-i386-64bit 2017-10-22 12:39:26

[config]
.> app:         celery-task-queue:0x10ca73690
.> transport:   redis://localhost:6379//
.> results:     redis://localhost/
.> concurrency: 4 (prefork)
.> task events: OFF (enable -E to monitor tasks in th

[queues]
.> celery           exchange=celery(direct) key=cele
.> low-priority     exchange=low-priority(direct) key
```

This worker consumes jobs from both the default `celery` queue and the `low-priority` queues. Running other workers with just the `celery` queue would make sure that the low-priority queue is only acted on by

one worker when the worker has time to do it, whereas all the other workers would keep waiting for normal priority jobs on the default `celery` queue.

There is no magic recipe to determine the number of queues that your application may need. Using the jobs priority as a criteria to split them is the most common and obvious use case. Queues give you access to more finely grained scheduling possibilities, so don't hesitate to use them.

5.2.4. Monitoring

Celery comes with a lot of monitoring tools built in, and this allows you to supervise things and get information about what is going on inside the cluster. That also makes it a great option over a custom built solution where all of this would have to be implemented again.

The basic monitoring command is `status`, which returns the state of the workers:

```
$ celery status --app celery-task-queue
celery@abydos: OK

1 node online.
```

The `inspect` command accepts a few subcommands, among them `active`, which returns the task currently being done:
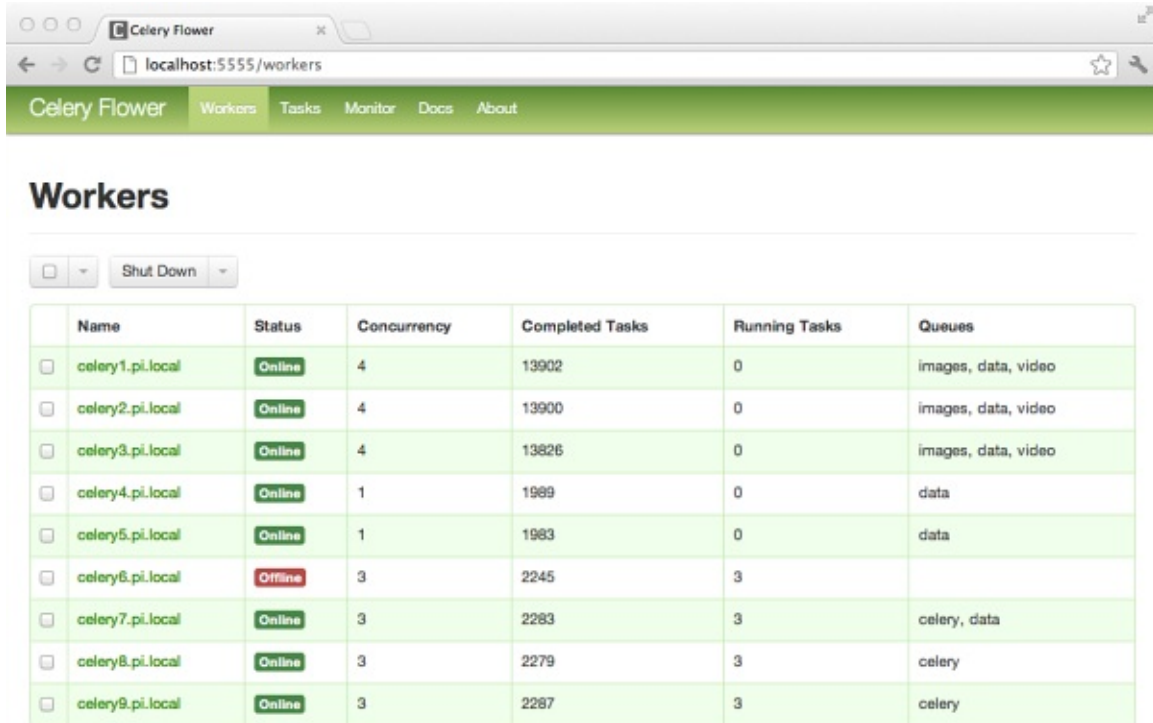
```
$ celery inspect active --app celery-task-queue
-> celery@abydos: OK
    - empty -
```

Celery also comes with a nicely designed Web dashboard that allows for supervising the activity of the workers and the queue. It is named *Flower* and it is easy to install using `pip install flower`. It is simple enough to start:

```
$ celery flower --app celery-task-queue
[I 171022 13:06:46 command:139] Visit me at http://l
[I 171022 13:06:46 command:144] Broker: redis://local
[I 171022 13:06:46 command:147] Registered tasks:
    [u'celery-task-queue.add',
     u'celery.accumulate',
     u'celery.backend_cleanup',
     u'celery.chain',
     u'celery.chord',
     u'celery.chord_unlock',
     u'celery.chunks',
     u'celery.group',
     u'celery.map',
     u'celery.starmap']
```

Once started, it is available on the shown URL:
`http://localhost:5555`. It offers a nice dashboard as shown in
Figure 5.3, "Celery Flower dashboard".

Figure 5.3. Celery Flower dashboard

Celery is widely used, and many resources are available that cover how to monitor it in a production system. That makes it safe bet in this regard!

5.3. Joshua Harlow on Task Distribution

Hi Josh! Could you introduce yourself and explain how you came to Python?

Well hi there! I grew up in upstate New York. I went to school at RIT (and prior to that Clarkson University as well as a NY state college)

and graduated in 2007 with a Masters in Computer Science. During this time I interned at IBM (did some automation work using Jython) and Intel (where I helped the graphics team by interconnecting Ruby and C#). While I was in college, I got very interested in distributed systems, and the interconnect/potential when combined with AI (as well as a stint in language theory and applications).

After graduation, I came to work at Yahoo. After working on various projects such as the homepage (www.yahoo.com), I got recruited into a sub-team under the CTO organization where we were tasked with determining the cloud solution Yahoo should invest in and use. OpenStack was a nascent open source cloud technology back then, but it was what we thought had the most potential. Since OpenStack was being written entirely in Python, this is where I got my actual initiation into Python. Over time I have come to enjoy Python, come to learn it deeply, been featured in a book on it and never looked back!

What's your experience with building large scale systems?

I have been working in the industry for around ten years. Two-thirds of that was working at Yahoo where I was introduced to various patterns and systems around how to design a scalable system. At Yahoo, anything that runs must be able to scale. Part of my time there was being involved in the OpenStack community (while also being one of the key Yahoo OpenStack contributors and one of the technical leads/architects on the larger team there) where I have tried to work, inject, or at least share some of those same ideas/lessons learned into various OpenStack projects.

Before this, I did graduate work using a highly distributed paradigm called [agent-oriented programming](#) which I applied to train (and automatically retrain) classifier agents in an unsupervised manner ([thesis](#)). This paradigm is similar to the one that the Erlang language has (nearly built-in), and in general, it favors the creation of small agents that do specialized units of work, and to accomplish a substantial piece of work those agents intercommunicate over prescribed channels. It was my first significant involvement in a highly distributed pattern and was a unique way to combine distributed systems concepts with machine learning/training. It likely is still not fully explored.

After Yahoo, I moved over to GoDaddy, as a technical lead and architect and one of their primary OSS contributors/evangelists/strategists. I am continuing with OpenStack and evolving the IAAS (infrastructure as a service) and CAAS (containers as a service) landscape that GoDaddy is actively involved in and contributing to (both of which involve scale).

I am part owner, part maintainer, and part creator of multiple Python libraries such as:

- Kazoo - the most widely used [ZooKeeper](#) client/Python binding library (written in pure python) which supports various async methodologies (selectable by the user using this library).
- TaskFlow - a Python library built to help design workflows in a programmatic manner (by writing them in Python) that once encoded and structured can be ran across workers (similar to *[Celery](#)*) in the order the workflow defines (or even in parallel). It also comes built-in with concepts that make it easier to build in HA execution of those workflows (in part by using kazoo).
- Tooz - A self-proclaimed distributed primitives build block library

that provides high-level blocks and various connections to different supporting backends (each with its limitations and drawbacks) for each of those primitives.

- Oslo.messaging - Widely used (inside of OpenStack) messaging and RPC framework that is used for internal service to service communication inside most OpenStack services.

- Fasteners - Interprocess (local to the same machine) locking library that also provides various thread specific functionality that was found to be missing as built-in to Python (used inside and outside of OpenStack).

Most of these libraries are focused on providing primitives and building blocks that help build higher level, user-centric features. The goal of these libraries is to build patterns out of the box to aid in building your application in a scalable, reliable, safe and straightforward manner, from the beginning, instead of trying to retrofit.

I also contribute to various other cloud computing core projects such as those in OpenStack but also ones that reach beyond OpenStack such as [cloud-init](#) (a boot-time program that is used in nearly all clouds virtual machine images). I also like dabbling in non-Python/new/upcoming languages, libraries and system desig in my spare time (among other things that I also partake in).

You mention Erlang as one of the languages that have built-in helpers for distributed systems. So what's the strong point of Python regarding distributed system in your opinion? On the other hand, where does it fall short?

Interesting question.

I would have to say the strong point of Python is its simplicity and

the built-ins that come with it; it is very manageable to hit the ground running with Python. The number of lines of code it requires to set up a small task queue using Redis with producers putting work into Redis and workers completing that work is surprisingly small.

However, where it falls short I feel is that this kind of operations, e.g., connect into Redis and set up a producer/worker/consumer system, is not natively built-in. This means that it gets reinvented and that application builders **eventually** hit some limit where they then have to do rewrites to support broader scale. Other languages, such as Erlang (and to a degree, Go) build some of these (but not all) paradigms right into the language; this helps alleviate the need to rewrite at a later date.

Though I would say, I have yet to see a language that natively builds in (as language constructs) all the various primitives that I would like. Most just pick a few, and you have to adapt your paradigms either to those or around those. Perhaps someday in the future, a language with all these built-in that is also **simple** to use will exist (but it does not seem to have appeared yet, at least that I am aware of). Maybe someone reading this will build it!

What would be your advice to fulfill these shortcomings when using Python, and avoid a rewrite later on?

So my advice around this topic is slightly multi-faceted and is broken up into a few thought experiments that I try to run ahead of time (before getting too down deep into the code and getting locked into a certain design).

One key thing and just a good practice, in general, is to define your

interfaces (the API that external users use) carefully. Some of this you learn over time, but it applies that you should try to make it as simple as possible and leave out internal details. For example, as I have seen it in some OpenStack projects, everything that can be made an option is made an option; this is not a good API. Even though I do get the reasoning behind why this happens – there is always an operator somewhere that wants this to be configurable. The less you expose makes it easier to alter the backend later.

Another example, take the *memcached* API; it is pretty straightforward and simple. It is primarily composed of `add`, `get`, `set`, `cas`, `delete`. Offering this API and defining it well has allowed for a variety of different implementations of that API; some that scale differently, some that back the key store with a consistent database and so on. So put some thought into the API ahead of time, and it usually pays off in the long term in regards to what you can change to scale and keep it as simple as you can.

This also applies when creating Python libraries and is a good principle that goes back to the UNIX days of **do one thing and do it well**.

A second thing that I have seen, and seems rather standard, is to isolate your network I/O code behind a layer that lets that layer switch to a different implementation. For example, the Kazoo library is an example of this where the I/O layer can switch to *eventlet* or threads – or perhaps if someone submits the code for it, to asyncio. It is typically an excellent pattern to implement because if things are too tightly coupled then it becomes much harder to alter the layer in any manner at a later time. Of course, you need to benchmark your

library/application before going about such a change in the first place.

A third thing, and one that is relevant to not just Python code but as a general concept, is to document thoroughly what resources are used and how they are used in the clearest way possible. Take as an example a piece of code running on one machine that is altering some resource X and at the same time another machine is also trying to alter resource X. This is a classic example of a problem waiting to happen – i.e., what happens when they both modify X. It is one of those patterns that need to be documented in the design review phase of your software. One thing that can help here is creating (even a simple) domain-specific language (DSL) that can try to analyze your workflows at compilation time to find potential locks (before your program even starts running).

What advice would you have to write a resilient Python application, one ready to handle failures? How does Python help in this regard, and where does it need attention from the developer?

So a few things that I would suggest; not a complete comprehensive listing but some general points of interest that I think I have accumulated over the years; especially around building a resilient application ready to handle failure.

- Build in service discovery (especially if your app runs across multiple hosts) from the start (and please do not create it yourself, use existing technologies that already do it better than you will).
- Know where locks are needed and where they are not (and try to avoid needing more than a handful of them).

- Strive for crash tolerance (and plan for it). It is useful to think about how a database works and understand what a transaction log or journal is (and maybe such functionality can be useful for your application/library).

- Try to be stateless (avoids many of the previously mentioned issues if you have no state in the first place).

- Don't aim for the lowest bar (aim high from the start and compromise as needed, vs. doing the reverse); it will likely end better that way.

Though those kinds of things are healthy across just more than Python!

Chapter 6. Designing for Failure

Exception handling is one of the most brushed aside aspects of computer programming. Errors are complicated to handle, and often they are unlikely, so developers always forget to handle failures… sometimes they even forget on purpose.

However, in a world where applications are distributed over the network, across miles of fiber optic cable and on different computers, failure is not an exception. It must be considered as the norm for your software. Failure scenarios must be first-class citizens of the various testing scenarios being developed.

In an environment distributed over a network, anything that can fail **will** fail.

Python does not offer any help in that regard, and almost no programming language offers advanced error recovery or retrying capability – except maybe languages implementing [condition systems](#), such as Common Lisp. The following sections give details on some strategies that can help handle failures and retry logic.

6.1. Naive Retrying

There's a common pattern that can be seen across all sort of programs that can be summarized with the word "retrying". It is directly tied to the idea of trying something again that returned an error or raised an exception when it was not expected.

The basic pattern that most developers implement looks like what is shown in [Example 6.1, "Retrying pattern"](#).

Example 6.1. Retrying pattern

```
while True:
    try:
```

```
        do_something()
    except:
        pass
    else:
        break
```

The example in <u>Example 6.1, "Retrying pattern"</u> does not provide any down time for the called function. Usually, smarter strategies are implemented. That is especially important if the code is going to do something like connecting to an external system. The last thing you want is to repeatedly hammer constantly the remote system. <u>Example 6.2, "Retrying pattern with sleep"</u> illustrates the most common implementation of this retry and sleep pattern.

Example 6.2. Retrying pattern with sleep

```
import time
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


while True:
    try:
        do_something()
    except:
        # Sleep for one second before retrying
        time.sleep(1)
    else:
        break
```

This might be one of the most naive implementations that you can use.

The problem with that simple approach is that in complex systems it can increase congestion. If the job is to communicate over the network, the targeted system might be hammered every second by a bunch of programs trying to communicate with it, which is not going to help it in terms of coping with its potential load.

This problem is well-known in networks, and various techniques have been employed to avoid congestion collapse. Among them, the exponential backoff algorithm allows for spacing out retries. Example 6.3, "Retrying pattern with exponential backoff" provides a straightforward Python implementation of this algorithm.

Example 6.3. Retrying pattern with exponential backoff

```python
import time
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


attempt = 0
while True:
    try:
        do_something()
    except:
        # Sleep for 2^attempt seconds before retrying
        time.sleep(2 ** attempt)
        attempt += 1
    else:
        break
```

On the first attempt, the waiting time is 1 second. On the second, it is

2 seconds, then 4 seconds, then 8 seconds, and so on. Increasing the waiting delay makes sure that in the case of congestion, the system is not hit with too many requests at the same time in order to make sure that it might be able to recover.

We could go on and on about the different patterns and how to implement them in Python. Rather than doing that, it is preferable to use the *tenacity* library described in Section 6.2, "Retrying with *Tenacity*"

6.2. Retrying with *Tenacity*

Distributed applications often use the retry pattern described in Section 6.1, "Naive Retrying". As soon as an application scales over several nodes across a network, it needs to handle failure scenarios that might occur. For example, as soon as an application sends some request over HTTP, there is a possibility that the demand fails: the server might be down, the network might be unreachable, or the remote application might have a temporary overload. The requester needs to retry and handle all those different conditions correctly in order to be robust.

Since this is a very typical usage, a library called *tenacity* has been created for Python, and it can easily be used to implement this strategy on any function using a decorator.

Example 6.4, "Basic retrying with *tenacity*" implements the algorithm shown in Example 6.1, "Retrying pattern" in a few lines using *tenacity*. This will make the function `do_something` be called over and over again until it succeeds by not raising an exception of any kind.

Example 6.4. Basic retrying with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
```

```
        print("Failure")
        raise RuntimeError
    print("Success")


tenacity.Retrying()(do_something)
```

Obviously, this is a pretty rare case. Retrying without any delay is not what most applications want, as it can heavily burden the failing subsystem. An application usually needs to wait between retries. For that, *tenacity* offers a large panel of waiting methods. The equivalent implementation of [Example 6.2, "Retrying pattern with sleep"](#) can be seen in [Example 6.5, "Fixed waiting with *tenacity*"](#).

Example 6.5. Fixed waiting with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


@tenacity.retry(wait=tenacity.wait_fixed(1))
def do_something_and_retry():
    do_something()


do_something_and_retry()
```

tenacity can also be used as a decorator – this is actually the way it is used most of the time.

The problem when using a fixed time to wait is that in cases of temporary failure, the configured delay might be too long, and in cases of a more significant failure, the delay might be too short. Since the delay is fixed, there might be no good choice between the two options.

An excellent alternative is to use the exponential back-off method, which can be used instead as shown in Example 6.6, "Exponential back-off waiting with *tenacity*".

Example 6.6. Exponential back-off waiting with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


@tenacity.retry(
    wait=tenacity.wait_exponential(multiplier=0.5, ma
)
def do_something_and_retry():
    do_something()


do_something_and_retry()
```

The way the decorator is configured in Example 6.6, "Exponential back-off waiting with *tenacity*" makes the algorithm retry after waiting first 1 second, then waiting 2 seconds, then 4 seconds, 8 seconds, 16 seconds and then 30 seconds, keeping that last value for all the subsequent retries. The back-off algorithm computes the time to wait by computing `min(multiplier * (exp_base^retry_attempt_number), max)`

and using that number of seconds. It makes sure that if a request is unable to succeed quickly after a retry, the application waits longer and longer each time, rather than repeatedly hammering the targeted subsystem at a fixed interval.

tenacity provides other wait algorithms such as the `wait_random_exponential` which implements one of the back-off algorithms described in an [Amazon Web Services Architecture blog post](#). This algorithm adds some variability to the random exponential wait so retry attempts are spread evenly rather than packed when clients retry accessing a service.

Another interesting point of *tenacity* is that you can easily combine several methods. For example, you can combine `tenacity.wait.wait_random` with `tenacity.wait.wait_fixed` to wait a number of seconds defined in an interval, as done in [Example 6.7, "Combining wait time with *tenacity*"](#).

Example 6.7. Combining wait time with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


@tenacity.retry(
    wait=tenacity.wait_fixed(10) + tenacity.wait_ranc
)
def do_something_and_retry():
    do_something()
```

```
  do_something_and_retry()
```

This makes the function being retried wait randomly for between 10 and 13 seconds before trying again.

tenacity offers more customization, such as retrying on some exceptions only.

Example 6.8. Specific retry condition with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")


@tenacity.retry(wait=tenacity.wait_fixed(1),
                retry=tenacity.retry_if_exception_typ
def do_something_and_retry():
    return do_something()


do_something_and_retry()
```

In [Example 6.8, "Specific retry condition with *tenacity*"](), *tenacity* is leveraged to retry every second to execute the function only if the exception raised by `do_something` is an instance of `RuntimeError`.

You can combine several conditions easily by using the `|` or `&` binary operators. In [Example 6.9, "Combining retry condition with *tenacity*"](), the conditions set up make the code retry if an `RuntimeError` exception

is raised, or if no result is returned. Also, a stop condition is added using the `stop` keyword arguments. It allows specifying a stop condition based on a maximum delay.

Example 6.9. Combining retry condition with *tenacity*

```
import tenacity
import random


def do_something():
    if random.randint(0, 1) == 0:
        print("Failure")
        raise RuntimeError
    print("Success")
    return True


@tenacity.retry(wait=tenacity.wait_fixed(1),
                stop=tenacity.stop_after_delay(60),
                retry=(tenacity.retry_if_exception_ty
                       tenacity.retry_if_result(
                           lambda result: result is M
def do_something_and_retry():
    return do_something()


do_something_and_retry()
```

The functional approach of *tenacity* makes it easy and clean to combine different conditions for various use cases with simplistic binary operators.

tenacity can also be used without a decorator by using the object `Retrying`, that implements its main behavior and use its `call` method. This object allows one to call any function with different retry conditions, as in Example 6.10, "Using *tenacity* without decorator", or to retry any

piece of code that does not use the decorator at all – like code from an external library.

Example 6.10. Using *tenacity* without decorator

```
import tenacity

r = tenacity.Retrying(
    wait=tenacity.wait_fixed(1),
    retry=tenacity.retry_if_exception_type(IOError))
r.call(do_something)
```

There should be no need for an application to implement its own retry logic when using *tenacity*. At the worst, it just needs to enhance it.

Chapter 7. Lock Management

When a program tries to access a resource that should not be accessed concurrently, a lock is the easiest mechanism for preventing shared access. Most operating systems provide local primitives for using locks, allowing different programs to request access to a resource without risking stepping on each other's toes. Unfortunately, once the application is distributed across several nodes (regardless what operating system), things become complicated.

This scenario needs a *distributed lock manager.* It consists of a central service, possibly spread across several network nodes, which makes it possible to acquire and release locks over the network. That means that the lock might be managed by more than one node, and so it must be synchronized between the different nodes.

Using a distributed lock manager makes it a certainty that the application is going to be able to perform some tasks with exclusive access, avoiding negative outcomes like data corruption.

Python does not offer anything different when compared to many languages in this regard. There is nothing built into the language, whereas it would be the case with, Erlang, for example, which has cluster-wide locks. Therefore, external services must be relied upon, such as:

- ZooKeeper
- Redis
- etcd
- Consul

Those are the most commonly used systems. There are also other services that can act as lock-managers, though they offer less redundancy, such as memcached or even PostgreSQL.

The following chapter digs into the lock primitives you might need in order to make your application concurrency-safe, and how to leverage

external services.

7.1. Thread Locks

In order to protect concurrent accesses to resources by multiple threads, Python provides `threading.Lock`.

For example, if we consider `stdout` as a resource where only one thread can have access at the same time, we need to use a `threading.Lock` object to synchronize the access to that resource. [Example 7.1, "Threads using a lock"](#) illustrates how to use a lock to share access to `stdout`.

Example 7.1. Threads using a lock

```python
import threading

stdout_lock = threading.Lock()


def print_something(something):
    with stdout_lock:
        print(something)


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

Now there are only two outputs possible:

```
$ python examples/chapter7-lock-management/threading-
hello
thread started
$ python examples/chapter7-lock-management/threading-
thread started
```

```
hello
```

The lock does not force the order of the execution, but it makes sure only one of the threads can use `print` at the same time, avoiding any data corruption. Imagine that instead of `stdout` the thread would write to the same file – that would be some significant data corruption! That is why locks are always required when accessing shared resources.

Some data types in Python have atomic operations, for example:

- `list.append`
- `list.extend`
- `list.__getitem__`
- `list.pop`
- `list.__setitem__`
- `list.sort`
- `x = y`
- `setattr`
- `dict.__setitem__`
- `dict.update`
- `dict.keys`

What this means is that when a program calls `mylist.append(1)` and `mylist.append(2)` in different threads, the list `mylist` contains 1 and 2 and no data corruption ever happens – see [Example 1.1, "Thread-unsafe code without the GIL"](#).

That is good to know since using locks has a huge cost. When a lock is acquired, all the other threads which need that lock have to wait, slowing the program down. However, since some of the basic Python data types expose atomic operations, an application can, for example, use a list across different threads and be safe appending data to it without the use

of a `threading.Lock` object.

In the case that one of your threads might need to acquire a lock multiple times (e.g. a recursive function), the `threading.RLock` provides what is called a reentrant lock. This type of lock can be acquired multiple times by the same thread, rather than being blocked when already acquired.

Example 7.2. Threads using a reentrant lock

```python
import threading

rlock = threading.RLock()

with rlock:
    with rlock:
        print("Double acquired")
```

If a `threading.Lock` was being used in Example 7.2, "Threads using a reentrant lock" instead, the program would be in a *dead-lock* state, unable to continue its execution.

The last commonly used object for synchronizing thread is `threading.Event`. You can think about this object as it is a boolean value, being `True` or `False`. A thread can set the value to true by calling `threading.Event.set()` and another thread can wait for the value to become `True` by calling `threading.Event.wait()`.

One of the most common use cases for such an object is to synchronize background thread with your main thread on exit.

Example 7.3. Threads using an `threading.Event` object

```python
import threading
import time


stop = threading.Event()
```

```
def background_job():
    while not stop.is_set():
        print("I'm still running!")
        stop.wait(0.1)


t = threading.Thread(target=background_job)
t.start()
print("thread started")
time.sleep(1)
stop.set()
t.join()
```

As boring as [Example 7.3, "Threads using an `threading.Event` object"](#) might be, it shows pretty well how the main thread can let the secondary thread know that it is time to stop. The usage of a `threading.Event` object for such synchronization between threads is a popular design pattern when using threads.

7.2. Processes Locks

When using multiple processes to distribute workload, an application needs a dedicated lock type that can be accessed by multiple local processes.

They are two different use cases for inter-processes locks:

* All your processes come from the `multiprocessing` Python package using the methods described in [Section 2.2, "Using Processes"](#).
* You have processes running independently, launched by `os.fork` or by an external process manager – processes that might not even be in Python.

We will see how to handle both situations in the following sections.

7.2.1. Multiprocessing Locks

When all programs accessing a shared resources are written in Python, the natural method to secure accesses is to use the `multiprocessing.Lock` objects provided by Python. That class is built around POSIX or Windows semaphores (depending on your operating system) and allows use of a lock across several processes.

Example 7.4. Printing cats in parallel

```python
import multiprocessing
import time


def print_cat():
    # Add some randomness by waiting a bit
    time.sleep(0.1)
    print(" /\\_/\\")
    print("( o.o )")
    print(" > ^ <")


with multiprocessing.Pool(processes=3) as pool:
    jobs = []
    for _ in range(5):
        jobs.append(pool.apply_async(print_cat))
    for job in jobs:
        job.wait()
```

Example 7.4, "Printing cats in parallel" is a very simple example that uses `multiprocessing.Pool` to prints cats in ASCII art. The program launches up to 3 processes that have to print 5 cats in parallel. When executed, the program outputs the following:

```
$ python examples/multiprocessing-lock.py
 /\_/\
```

```
 /\_/\
( o.o )
( o.o )
 > ^ <
 > ^ <
 /\_/\
( o.o )
 > ^ <
 /\_/\
 /\_/\
( o.o )
 > ^ <
( o.o )
 > ^ <
```

The cats are a bit mixed up. The standard output is shared across all the processes printing cats, and they are stepping on each other's toes. The way to fix that is to lock the standard output until the cat is fully printed on the screen. This is easily done by using a `multiprocessing.Lock`.

Example 7.5. Printing cats in parallel with a lock

```python
import multiprocessing
import time


stdout_lock = multiprocessing.Lock()


def print_cat():
    # Add some randomness by waiting a bit
    time.sleep(0.1)
    with stdout_lock:
        print(" /\\_/\\")
        print("( o.o )")
        print(" > ^ <")
```

```
with multiprocessing.Pool(processes=3) as pool:
    jobs = []
    for _ in range(5):
        jobs.append(pool.apply_async(print_cat))
    for job in jobs:
        job.wait()
```

When running [Example 7.5, "Printing cats in parallel with a lock"](#), the output now looks much better:

```
$ python examples/multiprocessing-lock.py
 /\_/\
( o.o )
 > ^ <
 /\_/\
( o.o )
 > ^ <
 /\_/\
( o.o )
 > ^ <
 /\_/\
( o.o )
 > ^ <
 /\_/\
( o.o )
 > ^ <
```

The cats are never mixed up and they are correctly drawn on the screen. The lock has to be acquired by any process trying to draw a cat on the terminal, making sure it completes its work before releasing it.

7.2.2. Inter-Processes Locks

As stated earlier, `multiprocessing.Lock` only works for processes started from a single Python process. If your application is distributed

across several Python processes – such as a daemon started independently – you need an *interprocess lock* mechanism.

Those locks are usually not portable across operating systems. POSIX, System V or even Windows all offer different interprocess communication mechanisms, and they are not compatible with one another. You may want to look in this direction if you are not afraid of making your software platform-dependent.

The *fasteners* module provides an excellent implementation of a generic solution based on file locks in Python.

Note

Locks implemented by *fasteners* are based on file-system locks. They are not specific to Python. Therefore you could also implement the same file-locking mechanism in another programming language in order to have cross-language locking.

Fasteners provides a lock class `fasteners.InterProcessLock` that takes a file path as its first argument. This file path is going to be the identifier for this lock, and it can be used across multiple independent processes. This is why *fasteners* is helpful to lock access to resources across processes that were started independently.

Example 7.6. Using *Fasteners* for interprocess locking

```
import time

import fasteners


lock = fasteners.InterProcessLock("/tmp/mylock")
with lock:
    print("Access locked")
    time.sleep(1)
```

If you run multiple copies of the program in

Fasteners for interprocess locking", they all sequentially print `Access locked` after acquiring the lock.

There is no helper provided by *fasteners* nor is there any rule to determine which file path should be used by your application. The usual convention is for it to be in a temporary directory that is purged on system start, such as `$TMPDIR` or `/var/run`. It is up to you to determine the directory where to create the file and to have a filename that is unique to your application, but known to all processes.

Fasteners also provides a function decorator to easily lock an entire function as shown in Example 7.7, "Using *Fasteners* decorator for interprocess locking".

Example 7.7. Using *Fasteners* decorator for interprocess locking

```
import time

import fasteners


@fasteners.interprocess_locked('/tmp/tmp_lock_file')
def locked_print():
    for i in range(10):
        print('I have the lock')
        time.sleep(0.1)


locked_print()
```

Fasteners locks are reliable and efficient. They do not have a single-point-of-failure except the operating system itself, so they are a good choice for inter-process locking needs that are local to a machine.

7.3. Using *etcd* for Distributed

Locking

etcd is a popular distributed key/value store. It stores keys and values and replicates them on several nodes which can be used to query or update the keys. To keep its member in sync, *etcd* implements the [Raft algorithm](). Raft solves the consensus issue in distributed systems by electing a leader who is responsible for maintaining consistency across nodes. In short, Raft makes sure that data stored by *etcd* are consistent across all nodes, and that if a node crashes, the *etcd* cluster can continue to operate until its failed node comes back to life.

In that regard, *etcd* allows implementing a distributed lock. The basic idea behind the lock algorithm is to write some value in a pre-determined key. All the services in the cluster would pick, for example, the key `lock1` and would try to write the value `acquired` in it. As *etcd* supports transactions, this operation would be executed in a transaction that would fail if the key already existed. In that case, the lock would be unable to be acquired. If it succeeds, the lock is acquired by the process that managed to create the key and write the value.

To release a lock, the client that acquired it just has to delete the key from *etcd*.

etcd is able to notify clients when a key is modified or deleted. Therefore, any client that was unable to create the key (and acquire the lock) can be notified as soon as the lock is released. Then it can try to acquire it.

If the client that acquired the lock crashes, the lock becomes impossible to release. To avoid such cases, the keys have a time-to-live predefined at their creation and need to be refreshed as long as the lock is kept acquired by the client.

This workflow is a basic locking algorithm and it is implementable with many other key/value stores.

Figure 7.1. Simple *etcd* lock algorithm
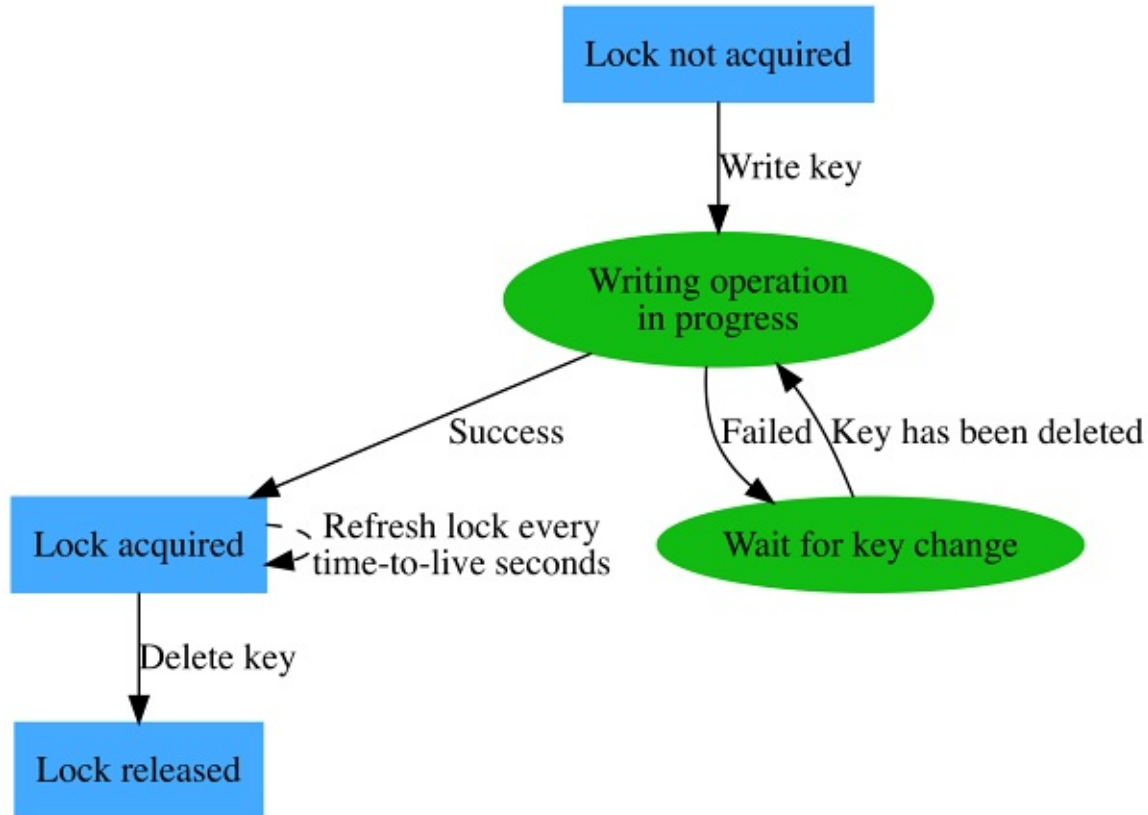
Rather than implementing this algorithm from scratch (I'll leave that to you as an exercise), leveraging the `etcd3.locks.Lock` class provided by the Python `etcd3` package is easier and safer, and therefore preferred.

[Example 7.8, "Locking with etcd"](#) shows how to get a lock from your local etcd server. As long as the lock is acquired, no other process can grab it. Since *etcd* is a network service, you can easily synchronize your process across a network using this simple lock mechanism.

Note

You need to start etcd to make examples work. Simply run `etcd` from the command line once it is installed.

Example 7.8. Locking with etcd

```
import etcd3
```

```
client = etcd3.client()
lock = client.lock("foobar")
lock.acquire()
try:
    print("do something")
finally:
    lock.release()
```

You can also use it with the `with` statement, which makes it more readable and handles exception in an easier way:

Example 7.9. Locking with etcd using the `with` statement

```
import etcd3

client = etcd3.client()
lock = client.lock("foobar")
with lock:
    print("do something")
```

For more robustness, deploying *etcd* as a cluster of several nodes makes sure that if the *etcd* server that your application connects to goes down, the rest of the cluster can continue to work, and likewise your clients, as long as they switch to a different server when an error occurs (though this feature is not implemented in *python-etcd3* yet).

Example 7.10, "Locking service with *etcd* and *Cotyledon*" implements a distributed service using the *Cotyledon* library (as discussed in Section 2.5, "Daemon Processes"). It spawns four different processes, and only one is authorized to print at any given time.

Example 7.10. Locking service with *etcd* and *Cotyledon*

```
import threading
import time


import cotyledon
```

```
import etcd3


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

    def __init__(self, worker_id):
        super(PrinterService, self).__init__(worker_
        self._shutdown = threading.Event()
        self.client = etcd3.client()

    def run(self):
        while not self._shutdown.is_set():
            with self.client.lock("print"):
                print("I'm %s and I'm the only one pr
                      % self.worker_id)
                time.sleep(1)

    def terminate(self):
        self._shutdown.set()


# Create a manager
manager = cotyledon.ServiceManager()
# Add 4 PrinterService to run
manager.add(PrinterService, 4)
# Run all of that
manager.run()
```

You can run this program any number of times on any number of machines on your network and you can be sure that one and only one process at a time will own this lock and be able to print its line. Since the lock is acquired for a tiny amount of time (a print operation and one second), we do not expect it to timeout. The default time-to-live for a lock is 60 seconds, which ought to be enough – if the program takes longer to print something and sleep one second, then something is wrong, and it might be better to let the lock expire.

However, for sustained operations, an application should not cheat and extend the time-to-live value. The program should keep the lock active by regularly calling the `Lock.refresh` method.

Combining such a distributed lock mechanism and a library like *Cotyledon* can make building a distributed service straightforward.

7.4. Using *Tooz* Locking Abstraction

Picking a distributed lock mechanism once and for all for your application is not necessarily obvious.

First, some solutions are heavier than other to deploy and maintain. For example, installing a memcached server is pretty straightforward, but maintaining a *ZooKeeper* cluster is much more complicated. Clearly, the two solutions are not strictly equivalent in terms of safety and guaranteed operation. However, as a developer, it might be handy to test using a small backend and operate at scale with a scalable backend.

Secondly, it is not always obvious which solution to pick. A few years ago, *ZooKeeper* was the hot thing and the only widely available implementation of the Paxos algorithm. However, nowadays, solutions such as *etcd* and its Raft implementation are getting more traction: the algorithm is simpler to understand and the project is less complicated to deploy and operate.

All those backends offer different levels of abstraction on top of distributed features. Some projects provide a full locking implementation whereas some others are only key/value stores.

The [*Tooz* library](#) was created a few years ago to solve those problems. It provides an abstraction on top of a varied set of backends, making it easy to switch from one service to another. This can be quite powerful, as it allows you to use memcached to test your distributed code on your laptop, for example, as it is lightweight to run and install, while you can

also support, something like a *ZooKeeper* cluster as a more robust solution. Moreover, whatever is the backend you pick, the distributed feature that you need, such as locking, are provided using the same API for your application, whatever the primitives provided by the underlying service.

To achieve that, *Tooz* provides a `Coordinator` object that represents the coordination service your application is connected to. The method `tooz.coordinator.get_coordinator` allows the program to get a new coordinator. All it needs is the URL to connect to and a unique identifier for the node, as you can see in [Example 7.11, "Getting a *Tooz* coordinator"](#).

Example 7.11. Getting a *Tooz* coordinator

```python
import uuid

from tooz import coordination

# Generate a random unique identifier using UUID
identifier = str(uuid.uuid4())
# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost", identifier)
# Start it (initiate connection)
c.start(start_heart=True)
# Stop when we're done
c.stop()
```

Note

You will need to start etcd to make the examples work. Simply run `etcd` from the command line once it is installed. If you do not have `etcd', you can use memcached instead.
Replace the `etcd://` URL with `memcached://localhost`.

The basic operations on a coordinator are pretty simple. Once instantiated, one just needs to start it and stop it when the program is done.

The `get_lock` method provided by the coordinator allows getting a distributed lock from the selected backend. This lock implements two main methods: `acquire` and `release`. They both return `True` or `False` based on their success or failure to operate.

Example 7.12. Getting a lock

```python
import uuid

from tooz import coordination

# Generate a random unique identifier using UUID
identifier = str(uuid.uuid4())
# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost", identifier)
# Start it (initiate connection)
c.start(start_heart=True)

lock = c.get_lock(b"name_of_the_lock")
# Grab and release a lock
assert lock.acquire() is True
assert lock.release() is True

# You can't release a lock you did not acquire
assert lock.release() is False

assert lock.acquire() is True
# You can acquire without blocking and fail early
assert lock.acquire(blocking=False) is False
# Wait 5 seconds before giving up
assert lock.acquire(blocking=5) is False
assert lock.release() is True
```

```
# Stop when we're done
c.stop()
```

Example 7.12, "Getting a lock" shows how you can use those methods to acquire and release a lock. The `acquire` lock method accepts a `blocking` parameter which is `True` by default. This makes the caller wait until the lock is available – which can take forever. If forever is too long for the program, `blocking` can also be set to the number of seconds (or `False`, which is equivalent to zero second) to wait before either succeeding and returning `True`, or giving up and returning `False`.

It is therefore easy to rewrite Example 7.9, "Locking with etcd using the `with` statement" using *Tooz* and getting the multiple backend support (which includes *etcd*) for free, as you can see in Example 7.13, "Using *Tooz* lock and the `with` statement."

Example 7.13. Using *Tooz* lock and the `with` statement.

```
import uuid

from tooz import coordination

# Generate a random unique identifier using UUID
identifier = str(uuid.uuid4())
# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost", identifier)
# Start it (initiate connection)
c.start(start_heart=True)

lock = c.get_lock(b"foobar")
with lock:
    print("do something")
```

Unless you are certain about the backend that your application needs,

using an abstraction layer such as *Tooz* can be a great solution. This is especially true for some of the backends that *Tooz* supports such as *PostgreSQL* or *memcached*, for example, as there are no other Python libraries implementing a locking mechanism in contrast to *etcd* or *ZooKeeper*.

Chapter 8. Group membership

Distributed applications all show the same set of well-known problems that they need to solve. One of them is distributed locking of resources so that no resource inadvertently gets shared access as this was discussed in Chapter 7, *Lock Management*. The second one involves cluster membership awareness and liveliness.

A typical use case for distributed workers is the need to know how many workers are alive in the cluster. This data allows for implementing different scenarios, such as high availability (one node doing the work and the other one standing-by) and load balancing (several nodes spreading some work across each other).

In all cases, an external service is needed to achieve this functionality. Rather than picking one and implementing all those low-level features in your application, this chapter demonstrates how to use the abstraction layer and algorithms that the *Tooz* library provides to solve that efficiently.

8.1. Creating, Joining and Leaving Groups

The *Tooz* library implements a common pattern that it calls "group membership". The pattern is quite easy to understand and is illustrated in Figure 8.1, "*Tooz* membership".

Figure 8.1. *Tooz* membership
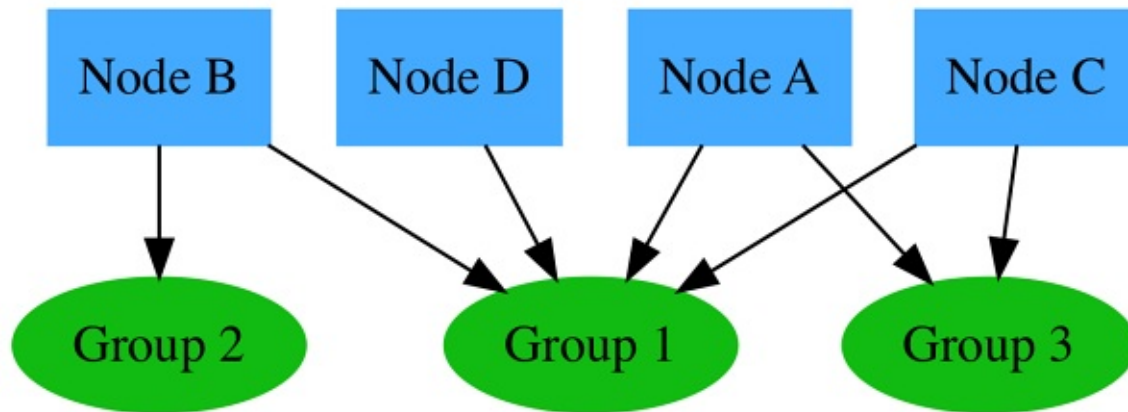
Here are the rules: a group is a set containing zero or more members. When a node joins a group, it becomes a member of this group. A member can leave the group at any time: either on its own (shutdown) or because it did not renew membership. The membership tracking is automatically done by *Tooz* and the backend it uses.

Note

As discussed in [Section 7.4, "Using *Tooz* Locking Abstraction"](#), *Tooz* supports several backends, from *memcached* to *etcd*. The robustness of the membership and liveliness tracking depends on the integrity of the backend. Using a backend such as *ZooKeeper* and its *Paxos* consensus algorithm is more robust than using a solo *Redis* instance. I invite you to dig look at requirements and dig into all the features provided by backends supported by *Tooz* for more information.

Example 8.1. Joining a group

```
import sys
import time

from tooz import coordination

# Check that a client and group ids are passed as arg
if len(sys.argv) != 3:
    print("Usage: %s <client id> <group id>" % sys.a
```

```
    sys.exit(1)

# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost",
    sys.argv[1].encode())
# Start it (initiate connection).
c.start(start_heart=True)

group = sys.argv[2].encode()

# Create the group
try:
    c.create_group(group).get()
except coordination.GroupAlreadyExist:
    pass

# Join the group
c.join_group(group).get()

# Print the members list
members = c.get_members(group)
print(members.get())

# Wait 5 seconds
time.sleep(5)

# Leave the group
c.leave_group(group).get()

# Stop when we're done
c.stop()
```

Example 8.1, "Joining a group" shows a complete workflow using *Tooz* group membership API.

Note

You will need to start *etcd* to make the examples work. Simply run `etcd` from the command line once it is installed. If you do not have `etcd', you can use memcached instead.
Replace the `etcd://` URL with `memcached://`.

The first step after starting the coordinator is to create the group. It is possible that the group has been already created, in that another client could have already been joining that group. If that is the case, we just ignore the exception.

The second step is to join the group. That is done using the `join_group` method. You might have noticed that the `get` method is called every time on the result of the `join_group` or `create_group` methods. Indeed, *Tooz* exposes an asynchronous API; that means that calling `join_group` starts the process of joining the group, but the group is not fully joined for sure until the application calls the `get` method of the returned value of `join_group`.

Once the group is joined, you can print the list of the members that are in the group. That is done by using the `get_members` method.

As the example allows you to specify the group id and the member id, you can run any number of those examples on any number of nodes. As long as they connect to the same *etcd* service, they see each other in the same group.

8.2. Using Capabilities

Tooz provides a capability mechanism on top of the group feature. When joining a group, a member can specify what its capabilities are. It can also update them later as needed.

Capabilities are any simple data structure that your program needs to share. It can be as simple as a string and as a complicated as a nested dictionary. The typical usage of this data structure is to pass information about the member, such as its processing power or its feature set. It

allows filtering members based on their functionality or weighing them differently based on their computing power.

Example 8.2, "Using *Tooz* capabilities" shows a complete example of joining a group with a capability set. In this example, each client can provide a *mood* when joining a group. All members can then get member capabilities and retrieve their moods.

Example 8.2. Using *Tooz* capabilities

```
import sys
import time

from tooz import coordination

# Check that a client and group ids are passed as arg
if len(sys.argv) != 4:
    print("Usage: %s <client id> <group id> <mood>"
          % sys.argv[0])
    sys.exit(1)

# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost",
    sys.argv[1].encode())
# Start it (initiate connection).
c.start(start_heart=True)

group = sys.argv[2].encode()

# Create the group
try:
    c.create_group(group).get()
except coordination.GroupAlreadyExist:
    pass

# Join the group
```

```
c.join_group(
    group,
    capabilities={"mood": sys.argv[3]}
).get()

# Print the members list and their capabilities.
# Since the API is asynchronous, we mit all the reque
# at the same time so they are run in parallel.
get_capabilities = [
    (member, c.get_member_capabilities(group, member)
    for member in c.get_members(group).get()
]

for member, cap in get_capabilities:
    print("Member %s has capabilities: %s"
          % (member, cap.get()))

# Wait 5 seconds
time.sleep(5)

# Leave the group
c.leave_group(group).get()

# Stop when we're done
c.stop()
```

When [Example 8.2, "Using *Tooz* capabilities"](#) is run twice at the same time, you can see the following output:

```
$ python examples/tooz-capabilities.py foo group1 sad
$ python examples/tooz-capabilities.py bar group1 hap
Member b'foo' has capabilities: {'mood': 'sad'}
Member b'bar' has capabilities: {'mood': 'happy'}
```

You can run this program as many time as you want, and use different groups if you need to. That should give you a glimpse of all the possibility

that this API can offer for your distributed application.

8.3. Using Watchers Callbacks

When a member joins or leaves a group, applications usually want to run an action. There is a mechanism provided by *Tooz* to help with that named *watchers*. It works by caching a list of the members and running the specified callback functions each time a member joins or leaves the group.

To use those callbacks, *Tooz* provides the `watch_join_group` and `watch_leave_group` methods to register functions to call on join or leave events. If you ever need to un-register a callback, `unwatch_join_group` and `unwatch_leave_group` provide this functionality.

Once the callback functions are registered, they are only run when the `run_watcher` method is called. If your application is thread-safe, you can run this method regularly in a different thread. If not, you should call it in any loop your program provides.

[Example 8.3, "Using watchers with *Tooz*"](#) provides an example of an application joining a group and checking when a member joins or leaves that group. As soon as this happens and that `run_watchers` is executed, it prints which member joined or left which group.

Example 8.3. Using watchers with *Tooz*

```
import sys
import time

from tooz import coordination

# Check that a client and group ids are passed as arg
if len(sys.argv) != 3:
    print("Usage: %s <client id> <group id>" % sys.ar
```

```python
        sys.exit(1)

    # Get the Coordinator object
    c = coordination.get_coordinator(
        # Set a short timeout to see effects faster
        "etcd3://localhost/?timeout=3",
        sys.argv[1].encode())
    # Start it (initiate connection).
    c.start(start_heart=True)

    group = sys.argv[2].encode()

    # Create the group
    try:
        c.create_group(group).get()
    except coordination.GroupAlreadyExist:
        pass

    # Join the group
    c.join_group(group).get()

    # Register the print function on group join/leave
    c.watch_join_group(group, print)
    c.watch_leave_group(group, print)

    while True:
        c.run_watchers()
        time.sleep(1)

    # Leave the group
    c.leave_group(group).get()

    # Stop when we're done
    c.stop()
```

To see the effect of this example, you should run it twice (at least) in two different terminals. That makes it possible to see the callback function

being called. An example transcript is provided in [Example 8.4, "Example output of Example 8.3, "Using watchers with *Tooz*""](#).

Example 8.4. Example output of [Example 8.3, "Using watchers with *Tooz*"](#)

```
# Run the first client in a first terminal
$ python examples/tooz-run-watchers.py client1 group

# In another terminal run the second client
$ python examples/tooz-run-watchers.py client2 group

# In the first terminal you will see
<MemberJoinedGroup: group group1: +member client2>

# Interrupt the client from the second terminal by pr
# You will see in the first terminal ~3 seconds later
<MemberLeftGroup: group group1: -member client2>
```

As soon as `Ctrl+c` is pressed to interrupt the program, it loses its connection to *etcd* and its keys will expire after the configured timeout. You can see the configured timeout as being passed in the connection URL as `?timeout=`.

Using `Ctrl+c` to interrupt the program simulates a crash or a violent interruption, and you can see that this system is pretty robust in this regard. Obviously, the program would also be notified of the member leaving the group if it would have called the `leave_group` method.

This feature allows following all members joining and leaving a distributed system and it is very handy in higher-level applications, as discussed in [Section 8.5, "Partitioner"](#).

8.4. Consistent Hash Rings

Being able to manage distributed groups enables the utilization of several

distributed algorithms. One of them is named consistent hash rings.

A traditional way to spread keys across a distributed systems, which is composed of `n` nodes, is to compute which node should be responsible for a key by using: `hash(object) % n`. Unfortunately, as soon as `n` changes, the result of the modulo operation changes and therefore all keys are remapped to a new node. This remapping causes a massive shuffling of data or processing in a cluster.

The point of consistent hashing is to avoid that. By using a different method of computing, when `n` changes, only `K / n` of the keys are remapped to the remaining nodes (where `K` is the number of keys and `n` the number of nodes handling those keys).

A hash ring is a hashing space that wraps around itself to form a circle – that's why it is called a ring. Every key computed using the consistent hashing function maps somewhere on this hash space: that means that a key is always in the same place on the ring. The ring is then split into `P` partitions, where `P` is a magnitude larger than the number of nodes. Each node is then responsible for `1 / n` partitions of the ring.

This implementation also has the upside of making it easy to add a replication mechanism, meaning a set of keys managed by more than just one node. Replication is handy in case of the failure of a node, as the keys are still managed/stored by another node.

Figure 8.2. Consistent hash ring

A partition

node 0

node 1

node 2

node 3

a ring with 32 partitions

Hash(object) % partitions
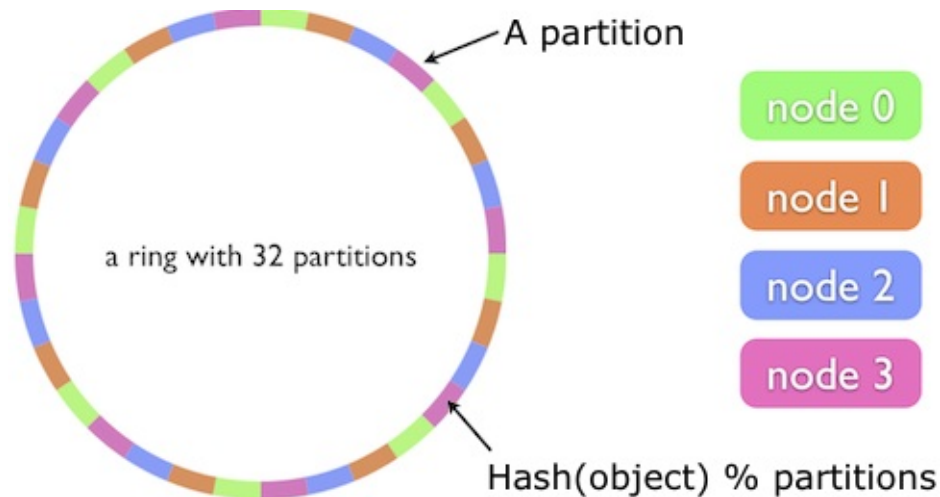
As we saw just before in Section 8.1, "Creating, Joining and Leaving Groups", it is easy to leverage *Tooz* to accurately know which nodes are alive. That is why *Tooz* also provides a hash ring implementation that can be used to map objects onto nodes, with as little re-balancing in case a node leaves or joins a group.

Example 8.5. Using *Tooz* hashring

```
# -*- encoding: utf-8 -*-
from tooz import hashring


NUMBER_OF_NODES = 16

# Step #1 - create a hash ring with 16 nodes
hr = hashring.HashRing(["node%d" % i for i in range(N
nodes = hr.get_nodes(b"some data")
print(nodes)
nodes = hr.get_nodes(b"some data", replicas=2)
print(nodes)
nodes = hr.get_nodes(b"some other data", replicas=3)
print(nodes)
nodes = hr.get_nodes(b"some other of my data", replic
print(nodes)
```

```python
# Step #2 - remove a node
print("Removing node8")
hr.remove_node("node8")
nodes = hr.get_nodes(b"some data")
print(nodes)
nodes = hr.get_nodes(b"some data", replicas=2)
print(nodes)
nodes = hr.get_nodes(b"some other data", replicas=3)
print(nodes)
nodes = hr.get_nodes(b"some other of my data", replic
print(nodes)

# Step #3 - add a new node
print("Adding node17")
hr.add_node("node17")
nodes = hr.get_nodes(b"some data")
print(nodes)
nodes = hr.get_nodes(b"some data", replicas=2)
print(nodes)
nodes = hr.get_nodes(b"some other data", replicas=3)
print(nodes)
nodes = hr.get_nodes(b"some other of my data", replic
print(nodes)
nodes = hr.get_nodes(b"some data that should end on r
print(nodes)

# Step #4 - add back a node with a greater weight
print("Adding back node8 with weight")
hr.add_node("node8", weight=100)
nodes = hr.get_nodes(b"some data")
print(nodes)
nodes = hr.get_nodes(b"some data", replicas=2)
print(nodes)
nodes = hr.get_nodes(b"some other data", replicas=3)
print(nodes)
nodes = hr.get_nodes(b"some other of my data", replic
print(nodes)
```

[Example 8.5, "Using *Tooz* hashring"](#) shows how to use the `HashRing` object provided by *Tooz* and what you can achieve with it. We will cover the example step by step, so it is easy to understand.

```
NUMBER_OF_NODES = 16

# Step #1
hr = hashring.HashRing(["node%d" % i for i in range(N
hr.get_nodes(b"some data")
hr.get_nodes(b"some data", replicas=2)
hr.get_nodes(b"some other data", replicas=3)
hr.get_nodes(b"some other of my data", replicas=2)
## Output:
# {'node8'}
# {'node8', 'node11'}
# {'node6', 'node2', 'node13'}
# {'node8', 'node7'}
```

The first step of this example demonstrates how to create a hash ring. The hashring created has 16 initial nodes, named `node1` to `node16`.

Once the hash ring is created, the main method for using it is `get_nodes`. It expects bytes as input. It is up to the application developer to come up with a chain of bytes that makes sense in his application. It could be a simple key or a stringified version of an object or its hash.

The return value of `get_nodes` is a set of nodes that are responsible for handling this piece of data. By default, only one node is returned, but the method also accepts a number of replicas as an argument. In this case, the returned set will contain `R` more nodes, where `R` is the number of replicas.

```
# Step #2 - remove a node
hr.remove_node("node8")
hr.get_nodes(b"some data")
```

```
hr.get_nodes(b"some data", replicas=2)
hr.get_nodes(b"some other data", replicas=3)
hr.get_nodes(b"some other of my data", replicas=2)
## Output:
# Removing node8
# {'node11'}
# {'node6', 'node11'}
# {'node6', 'node2', 'node13'}
# {'node5', 'node7'}
```

The second step of Example 8.5, "Using *Tooz* hashring" removes a node from the hash ring. At this stage, there are only 15 nodes in this ring. The `get_nodes` calls are identical to step 1, but as you can see the output is quite different. Since we removed `node8` from the hash ring, the partitions that it managed are now handled by the nodes managing the adjacent partitions in the ring.

For the first key, `node8` is replaced by `node11`. For the second key, `node6` is used instead. There is no change for the third key as `node8` was not a replica. Finally, `node5` is picked for the last key instead of the missing `node8`.

As you can see, the promise of the hash ring is kept. One node has been removed, but only some of the keys were remapped, exclusively the ones being assigned to the removed node.

```
# Step #3 - add a new node
print("Adding node17")
hr.add_node("node17")
hr.get_nodes(b"some data")
hr.get_nodes(b"some data", replicas=2)
hr.get_nodes(b"some other data", replicas=3)
hr.get_nodes(b"some other of my data", replicas=2)
hr.get_nodes(b"some data that should end on node17",
## Output:
# Adding node17
```

```
# {'node11'}
# {'node6', 'node11'}
# {'node6', 'node2', 'node13'}
# {'node5', 'node7'}
# {'node17', 'node9'}
```

The third step adds a new node called `node17` to the ring. Once again, the promise is kept, and no re-balancing of the key we previously showed was done. To show that the `node17` was indeed responsible for some partitions, I have added a `get_node` with a byte string where a replica is `node17` (as there is no way to know in advance where a key will end up, I have just edited the key until the returned node would be `node17`).

```
# Step #4 - add back a node with a greater weight
print("Adding back node8 with weight")
hr.add_node("node8", weight=100)
hr.get_nodes(b"some data")
hr.get_nodes(b"some data", replicas=2)
hr.get_nodes(b"some other data", replicas=3)
hr.get_nodes(b"some other of my data", replicas=2)
## Output:
# Adding back node8 with weight
# {'node8'}
# {'node11', 'node8'}
# {'node2', 'node8', 'node6'}
# {'node8', 'node7'}
```

In the last step, `node8` is added back to the hash ring, but this time with a weight of 100, which means it will be responsible for up to 100 times more keys than the other nodes. As the hash ring is deterministic, the key that `node8` was responsible for before being removed are returned to it. That means the key `some data` causes `node6` to be replaced by `node8`, in the same spot it was at the beginning of the program. Since the weight is 100, that also means that this time `node8` will get a lot more keys than the others. That is why you can notice that `node8` is now one of the

replica of the key `some other data` instead of `node13`.

Hash rings are very convenient though they are not perfect. For one, the distribution of keys is not uniform, and some nodes will be responsible for more keys than others. This might or might not be a problem depending on your application. Nonetheless, it still offers interesting properties and once coupled with cluster membership, offers valuable solutions as we will see in Section 8.5, "Partitioner".

8.5. Partitioner

Now that you know how a hash ring works (see Section 8.4, "Consistent Hash Rings") and how to manage group memberships (see Section 8.1, "Creating, Joining and Leaving Groups"), we can start thinking about mixing the two.

On one side, we have a group system that can tell us which nodes in our distributed system are live, and on the other side, we have an object that tells us which node handle a piece of data. By updating the hash ring members using the group system that *Tooz* offers, we can construct a new object called a **partitioned group**.

Instead of building that into each application, *Tooz* provides an API that can be leveraged while relying on both those mechanism. The `join_partitioned_group` method allows an application to join a group in which all members share some workload using a consistent hash ring.

Example 8.6. Using *Tooz* `join_partitioned_group` method

```
import sys
import time

from tooz import coordination

# Check that a client and group ids are passed as arc
if len(sys.argv) != 3:
```

```
    print("Usage: %s <client id> <group id>" % sys.ai
    sys.exit(1)

# Get the Coordinator object
c = coordination.get_coordinator(
    "etcd3://localhost",
    sys.argv[1].encode())
# Start it (initiate connection).
c.start(start_heart=True)

group = sys.argv[2].encode()

# Join the partitioned group
p = c.join_partitioned_group(group)

print(p.members_for_object("foobar"))

time.sleep(5)

# Leave the group
c.leave_group(group).get()

# Stop when we're done
c.stop()
```

The `Partitioner` object provides a `members_for_object` which
returns a set of members responsible for an object.

Note

To compute a unique byte identifier for a Python object, the *Tooz*
partitioner calls the `__tooz_hash__` method on the object
passed as argument to `members_for_object`. If this method
does not exist, the standard `hash` Python module is called
instead. Having this method defined is important because each
object must be uniquely but consistently identified across the
cluster.

You can run [Example 8.6, "Using *Tooz* `join_partitioned_group` method"](#) in parallel in different terminals. You'll see that they will join the same group, and output the same member id:

```
$ python examples/tooz-join-partitioned-group.py clie
{b'client1'}
$ python examples/tooz-join-partitioned-group.py clie
{b'client1'}
```

That will obviously only work if both clients are connected at the same time and are part of the same group. If you run `client2` alone, the output will be `client2`, as it is the only member of the group.

Note

In order to work correctly, the *Tooz* partitioner relies on the watchers (as seen in [Section 8.3, "Using Watchers Callbacks"](#)). Do not forget to regularly call `tooz.coordination.Coordinator.run_watchers` so the hash ring of the partitioner is kept aware of members joining and leaving the partitioned group.

This mechanism is quite powerful and can solve some problems.

For example, you can automatically divide up the workload across several nodes. Imagine you have 100 URLs to continuously fetch from the Web. You can spread that workload pretty easily using this mechanism.

Example 8.7. Using hash ring to spread Web pages fetching

```python
import itertools
import uuid

import requests
from tooz import coordination
```

```python
class URL(str):
    def __tooz_hash__(self):
        # The unique identifier is the URL itself
        return self.encode()


urls_to_fetch = [
    # Return N bytes where the number of bytes
    # is the number at the end of the URL
    URL("https://httpbin.org/bytes/%d" % n)
    for n in range(100)
]

GROUP_NAME = b"fetcher"
MEMBER_ID = str(uuid.uuid4()).encode('ascii')

# Get the Coordinator object
c = coordination.get_coordinator("etcd3://localhost",
# Start it (initiate connection)
c.start(start_heart=True)

# Join the partitioned group
p = c.join_partitioned_group(GROUP_NAME)

try:
    for url in itertools.cycle(urls_to_fetch):
        # Be sure no membership changed
        c.run_watchers()
        # print("%s -> %s" % (url, p.members_for_obje
        if p.belongs_to_self(url):
            try:
                r = requests.get(url)
            except Exception:
                # IF an error occur, just move on
                # to the next item
                pass
```

```
            else:
                print("%s: fetched %s (%d)"
                    % (MEMBER_ID, r.url, r.status_c
    finally:
        # Leave the group
        c.leave_group(GROUP_NAME).get()

        # Stop when we're done
        c.stop()
```

In Example 8.7, "Using hash ring to spread Web pages fetching", the
program connects to the coordinator, joins a group named `fetcher` and
then starts to iterate on the pages to retrieve. While the program is the
only one connected to the coordinator, it will fetch every page, as can be
seen in Example 8.8, "Running only one Web page fetching program".

Example 8.8. Running only one Web page fetching program

```
b'8e6d3ff7-0dc7-4a71-aa39-8b1405cbc064': fetched http
b'8e6d3ff7-0dc7-4a71-aa39-8b1405cbc064': fetched http
b'8e6d3ff7-0dc7-4a71-aa39-8b1405cbc064': fetched http
[...]
```

As soon as a second instance of the program starts and joins the hash
ring, the first instance of the program starts skipping some of the pages.
Those skipped pages are fetched by the other running program as you can
see in the output of Example 8.9, "Running two Web page fetching
programs".

Example 8.9. Running two Web page fetching programs

```
# Output from first instance:
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
```

```
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
b'24ed57b0-5a8b-4497-9661-24b2d99e0cdb': fetched http
(200)
[…]
# Output from second instance:
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
c40306bd-1e5b-4287-8f6d-1477ba98403a: fetched https:/
[…]
```

As soon as a member joins the group, the members spread the URLs to fetch between them using the hash ring; therefore, it all goes twice as fast as before.

Of course, this kind of mechanism does not fit every workload. Non-cyclic workloads which need a queue are still better handled by queue mechanism as discussed in Chapter 5, *Queue-Based Distribution*. However, it is easy to imagine using that mechanism to distribute the consuming of multiple queues across workers.

8.6. Alexys Jacob-Monier on Cluster Management

Hi Alexys! Could you introduce yourself and explain how you came to Python?

I tend to describe myself as an autodidact who spent more time trying to figure out how to make something out of his machines than actually making them work. The constant struggle of the fail and retry strategy taught me better how not to build things than how to build them right the first try. I guess this is why I define empiricism as the art of frustration management!

I started my professional life as a Flash web developer also doing ASP/C# at 1000mercis which was a digital advertising and marketing startup at the time. Since we were satisfied, I started multitasking a lot, and my job shifted to be a network administrator added to a system administrator before I became an accidental programmer after building a somewhat elaborate email parsing platform using bash (1000+ lines of code).

This insanity made me realize that I needed a programming language that would fit my empirical approach as well as my broad range of use cases.

I first heard about Python through my interest and heavy usage of Gentoo Linux which I brought to production state at work. Then a Linux Magazine introducing Python came into my mailbox, and I decided to give it a shot. It was a shock. All I needed was there, and I felt so galvanized that I started (re)writing everything using Python!

Nowadays I am CTO at 1000mercis, which scaled into an international group known as Numberly while embracing Linux and Python as its central technological keystones. I am very fortunate to work with a lot of talented people who uses the language to build distributed, reliable and performant data-oriented platforms at scale.

I am also part of the Open Source software community. I have been a Gentoo Linux developer since 2011 with a particular interest in packaging and contributing to clustering and NoSQL related projects and their corresponding Python libraries. I have also given a few talks at EuroPython conferences related to distributed applications and systems.

Over the last couple of years, I have also been extremely pleased to see Python becoming the reference language for data science and engineering!

What do you think makes Python great (or not) when building distributed systems? What are the things you consider to be advantages or drawbacks?

At the risk of disappointing you, I actually think that Python is a mediocre language to build distributed systems!

To be fair, this statement really depends on how far you want your application to handle all the complexity that comes with distributed

systems such as inter-node communication, fault tolerance, partial failure management, split brain situations, consistency vs. availability constraints and so on.

Python can be fabulous if you want to build a distributed application which relies on a set of robust services such as distributed coordinators, key/value stores, message brokers, metrics stores, and databases. Those distributed services implement and provide features such as leader election, distributed locks, service discovery, message queuing/streaming and metrics graphing. They can serve as the building blocks of your distributed application.

The immediate drawback is that you have to set up, operate and maintain those external services before you start working on your application. They become a firm dependency, and their features (or lack of them) dictate what your application is capable of doing or not. While this is a general acceptance for databases, it may be not so innocent for distributed coordinators, and those operational costs could be a problem and a burden depending on the maturity of your organization.

If you are ready to accept that though, the advantage of Python is that you can concentrate on the application logic, and build a prototype quickly. Most importantly you have a great chance of finding suitable client libraries to rely on to interact with all those external components and services. Python's influence is powerful enough that bindings/libraries/drivers are almost always available for widespread or serious projects nowadays.

It makes Python a great client oriented language for building the business logic side of a distributed application. However, this also

means that its internals and ecosystem are not good enough to build the core components needed by distributed systems.

Why is that so?

Well, let's start by facing some facts about the distributed components I have been talking about so far.

Distributed coordinators:

- Zookeeper is written in Java
- Etcd is written in Go
- Consul is written in Go

Distributed key/value stores and databases:

- Redis is written in C
- MongoDB is written in C++
- Cassandra is written in Java
- Cockroachdb is written in Go
- Riak is written in Erlang

Distributed message brokers:

- Kafka is written in Java
- RabbitMQ is written in Erlang

Distributed time-series databases:

- Graphite is written in Python
- InfluxDB is written in Go

We can reasonably think that if Python were great to build

distributed systems, it would have been used to build more than one of those distributed server projects right? I mean, our community is made of equivalently brilliant people in Java or Go... still, would you imagine building a distributed and highly available database using Python?

In my opinion, the main Python weaknesses related to this fact are its poor packaging and concurrency management. A quick look at major Python conferences' schedules suffices to see that they are still hot topics in 2017!

- Python has no straightforward and standardized way to be packaged, deployed and run.
- Efficient and robust concurrent coding is too new (available since version 3.5) and still too low level to be efficiently and widely used
- The Global Interpreter Lock is still here and no Just In Time compiler is in the process of replacing CPython (even if the work of the PyPy team is impressive!)

Go is becoming the leading language there because it does not suffer from those problems and also because it is often seen as an efficient compromise between Java, Erlang, and Python.

Another drawback is that as a community, we have not developed a strong enough asset of higher-level libraries to ease the development of distributed systems. We have almost no mature, community supported libraries/projects providing reliable multi-node communication, coordination, and distribution.

Then we also have to consider lower level libraries implementing research papers about distributed hash tables (DHT) and consensus algorithms (PAXOS, Raft) for example. I am mentioning this because

I like the idea of having a consensus algorithm implemented at the application level. I encourage you to read [this good explanation about "application level consensus" development](): I would love Python to be a real player there.

Because acting is better than ranting, I have created the [uhashring]() project to provide a feature-rich library you can rely on to make use of consistent hashing in a distributed application. The [OpenStack tooz]() project also offers a consistent hashing library with a bit less flexibility.

Consensus algorithm implementations in Python are almost nonexistent and not meant for production use today. Since this is usually a keystone component of any decent distributed system, I guess we still have work to do to get there. I badly want to create a solid Raft library, so if I ever find the time for this, I would be happy to try to fill this gap, while hoping secretly for a smarter fellow developer to take on the challenge before I do!

To finish on positive a note, I want to mention and thank all the contributors of the Python-based Graphite and Grafana projects. Their marvelous work allows Python to shine in one of the most critical components of distributed systems: metrics storage and graphing. While metrics are essential for any serious project, they are fundamental to monitor and understand what's going on in a distributed system, so thank you again for your work!

Finally, there are three very exciting Python-based projects that I want to highlight:

- The OpenStack tooz project that I mentioned earlier provides a good set of tools to build distributed systems relying on external

components.

- The RPyC project is elegant and offers an RPC-like way to orchestrate a computation on multiple machines.

- The [dask](#) project is as promising as it is unusual for parallel computation in the data science world. I have the feeling that its core is strong enough to be abused to do way more than data science related distributed computation.

What would be your the top N mistakes to avoid or advice/best practice to follow when building scalable, distributed systems using Python?

- Try to avoid building distributed systems as much as you can! No, most of us do not need a fully distributed application and the complexity that comes with it. Do not mistake scaling out with distributing your application.

- Resist the hype behind the "distributed system" words. Leave your ego in the locker room and try hard to find all the alternatives that you have before considering moving on. I would rather run a "standard" stack that delivers just fine than a "distributed" stack that can quickly get out of control because my organization and myself were not prepared enough for it. Uptime and reliability are the words you are looking for.

- Scaling does not mean turning your application into a distributed system. While adding more Web servers behind your load balancer to scale out your Web application does not make it a distributed one, it is certainly the most sensible way to grow your Web project! If you chose your external components wisely (database, key/value store), they also should be able to scale out on their own before you need to distribute your application. If not, try to change them first!

- Be humble. You will unquestionably learn how not to build your distributed application after you've built it. You will make mistakes, bad judgments and fail to cover all the edge cases that can happen. Even Google did not make it right the first time!

- Design with data locality and isolation in mind. Distributing an application means relying on inter-machine communication using unreliable networks – yes, even your LAN. Networks being unstable means that you should work hard on avoiding data transfers as much as you can. Apply the UNIX philosophy carefully and do not fall into the microservices trap even if you want to be a cool kid too. I gave [a talk about scalability and distributed design considerations at EuroPython 2015](#).

- Meter everything. And I mean everything! Function execution time, the number of queries per second to every endpoint of your frontend, the queries per second and response time for your backends per database and table, overall processing time, every possible latency between components, and of course any other business related metric. Do not trust the averages, trust the percentiles and volumes comparison over time. Make it a clear and readable dashboard. Put it on a screen on the wall to have it under your nose at all times. Metering is also essential for capacity planning and monitoring; you will not regret it.

- Failing is an art. When you have entered the distributed systems world, "what if this component fails?" becomes the question that you ask yourself most of the time. Fault tolerance is not only about not losing data, but it is also about what you decide to show or not to the user in the face of a problem for example. Failure management requires careful consideration.

- Plan for the worst… because it will happen, sooner than you thought. When the shit hits the fan, you should have worked out how long your system can sustain a failure. You will want to implement mechanisms to mitigate the impacts or to allow your system to continue working in a degraded way. Degrading fast and smoothly is often better than crashing late and hard.

- Get a Python code deployment strategy. Distributed systems are living systems. Rolling out a new version of your code can have dreadful impacts so consider your deployment strategy very carefully. Your application should be able to reload gracefully with failsafe timeouts and in a rolling manner. It allows for fast and

safe rollbacks when you see errors popping out. Progressive deployment on your nodes means that your releases should always be backward compatible. Sometimes this requires multi-stage deployments to roll out a somewhat breaking change. There are tools, stacks and application containers that can help you on this quest. Kubernetes, Gitlab, and uWSGI are my favorites so far.

If you are still not disgusted at the idea of building a distributed system, then my last advice is to step up and help the Python community address the issues and challenges I listed before. We need you, dear reader!

Chapter 9. Building REST API

When building applications using service-oriented architecture, you have to pick the protocol to use to communicate between your services. There is a variety of protocols available out there. You could also probably roll your own, though, in most cases, it is not a good idea.

The HTTP protocol has been the Web standard for the last 25 years, and it does not seem like it is about to change anytime soon. It certainly has a few drawbacks but also has the amazing advantages being massively deployed, simple to understand and simple to debug for humans. It is also cache-able and easy to transport on most networks: it is rarely blocked. The number of tools surrounding HTTP is huge, making it a perfect candidate for rapid development and easy debugging.

A REST Web service (based on HTTP) has the advantage of being stateless. This property is aligned with the service-oriented architecture described in [Section 1.3, "Service-Oriented Architecture"](). Therefore, this chapter covers the basics of building a REST Web service using Python along with what you need to know in order to make it scale properly.

Tip

There is no single source of truth on best practice for REST API. The first thing to read and follow is the HTTP RFC: this is the absolute references that you cannot break, no matter what. There are also a few other references that you should check, like [HAT OAS](), the [OpenAPI Initiative]() or the [OpenStack API Working Group Guidelines](). These documents are a good source of ideas before deciding on how to design your API endpoints. They answer a lot of common questions around creating semantically accurate APIs.

The software programming world is not short on frameworks for building a REST API, and Python is no exception. It has many Web libraries that make it possible to achieve this goal.

Note

The most popular framework in the Python world, Django, even has a layer offering this feature and it's called Django REST Framework. I am not a particular fan of Django, which while being nice to build Web sites, seems cluttered for this task. This is a matter of personal preference, and I do not see anything wrong about building a REST API with Django. The same might be said about many other Python frameworks out there.

Flask is one of the most used Web frameworks in the Python ecosystem. It is a good pick for building a REST API: it is rather lightweight, modular, extendable, and it does provide basic functionality. This is why it is used in this chapter.

Tip

Many examples in this chapter use the `http` command line tool to interact with HTTP REST APIs. This tool comes from httpie, a Python command line interface that is simple to use for humans, making it easy to interact with Web servers, and it has colorized output.

9.1. The WSGI Protocol

Before considering a REST API, one needs to know the first layer of abstraction of the Python world when HTTP is involved: WSGI.

WSGI stands for *Web Server Gateway Interface*. It started its life as part of PEP 0333, and was updated as part of PEP 3333. This PEP was designed to solve the problem of mixing frameworks and Web servers. It makes sure there is a common protocol between Web servers and Web frameworks, so they are not necessarily tied together. Indeed, it would be a shame to be forced to provide a Web server for each framework, wherein there is already a vast collection of probably better alternatives out there.

The WSGI protocol is pretty easy to understand, and this understanding is also valuable as all Python frameworks are based on it.

When a WSGI Web server loads an application, it looks for an application object that is callable. Calling this object must return a result that is shipped back to the HTTP client.

The callable must be named `application` and will receive two arguments: a dict filled with environment keys and values named `environ`, and a function named `start_response`. The WSGI application must use the latter to send the status and headers reply to the client, as demonstrated in Example 9.1, "Basic WSGI application".

Example 9.1. Basic WSGI application

```
def application(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain'
    start_response(status, response_headers)
    return ['Hello world!\n']
```

A Web development framework plugs itself in at this stage by providing the `application` object and letting the developer concentrate of the implementation of its business logic.

The WSGI specification has been built so applications implementing the WSGI protocol could be stacked. Some applications implement both sides of the WSGI protocol and are called *middleware*: that means they can handle a request and then pass it to the next WSGI application in the *pipeline* (if needed). You can, therefore, chain WSGI middlewares to do pre-processing (e.g., ACL management, rate limiting, etc.) until the pipeline reaches the actual WSGI application.

To serve a WSGI application, there are many different Web servers that are available. First off there is Python's own `wsgi` module, which has a built-in Web server that can be used as shown in Example 9.2, "Basic

WSGI application with `wsgiref.simple_server`".

Example 9.2. Basic WSGI application with `wsgiref.simple_server`

```
from wsgiref.simple_server import make_server


def application(environ, start_response):
    """Return the environ keys as text/plain"""
    body = '\n'.join([
        '%s: %s' % (key, value) for key, value in so
    ])

    start_response("200 OK", [
        ('Content-Type', 'text/plain'),
        ('Content-Length', str(len(body)))
    ])

    return [body]


# Instantiate the server
httpd = make_server('localhost', 8051, application)
# Wait for a single request, serve it and quit
httpd.handle_request()
# Run `curl -v http://localhost:8051' to see the requ
```

While completely functional, the `wsgiref` server should probably be avoided for any serious use. It is very limited and neither offers good performance nor fine tuning, both of which are usually required when deploying production systems.

There are some other WSGI servers that you can use:

- The most famous is *Apache httpd* and its *mod_wsgi* companion. Both are well tested and have been supported for years, so this is one of the

safest choices. They allow a variety of combination when deploying, such as deploying several WSGI applications on the same port with different paths. A small downside though is that restarting the *httpd* process to reload the configuration usually restarts all the services – which might be a problem if you deploy several WSGI applications with the same *httpd* process.

- *Gunicorn* (Green Unicorn). It is relatively easy to use and deploy.

- *Waitress* is a pure-Python HTTP server.

- *uWSGI* is a very complete and very fast WSGI server – it is probably my favorite. It is a bit harder to use and configure as it presents a lot of configuration options and supports more than just WSGI. However, only a few options are needed to have a working application. It even supports HTTP 2 and other programming languages (Perl, Ruby, Go, etc.).

Example 9.3, "Using WSGI" shows how to deploy a simple application with *uWSGI*.

Example 9.3. Using WSGI

```
$ uwsgi --http :9090 --master --wsgi-file examples/ws
*** Starting uWSGI 2.0.14 (64bit) on [Mon Jan 16 21:3
nodename: abydos
machine: x86_64
clock source: unix
pcre jit disabled
detected number of CPU cores: 4
current working directory: /Users/jd/Source/scaling-p
detected binary path: /usr/local/Cellar/uwsgi/2.0.14/
your processes number limit is 709
your memory page size is 4096 bytes
detected max file descriptor number: 4864
lock engine: OSX spinlocks
thunder lock: disabled (you can enable it with --thun
uwsgi socket 0 bound to TCP address :9090 fd 4
Python version: 2.7.13 (default, Dec 20 2016, 16:45:1
```

```
*** Python threads support is disabled. You can enabl
Python main interpreter initialized at 0x7fcdca407670
your server socket listen backlog is limited to 100 o
your mercy for graceful operations on workers is 60 s
mapped 145520 bytes (142 KB) for 1 cores
*** Operational MODE: single process ***
WSGI app 0 (mountpoint='') ready in 0 seconds on inte
*** uWSGI is running in multiple interpreter mode ***
spawned uWSGI master process (pid: 86156)
spawned uWSGI worker 1 (pid: 86157, cores: 1)
# [ from another terminal ]
$ curl -v http://localhost:9090
* Rebuilt URL to: http://localhost:9090/
* Connected to localhost (127.0.0.1) port 9090 (#0)
> GET / HTTP/1.1
> Host: localhost:9090
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-type: text/plain
* no chunk, no close, no size. Assume close to signal
<
Hello world!
* Curl_http_done: called premature == 0
* Closing connection 0
```

The performance of your WSGI server is going to be important if you
need to scale your HTTP application to support a large number of clients.
There is certainly no go-to solution that can fit all use cases. You will have
to benchmark your different use cases and see what fit the best. If you do
not need any of the fancy features that *Apache httpd* provides, *uWSGI* or
Gunicorn are probably good picks.

Note

All the examples provided in this chapter can be run either using

the script directly or using *uWSGI* as described above.

9.2. Streaming Data

A common pattern with any HTTP API is the need to receive events. In a lot of cases, there's no way to achieve this other than polling the API regularly. That can cause a lot of stress on the HTTP endpoint, as it requires setting up a new connection – which means a lot of overload with TCP and SSL.

A more efficient way is to use streaming. An adequate technology here is **Server-Sent Events** message protocol defined by HTML5. Alternatively, it would be possible to use `Transfer-Encoding: chunked` defined by HTTP/1.1 or even the [WebSocket](#) protocol. However, chunked encoding is more complicated and the WebSocket a little bit overkill for the simple use case presented here.

To implement any streaming mechanism in a scalable and efficient manner, you need to be sure that your backend offers that feature. It could be a messaging queue, a database or any other software that provides a stream of events that the application can subscribe to.

If the offered API has to poll its backend regularly to know about new events, it is simply moving the problem from one layer to another. It is better than nothing, but it is far from ideal.

The example illustrated in this section is a small application that stores messages in *Redis* and provide access to those messages via an HTTP REST API. Each message consists of a channel number, a source string, and a content string. The backend used in the examples in this section is *[Redis](#)*, as it provides a notification mechanism that is close to what a message queue could offer.

The goal is to stream these messages to the client so that it can process them in real time on its side. To do this, we're going to use the *[Redis Pub/Sub](#)* mechanism provided by the `PUBLISH` and `SUBSCRIBE` commands.

These features allow us to subscribe and receive messages sent by other processes.

Example 9.4. The `PUBLISH` command

```python
import redis

r = redis.Redis()
r.publish("chatroom", "hello world")
```

Example 9.4, "The PUBLISH command" shows how to publish a message to a channel. The `publish` method sends the message to the channel passed as the first argument. The second argument is a string carrying the actual payload.

It is possible to check that the trigger works by using the SUBSCRIBE command. If everything is OK, it receives a notification as soon as the PUBLISH command is executed.

Example 9.5. Checking the `SUBSCRIBE` command

```
$ redis-cli
127.0.0.1:6379> SUBSCRIBE chatroom
Reading messages... (press Ctrl-C to quit)
1) "subscribe" (1)
2) "chatroom"
3) (integer) 1
1) "message" (2)
2) "chatroom"
3) "hello world"
```

(1) This is an automatic message sent by Redis to indicate that the subscription is working.

(2) This is our first actual message received.

Example 9.5, "Checking the SUBSCRIBE command" shows how to check

that the notification works. As soon as the message is sent it is received by the *Redis* client.

Example 9.6. Receiving messages in Python

The application provided in [Example 9.6, "Receiving messages in Python"](#) connects to *Redis* using the *pyredis* library. The program listens on the provided channel. As soon as it receives a notification, it prints it to the screen. Running the program outputs the content from [Example 9.7, "Output of `listen.py`"](#).

Example 9.7. Output of `listen.py`

```
$ python listen.py
{'pattern': None, 'type': 'subscribe', 'channel': 'ch
{'pattern': None, 'type': 'message', 'channel': 'chat
```

With that first brick in place, what is missing now is plugging that into a Web API. [Example 9.8, "*Flask* based streamer application"](#) is an implementation of this application, providing an endpoint for writing an endpoint that sends messages as a stream.

Example 9.8. *Flask* based streamer application

```python
import json

import flask
import redis


application = flask.Flask(__name__)


def stream_messages(channel):
    r = redis.Redis()
    p = r.pubsub()
```

```
        p.subscribe(channel)
        for message in p.listen():
            if message["type"] == "message":
                yield "data: " + json.dumps(message["data


@application.route("/message/<channel>", methods=['GF
def get_messages(channel):
    return flask.Response(
        flask.stream_with_context(stream_messages(cha
        mimetype='text/event-stream')


@application.route("/message/<channel>", methods=['PC
def send_message(channel):
    data = flask.request.json
    if (not data or 'source' not in data or 'content'
        flask.abort(400)
    r = redis.Redis()
    r.publish(channel, "<{}> {}".format(data["source'
    return "", 202
```

Warning

The above application needs at least two connections at the same
time when being used: one for streaming, and one for sending
messages. Therefore it requires a Web server that can handle
multiple connections at once, such as a *uWSGI*. The default Web
server provided by Flask would **not** work as it only handles one
connection at a time.

The first endpoint is accessible by calling `GET /message/<channel>`.
It returns a response with the mime type `text/event-stream`, sending
back a generator function instead of a string. Flask calls this function and
sends the results each time the generator yields something.

The generator, `stream_messages`, reuses the code we wrote earlier to

listen to *Redis* notifications. It receives the channel identifier as an argument, listens to that channel, and then yields the payload.

The second endpoint is accessible by calling `GET` `/message/<channel>`. It accepts a JSON payload that must contain the `source` and `content` field and that is then sent to *Redis*.

You can run the server using *uWSGI*:

```
$ uwsgi --http :5000 --master \
>   --workers 10 \
>   --wsgi-file http.py
```

On another terminal, an HTTP client can connect and retrieve the events as they are sent.

```
$ http --stream GET http://127.0.0.1:5000/message/cha
HTTP/1.1 200 OK
Content-Type: text/event-stream; charset=utf-8
```

In a third terminal, it is possible to then send a message using the writing endpoint:

```
http --json --stream POST http://127.0.0.1:5000/messa
HTTP/1.1 202 ACCEPTED
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

As soon as the request returns, it is possible to see messages coming into the terminal connected to the event stream:

```
$ http --stream GET http://127.0.0.1:5000/message/cha
HTTP/1.1 200 OK
Content-Type: text/event-stream; charset=utf-8
```

```
data: <jd> it works
```

A naive implementation of this application would instead continously loop over a query statement to poll for new data inserted in *Redis* – or any other database. However, as noted earlier, there is no need to demonstrate that a push system like this is much more efficient than constantly polling a database. This mechanism can be applied to many other backend systems, not only *Redis*. For example, *PostgreSQL* has support two commands LISTEN and NOTIFY that make it possible to implement this too.

Such an application is also horizontally scalable. As it is stateless, you can run as many HTTP servers as needed to handle the number of requests being received.

9.3. Using ETag

An ETag, abbreviated from *entity tag*, is a header part of the HTTP standard. It allows a client to make conditional requests using its cache, limiting bandwidth and usage of server resources.

When a client sends a request to a server, the latter can reply with a response including an ETag header. Common methods of ETag generation include using a collision-resistant hash function of the resource's content, a hash of the last modification timestamp, or even just a revision number.

Example 9.9, "ETag header" demonstrates a server reply with an ETag header.

Example 9.9. ETag header

```
$ http HEAD http://gnocchi.xyz
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: Keep-Alive
Content-Length: 15919
```

```
Content-Type: text/html; charset=UTF-8
Date: Tue, 12 Sep 2017 09:17:26 GMT
ETag: "3e2f-558f9d5587b40"
Keep-Alive: timeout=5, max=100
Last-Modified: Tue, 12 Sep 2017 08:28:53 GMT
Server: Apache
```

An application can use the ETag value to decide whether the page should be downloaded again using an `If-None-Match` header, as shown in Example 9.10, "Using `If-None-Match` header".

Example 9.10. Using `If-None-Match` header

```
$ http GET http://gnocchi.xyz If-None-Match:\"3e2f-55
HTTP/1.1 304 Not Modified
Connection: Keep-Alive
Date: Tue, 12 Sep 2017 09:19:28 GMT
ETag: "3e2f-558f9d5587b40"
Keep-Alive: timeout=5, max=100
Server: Apache
```

If the ETag of the URL matches the value given in the `If-None-Match` header, the HTTP status code returned is `304 Not Modified` and no content is returned by the server. The client thus knows that the content did not change on the server and it can use its cached copy.

Ideally, the computing of your ETag should be minimal in order to make it possible to save both CPU usage and network bandwidth. For a file, a simple ETag could be computed from the timestamp when the document has last been modified combined with its size since they are both available at a low cost from the operating system using a single `stat` call (or equivalent). A more robust ETag could be generated from computing the MD5 hash of the data about to be returned. However, this could be way more expensive, especially if the amount of data is large. Be creative.

The application in Example 9.11, "Flask application example with ETag

shows how one can use the `ETag` header to avoid returning the content of a request by checking the headers `If-None-Match` and `If-Match`.

Example 9.11. Flask application example with ETag usage

```python
import unittest

import flask
import werkzeug


application = flask.Flask(__name__)


class NotModified(werkzeug.exceptions.HTTPException):
    code = 304


@application.route("/", methods=['GET'])
def get_index():
    # Since the content here is always the same, we c
    ETAG = "hword"

    if_match = flask.request.headers.get("If-Match")
    if if_match is not None and if_match != ETAG:
        raise NotModified

    if_none_match = flask.request.headers.get("If-Nor
    if if_none_match is not None and if_none_match ==
        raise NotModified

    return flask.Response("hello world",
                          headers={"ETag": "hword"})


class TestApp(unittest.TestCase):
```

```
    def test_get_index(self):
        test_app = application.test_client()
        result = test_app.get()
        self.assertEqual(200, result.status_code)

    def test_get_index_if_match_positive(self):
        test_app = application.test_client()
        result = test_app.get(headers={"If-Match": "h
        self.assertEqual(200, result.status_code)

    def test_get_index_if_match_negative(self):
        test_app = application.test_client()
        result = test_app.get(headers={"If-Match": "i
        self.assertEqual(304, result.status_code)

    def test_get_index_if_none_match_positive(self):
        test_app = application.test_client()
        result = test_app.get(headers={"If-None-Match
        self.assertEqual(304, result.status_code)

    def test_get_index_if_none_match_negative(self):
        test_app = application.test_client()
        result = test_app.get(headers={"If-None-Match
        self.assertEqual(200, result.status_code)


if __name__ == "__main__":
    application.run()
```

Tip

[Example 9.11, "Flask application example with ETag usage"](#)
embeds unit tests as an example, to show how easy it is to test –
you can run them by calling `nosetests examples/flask-etag.py` if you have *nose* installed. If not, you can install it by
using pip and running `pip install nose`.

ETags can also be used for optimistic concurrency control to help prevent simultaneous updates of a resource from overwriting each other. By comparing the ETag received during the first GET request and by passing it in a subsequent PUT, POST, PATCH or DELETE request [6], it is possible to make sure that concurrent operations are not writing on each other.

The application presented in Example 9.12, "Flask application example with ETag usage and PUT" implements two endpoints: GET / which returns an integer stored in the global variable VALUE and POST / which allows this variable to be incremented this variable by 1. The ETag is generated by incrementing a different counter named ETAG, and it is incremented randomly each time the value is changed.

Example 9.12. Flask application example with ETag usage and PUT

```python
import random
import unittest

import flask
from werkzeug import exceptions


app = flask.Flask(__name__)


class NotModified(exceptions.HTTPException):
    code = 304


ETAG = random.randint(1000, 5000)
VALUE = "hello"


def check_etag(exception_class):
    global ETAG

    if_match = flask.request.headers.get("If-Match")
```

```python
        if if_match is not None and if_match != str(ETAG)
            raise exception_class

        if_none_match = flask.request.headers.get("If-Nor
        if if_none_match is not None and if_none_match ==
            raise exception_class


@app.route("/", methods=['GET'])
def get_index():
    check_etag(NotModified)
    return flask.Response(VALUE, headers={"ETag": ET


@app.route("/", methods=['PUT'])
def put_index():
    global ETAG, VALUE

    check_etag(exceptions.PreconditionFailed)

    ETAG += random.randint(3, 9)
    VALUE = flask.request.data
    return flask.Response(VALUE, headers={"ETag": ET


class TestApp(unittest.TestCase):
    def test_put_index_if_match_positive(self):
        test_app = app.test_client()
        resp = test_app.get()
        etag = resp.headers["ETag"]
        new_value = b"foobar"
        result = test_app.put(headers={"If-Match": et
                              data=new_value)
        self.assertEqual(200, result.status_code)
        self.assertEqual(new_value, result.data)

    def test_put_index_if_match_negative(self):
```

```
        test_app = app.test_client()
        result = test_app.put(headers={"If-Match": "v
        self.assertEqual(412, result.status_code)


if __name__ == "__main__":
    app.run()
```

As you can easily spot in the unit tests I have included, it is easy for a client to check that the resource has not been modified before overwriting it: if the data has changed, then the ETag is going to be different than the one included in `If-Matches`, and therefore the request is aborted with a `412 Precondition Failed` status code being returned. The client can then issue a new `GET /` request to retrieve the new content and ETag, redo any computation it wants to do and try again to update it. The code shown in [Example 9.13, "Flask application example with ETag usage and PUT"](#) implements such a retry loop that implements this optimistic concurrency model.

Example 9.13. Flask application example with ETag usage and PUT

```
while True:
    resp = client.get()
    etag = resp.headers["ETag"]

    new_data = do_something(resp.data)

    resp = client.put(data=new_data, headers={"If-Mat
    if resp.status_code == 200:
        break
    elif resp.status_code == 412:
        continue
    else:
        raise RuntimeError("Unknown exception: %d" %
```

Note

As you may have noticed, there is no delay between each retry implemented in the example from [Example 9.13, "Flask application example with ETag usage and PUT"](). Hammering the HTTP endpoint is not a very good practice, as it can overload the HTTP server easily until a client wins the race. Since this case is a (soft) failure scenario, the correct strategies to implement in order to retry correctly on failure are described in [Chapter 6, *Designing for Failure*]().

Using ETag header is a great way to implement basic concurrency control on your HTTP API while also providing hints for clients to cache data as needed. It improves scalability by reducing the network traffic and CPU usage of your application, and moreover it allows consumers to work on the same data without conflicting with each other.

9.4. Asynchronous HTTP API

It is pretty common when writing HTTP API to return a `200 OK` status code to the caller to indicate the request succeeded. While easy and convenient, it is possible that the action triggered by the caller takes a lot of time. If the call requires a long delay to be executed, it blocks the caller as it has to wait for the reply, and this increases the risk of failure.

Indeed, if the connection lasts for too long (let's say, a few seconds) and the network is having issues, the connection can be interrupted. In such a case, the caller has to retry the request. If that problem happens thousands of time with flaky clients, it means that tons of CPU time and network bandwidth are spent for nothing.

An obvious way to avoid those problems is to make lengthy operations asynchronous. This can be done easily by returning a `202 Accepted` HTTP status code and returning little or no content. This status code indicates that the request has been accepted and is being processed by the server. Another asynchronous process can then handle the request and take care of it.

[Example 9.14, "Flask application example using asynchronous job]()

handling" provides an application implementing this mechanism. The provided API allow any client to use its *sum* service: pass number to this service, and it sums them. The client can later request the results, as soon as it is ready.

Example 9.14. Flask application example using asynchronous job handling

```python
import queue   # Queue on Python 2
import threading
import uuid

import flask
from werkzeug import routing


application = flask.Flask(__name__)
JOBS = queue.Queue()
RESULTS = {}


class UUIDConverter(routing.BaseConverter):

    @staticmethod
    def to_python(value):
        try:
            return uuid.UUID(value)
        except ValueError:
            raise routing.ValidationError

    @staticmethod
    def to_url(value):
        return str(value)


application.url_map.converters['uuid'] = UUIDConverte
```

```
@application.route("/sum/<uuid:job>", methods=['GET']
def get_job(job):
    if job not in RESULTS:
        return flask.Response(status=404)
    if RESULTS[job] is None:
        return flask.jsonify({"status": "waiting"})
    return flask.jsonify({"status": "done", "result":


@application.route("/sum", methods=['POST'])
def post_job():
    # Generate a random job identifier
    job_id = uuid.uuid4()
    # Store the job to be executed
    RESULTS[job_id] = None
    JOBS.put((job_id, flask.request.args.getlist('nur
    return flask.Response(
        headers={"Location": flask.url_for("get_job",
        status=202)


def compute_jobs():
    while True:
        job_id, number = JOBS.get()
        RESULTS[job_id] = sum(number)


if __name__ == "__main__":
    t = threading.Thread(target=compute_jobs)
    t.daemon = True
    t.start()
    application.run(debug=True)
```

The application offers two endpoints. A request made to `POST /sum` should contain numbers in the query string. The application stores those numbers in a `queue.Queue` object along with a unique identifier

(UUID). Python provides thread-safe queues as the `queue.Queue` objects. Since this application uses a background (daemon) thread, it needs such a thread-safe data structure. This thread is responsible for computing the sum of the numbers stored as part of the sum job.

The second endpoint allows the client to retrieve the result of its operation by querying the `RESULT` variable with the job id, and returning the result if available.

```
$ http POST http://localhost:5000/sum number==42 numk
HTTP/1.0 202 ACCEPTED
Content-Length: 0
Content-Type: text/html; charset=utf-8
Date: Tue, 12 Sep 2017 14:17:59 GMT
Location: http://localhost:5000/sum/0e02e34b-ee31-42d
Server: Werkzeug/0.11.9 Python/2.7.13
```

The reply contains the URL of the result in the `Location` header. The client then only has to send a `GET` request to that URL to retrieve the result.

```
$ http GET http://localhost:5000/sum/0e02e34b-ee31-42
HTTP/1.0 200 OK
Content-Length: 41
Content-Type: application/json
Date: Tue, 12 Sep 2017 14:17:43 GMT
Server: Werkzeug/0.11.9 Python/2.7.13

{
    "result": 100.0,
    "status": "done"
}
```

As expected, the reply of 100.0 is the sum of the numbers sent via the first call: 42, 23 and 35. If the result is not available, the server would reply with a `{"status": "waiting"}` response.

If the result is not computed by the time the request arrives, the server will reply with a `{"status": "waiting"}`. The client will have to try again. This method is often referred to as *polling* as it forces the client to regularly send requests to retrieve its final result. This is definitely not optimal, so two solutions are possible to improve this:

- Implement streaming as shown in [Section 9.2, "Streaming Data"](). It is possible to allow the client to connect to a special endpoint where the application will push the result as soon as it is available.

- Implement a [webhook](). The server should store a URL sent by the client with the numbers. This URL will then be called by the application with the result included. This model is referred as *push* since it pushes the result to the client as soon as they are ready to the client. However, that requires the client to be able to receive such events by having a Web server of its own running.

By using this design model, it can be ensured that the service can receive a substantial amount of jobs requests while being able to handle those tasks in the background. Obviously, adding numbers together is not very CPU intensive nowadays, but I am sure you can imagine heavier jobs that would profit from this design.

9.5. Fast HTTP Client

It is more than likely that you will have to write a client for your server software, or that at some point your application will have to talk to another HTTP server. The ubiquity of REST API makes their optimization patterns a prerequisite nowadays.

Note

There are a couple of ways to optimize the underlying TCP connections, but since many of them rely on operating system hacking and changing settings, obviously they are not covered in this book.

There are many HTTP clients in Python, but the most widely used and easy to work with is *requests*.

The first optimization to take into account is the use of a persistent connection to the Web server. Persistent connections are a standard since HTTP 1.1 though many applications do not leverage them. This lack of optimization is simple to explain if you know that when using *requests* in its simple mode (e.g. with the `get` function) the connection is closed on return. To avoid that, an application needs to use a `Session` object that allows reusing an already opened connection.

Example 9.15. Using *Session* with *requests*

```
import requests

session = requests.Session()
session.get("http://example.com")
# Connection is re-used
session.get("http://example.com")
```

Each connection is stored in a pool of connections (10 by default), the size of which is also configurable, as shown in Example 9.16, "Configuring pool size with *requests*".

Example 9.16. Configuring pool size with *requests*

```
import requests


session = requests.Session()
adapter = requests.adapters.HTTPAdapter(
    pool_connections=100,
```

```
    pool_maxsize=100)
session.mount('http://', adapter)
response = session.get("http://example.org")
```

Reusing the TCP connection to send out several HTTP requests offers a number of performance advantages:

- Lower CPU and memory usage (fewer connections opened simultaneously).
- Reduced latency in subsequent requests (no TCP handshaking).
- Exceptions can be raised without the penalty of closing the TCP connection.

The HTTP protocol also provides pipelining, which allows sending several requests on the same connection without waiting for the replies to come (think batch). Unfortunately, this is not supported by the *requests* library. However, pipelining requests may not be as fast as sending them in parallel. Indeed, the HTTP 1.1 protocol forces the replies to be sent in the same order as the requests were sent – first-in first-out.

requests also has one major drawback: it is synchronous. Calling `requests.get("http://example.org")` blocks the program until the HTTP server replies completely. Having the application waiting and doing nothing can be a drawback here. It is possible that the program could do something else rather than sitting idle.

A smart application can mitigate this problem by using a pool of threads as discussed in Section 2.3, "Using Futures". It allows parallelizing the HTTP requests in a very rapid way.

Example 9.17. Using *futures* with *requests*

```
from concurrent import futures

import requests
```

```
with futures.ThreadPoolExecutor(max_workers=4) as exe
    futures = [
        executor.submit(
            lambda: requests.get("http://example.org'
        for _ in range(8)
    ]

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

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

This pattern being quite useful, it has been packaged into a library named requests-futures. As you can see in Example 9.18, "Using *requests-futures*", the usage of `Session` objects is made transparent to the developer.

Example 9.18. Using *requests-futures*

```
from requests_futures import sessions

session = sessions.FuturesSession()

futures = [
    session.get("http://example.org")
    for _ in range(8)
]

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

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

By default a worker with two threads is created, but a program can easily customize this value by passing the `max_workers` argument or even its own executor to the `FuturSession` object – for example like this:

```
FuturesSession(executor=ThreadPoolExecutor(max_workers=
```

As explained earlier, *requests* is entirely synchronous. That makes the application being blocked while waiting for the server to reply, slowing down the program. Making HTTP requests in threads is one solution, but threads do have their own overhead and this implies concurrency, which is not something everyone is always glad to see in a program.

Starting with version 3.5, Python offers asynchronicity as its core using *asyncio*. The aiohttp library provides an asynchronous HTTP client built on top of *asyncio*. This library allows sending requests in series but without waiting for the first reply to come back before sending the new one. In contrast to HTTP pipelining, *aiohttp* sends the requests over multiple connections in parallel, avoiding the ordering issue explained earlier.

Example 9.19. Using *aiohttp*

```
import aiohttp
import asyncio


async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return response


loop = asyncio.get_event_loop()

coroutines = [get("http://example.com") for _ in rang

results = loop.run_until_complete(asyncio.gather(*cor
```

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

All those solutions (using sessions, threads, futures or asyncio) offer different approaches to making HTTP clients faster.

[Example 9.20, "Program to compare the performances of different *requests* usage"](#) is an HTTP client sending requests to `httpbin.org`, an HTTP API that provides (among other things) an endpoint simulating a long request (a second here). This example implements all the techniques listed above and times them.

Example 9.20. Program to compare the performances of different *requests* usage

```python
import contextlib
import time

import aiohttp
import asyncio
import requests
from requests_futures import sessions

URL = "http://httpbin.org/delay/1"
TRIES = 10


@contextlib.contextmanager
def report_time(test):
    t0 = time.time()
    yield
    print("Time needed for `%s' called: %.2fs"
          % (test, time.time() - t0))


with report_time("serialized"):
    for i in range(TRIES):
        requests.get(URL)
```

```python
session = requests.Session()
with report_time("Session"):
    for i in range(TRIES):
        session.get(URL)


session = sessions.FuturesSession(max_workers=2)
with report_time("FuturesSession w/ 2 workers"):
    futures = [session.get(URL)
               for i in range(TRIES)]
    for f in futures:
        f.result()


session = sessions.FuturesSession(max_workers=TRIES)
with report_time("FuturesSession w/ max workers"):
    futures = [session.get(URL)
               for i in range(TRIES)]
    for f in futures:
        f.result()


async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            await response.read()

loop = asyncio.get_event_loop()
with report_time("aiohttp"):
    loop.run_until_complete(
        asyncio.gather(*[get(URL)
                         for i in range(TRIES)]))
```

Running this program gives the following output:

Example 9.21. Output of requests-comparison.py

```
Time needed for `serialized' called: 12.12s
Time needed for `Session' called: 11.22s
Time needed for `FuturesSession w/ 2 workers' called:
Time needed for `FuturesSession w/ max workers' calle
Time needed for `aiohttp' called: 1.19s
```

Without any surprise, the slower result comes with the dumb serialized version, since all the requests are made one after another without reusing the connection.

Using a `Session` object and therefore reusing the connection means saving 8% in terms of time, which is already a big and easy win. Minimally, you should always use a session.

If your system and program allow the usage of threads, it is a good call to use them to parallelize the requests. However threads have some overhead, and they are not weight-less. They need to be created, started and then joined, which does not make them much faster.

Unless you are still using old versions of Python, without a doubt using *aiohttp* should be the way to go nowadays if you want to write a fast and asynchronous HTTP client. It is the fastest and the most scalable solution as it can handle hundreds of parallel requests. The alternative, managing hundreds of threads in parallel is probably not a great option.

Another speed optimization that can be efficient is streaming the requests. When making a request, by default the body of the response is downloaded immediately. The `stream` parameter provided by the *requests* library or the `content` attribute for `aiohttp` both provide a way to not load the full content in memory as soon as the request is executed.

Example 9.22. Streaming with *requests*

```
import requests
```

```
# Use `with` to make sure the response stream is clos
# be returned back to the pool.
with requests.get('http://example.org', stream=True)
    print(list(r.iter_content()))
```

Example 9.23. Streaming with *aiohttp*

```
import aiohttp
import asyncio


async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.content.read()

loop = asyncio.get_event_loop()
tasks = [asyncio.ensure_future(get("http://example.co
loop.run_until_complete(asyncio.wait(tasks))
print("Results: %s" % [task.result() for task in task
```

Not loading the full content is extremely important in order to avoid allocating potentially hundred of megabytes of memory for nothing. If your program does not need to access the entire content as a whole but can work on chunks, it is probably better to just use those methods.

9.6. Testing REST API

Writing REST APIs is nice, but writing REST APIs that work is better. That is why you should always write and run tests against those APIs. It can be tedious and feel unrewarding; however, it is always the solid call in the long run.

Running a distributed system requires cooperation between developers, quality control and operations engineers. It also means that all those

involved should be able to make sure that a service is going to be OK. It is important for developers to be able to do rapid prototyping and testing of an idea. It is crucial for quality engineers to be able to validate what is expected from the developed service. It is essential that engineers deploying the service can test that it is deployed and works as expected.

With all of that in mind, the traditional way of writing Python unit and functional tests sounds a bit far from this objective. However, there is a Python tool that is great at solving this problem: it is named *Gabbi*.

Gabbi is an HTTP testing tool. It allows writing testing scenarios in a declarative YAML-based format. This file format is powerful enough to write all the tests you could use and imagine while staying simple enough so that it is easy to write and maintain. Having an uncomplicated way of writing tests is helpful as it lowers the friction and the barrier of entry when it comes to writing tests. Concretely, this means it is less of a burden for the software engineer to write tests and it is simpler for quality engineers to provide new checks.

To run tests, *Gabbi* needs a YAML file per scenario to be run. A scenario is a sequence of HTTP calls, each one described as an entry, such as shown in Example 9.24, "Basic *Gabbi* test file".

Example 9.24. Basic *Gabbi* test file

```
tests:
- name: A test
  GET: /api/resources/id
```

Writing tests is not more complicated than this, and therefore it makes it painless to add more as the application grows.

I have replaced the tests based on `unittest` from Example 9.11, "Flask application example with ETag usage" with a *Gabbi* version in Example 9.25, "Basic *Gabbi* inclusion in a Flask app" and Example 9.26, "*Gabbi* test file for Flask ETag application".

Example 9.25. Basic *Gabbi* inclusion in a Flask app

```python
import os

import flask
from gabbi import driver
import werkzeug


application = flask.Flask(__name__)


class NotModified(werkzeug.exceptions.HTTPException):
    code = 304


@application.route("/", methods=['GET'])
def get_index():
    # Since the content here is always the same, we c
    ETAG = "hword"

    if_match = flask.request.headers.get("If-Match")
    if if_match is not None and if_match != ETAG:
        raise NotModified

    if_none_match = flask.request.headers.get("If-Nor
    if if_none_match is not None and if_none_match ==
        raise NotModified

    return flask.Response("hello world",
                          headers={"ETag": "hword"})


# Run tests using:
# python3 -m unittest -v examples/gabbi/app.py
def load_tests(loader, tests, pattern):
    return driver.build_tests(os.path.dirname(__file_
                              loader, intercept=lambd
```

```
if __name__ == "__main__":
    application.run()
```

Example 9.26. *Gabbi* **test file for Flask ETag application**

```
tests:
  - name: GET root with If-Match match
    GET: /
    request_headers:
      If-Match: hword
    status: 200
    response_headers:
      ETag: hword

  - name: GET root with If-Match no match
    GET: /
    request_headers:
      If-Match: foobar
    status: 304
    response_forbidden_headers:
      - ETag

  - name: GET root with If-None-Match no match
    GET: /
    request_headers:
      If-None-Match: hword
    status: 304
    response_forbidden_headers:
      - ETag

  - name: GET root with If-None-Match match
    GET: /
    request_headers:
      If-None-Match: foobar
    status: 200
    response_headers:
```

```
      ETag: hword
```

The tests can easily be run using the `unittest` module:

```
$ python -m unittest -v app.py
test_request (gabbi.suitemaker.app_basic_get_root_wor
gabbi.suitemaker.app_basic_get_root_works.test_reques
test_request (gabbi.suitemaker.app_basic_get_non-exis
gabbi.suitemaker.app_basic_get_non-existent.test_requ
test_request (gabbi.suitemaker.app_basic_put_root_doe
gabbi.suitemaker.app_basic_put_root_does_not_work.tes
test_request (gabbi.suitemaker.app_etag_get_root_with
gabbi.suitemaker.app_etag_get_root_with_if-match_matc
test_request (gabbi.suitemaker.app_etag_get_root_with
gabbi.suitemaker.app_etag_get_root_with_if-match_no_r
test_request (gabbi.suitemaker.app_etag_get_root_with
gabbi.suitemaker.app_etag_get_root_with_if-none-match
test_request (gabbi.suitemaker.app_etag_get_root_with
gabbi.suitemaker.app_etag_get_root_with_if-none-match

Ran 7 tests in 0.024s

OK
```

Another interesting aspect of *Gabbi* is that it can run the tests from the command line. This portable version allows validating any deployed application – even in production if the tests are written with that aspect in mind.

Example 9.27. Running *Gabbi* using `gabbi-run`

```
$ gabbi-run http://localhost:5000 < etag.yaml
... ✓ gabbi-runner.input_get_root_with_if-match_match
... ✓ gabbi-runner.input_get_root_with_if-match_no_ma
... ✓ gabbi-runner.input_get_root_with_if-none-match_
... ✓ gabbi-runner.input_get_root_with_if-none-match_
```

```
Ran 4 tests in 0.041s

OK
```

Using `gabbi-run` as in [Example 9.27, "Running *Gabbi* using `gabbi-run`"](), it is easy to run the YAML scenario file on a remote server. All it needs to be passed as argument is the root URL where to send the requests and a scenario on its standard input. This tool can be extremely powerful, for example for continuous integration jobs where one wants to do functional testing with a service that is really deployed.

Gabbi provides various other features, such as the use of content from previous request to send out subsequent requests or the usage of *JSONPath* to validate the returned content. Its characteristics make it incredibly powerful to test and validate HTTP REST API in Python.

9.7. Chris Dent on HTTP

Hey Chris! Could you introduce yourself and explain how you came to Python?

I have been working as a sysadmin or developer, somewhere near the Internet, since the early 90s. These days I write and review code in the world of OpenStack. For the ten or so years before joining that community I worked on creating tools and processes to enhance small group collaboration, mostly related to wikis.

I first used Python around 2002 to hack some functionality into the

MoinMoin wiki. At the time the programming language I used the most was Perl, and the meaningful whitespace and other constraints presented by Python put me off.

Five years later, when I finished working with a medium-sized team on a large Perl codebase for an enterprise wiki (Socialtext), I was given the freedom to choose a language when starting my new project (TiddlyWeb).

I had narrowed my choice down to either Ruby or Python. I made the choice to use Python based on a small number of concerns:

- I had Perl fatigue. I wanted as little syntactic sugar as possible (no #, $, @, or {}) and as much helpful constraint as possible. This quickly ruled out Ruby. Ruby and Perl are great, but in my experience Python is better in situations where there are multiple or ephemeral developers.
- I liked WSGI.
- My project was intended as a reference to be replicated in other languages. Python's readability was a big win.

Since then Python has become my language of choice for most projects, personal and professional.

Why do you like WSGI, what do you enjoy in this specification? Does it offer anything more than other protocols around?

I did my first programming for the web with CGI. WSGI built on that simplicity and made it better by providing a very pure functional and composable interface to go along with the structure in `environ`. So when I learned about it, it was familiar but more powerful and more

flexible. I liked that it was based on Python primitives and had no requirement in terms of special libraries or objects.

I had been a long time user of *mod_perl* under Apache and while it worked well I did not like that using it meant I was committed to both *mod_perl* and Apache. When I learned about WSGI, *mod_wsgi* was new and just beginning to become popular. WSGI + *mod_wsgi* provided the same "avoid compilation per request" benefits of mod_perl without all the Apache-specific overhead in mod_perl or mod_python. Initially, I appreciated the clean boundary between server and application that WSGI provided, even though I was mostly using Apache, because it kept concerns separated. Later I liked that my WSGI application could be run using any WSGI-capable server; using a simple one for testing and experimentation and something more configurable and capable (often NGINX+uwsgi) for production.

In some ways, WSGI was a precursor to Rack in Ruby, PSGI in Perl and Jack or JSGI in JavaScript. They all did a similar thing, so apparently the functionality provided by a clean interface was desirable. As time moved on people complained about WSGI, in particular, not being very capable in concurrent or asynchronous environments. This has never been much of a concern for me and is, in fact, a useful constraint: I want the application to be focused, as much as possible, on one request and one response.

What would be a few pieces of advice, a set of principles, for developers who are going to build a WSGI application? What are the pitfalls to avoid?

I do not know that there are any hard and fast rules, but some things

that seem to be true most of the time for me are:

- Don't use a framework if you are starting from scratch. Assemble the best individual pieces to do the work that a framework does.

- Use a library or your tooling that makes an explicit 1:1 association between a URL and a method and the code that handles it. I have used *selector* and *Routes* for this in the past. The important part is that it should be easy to look in one place in the code, given a URL, and easily find where the (one and only one) handler is. This usually means avoiding frameworks that do "object dispatch" or associate URLs with code by way of decorators.

- If you are the sort of person who is inclined to use objects to represent the resources in your WSGI application, avoid coupling persistence and serialization into those objects. Instead, have persistence and serialization interfaces that can accept or return those objects. Doing this makes it a lot easier to add or layer additional implementations, such as caching around a persistence system.

- Related to that: don't get trapped by the idea that the resources represented in URLs are the same as the entities represented in a storage system. This is especially the case when using an RDBMS. What makes sense at the level of HTTP may not be the same at all.

- Figure out a way to effectively test your HTTP requests and responses at the HTTP level. And test first. This will result in a more flexible application that can handle unexpected use cases. I use *wsgi-intercept* and *Gabbi* to do testing.

- If you are using middleware to handle authentication, logging, encoding, exception catching and things like that, make sure that it is easy to compose the middleware and the application easily. This can make testing more straightforward. For example, if you need to fake authentication handling, you can replace the default auth middleware with something that fakes it.

- Use exceptions to handle non-success HTTP response codes and let them rise out of the application to be caught by middleware.

This centralizes error formatting and keeps application code cleanly focused on the happy path.

Like all rules, there are always very valid reasons to break these.

Many services these days are designed as HTTP REST API – as it is stateless, it makes it easier to scale horizontally. Other than this property, what do you think can/should be leveraged in HTTP REST API to achieve better and larger scalability along with improved performances

I hesitate to use the term REST when referring to common HTTP APIs. Unless you are using hypermedia in some form, it is not truly REST, and the debates over REST levels of "lite-ness" are not particularly useful. Given adequate tooling on the client side, hypermedia can be a glorious thing, but more often than not all anyone wants to do is make some HTTP requests to a few known URLs and make some things happen, like changing state on the server.

I think that rather than designing for performance from the outset is best to have some good habits (like keeping statelessness in mind all the time) while building, and design in the capacity to measure the system and change it easily. This happens fairly naturally if you stick to good hygiene while developing: simple functions and classes (if you use them) with single responsibilities, small numbers of collaborators and minimal side effects. Then when you do some profiling the fact that method X is using much time is meaningful because it does not do much. Once you have a clear understanding of where the problems are they can be corrected with accuracy rather than guessing.

HTTP APIs that are read heavy are incredibly amenable to caching. *memcached* is a reliable old friend for this. Consider using caching at multiple layers: when reading from the database and when creating external representations of entities (such as JSON serialization). Cache invalidation can be complex, especially when working with collections of resources. Namespacing can help with that, allowing a large chunk of cache data to be invalidated simply by changing to a different namespace. With namespacing, even APIs which receive many writes can be effectively cached.

Use cache headers to ensure that clients only make the requests they must. If a client can reuse cached data, they need not make any request at all. If cache headers make it possible to validate locally cached data with the server, then no response body needs to be sent over the network. If the server can effectively validate cache headers (such at *ETags*) by validating requests without talking to the data store, work is saved. It is possible to manage an *ETag* cache on the server's side using the namespacing mentioned above.

Try to ensure that any GET request never changes any state on the server (except logs). More generally any code pathway which is doing a read should only read. This simplifies caching concerns and also opens up the possibility for having these read-only code paths use a read-only data store separate from the data store which accepts writes. In fact, it can be entirely reasonable for the read pathways to be completely separate applications from the writing pathway. The use of fast reverse proxies (like *NGINX*) can make this relatively straightforward.

If fast writes are required, and the client does not need 100% reliable confirmation that the data has reached its final resting place, accept

the data into a cache, return a 202 and let some other processes migrate the new data from the cache into the long-term persistent storage. In other words, do what [Gnocchi](#) does.

Finally, if managing writes that are updates to existing resources (instead of always creating new resources), ETags provide a standards-compliant way of avoiding the "[Lost Update Problem](#)" that can integrate with the above caching strategies, minimizing wasted reads of the data store.

Do you have any recommendation on how to test (unit, functional) HTTP API? Moreover, what about benchmarking and profiling?

I think the most important consideration when testing, and HTTP API, is to be sure you are clear about what you are testing in any given set of tests. It is easy, for example, to combine testing the persistence layer with testing the HTTP API layer. This is probably a bad idea. We should be able to assume that the persistence layer is correct (has its own robust tests) when testing the API layer. Similarly testing the API layer should not replace testing the serialization layer: we do not want to have to verify the entire contents of a response body over and over again when testing an HTTP API. Instead, we should validate that the response is what we expect.

So what's important is making sure that the HTTP parts of the API are in fact correct HTTP (good response codes, proper headers). I was frustrated enough with trying to do this with other tools that I made my own, called *Gabbi*, that makes it possible to express a sequence of HTTP test in a YAML-based format that is fairly

readable and expresses an HTTP request and response in a clear and flexible fashion. It can talk to a WSGI application directly or to a running service (doesn't have to be Python).

When using *Gabbi* to test an HTTP API, the result is a suite of files that form a solid introduction to how the API works explicitly from the HTTP interactions, not a client which hides the HTTP. Besides being helpful for maintenance and new contributors, this helps to make sure that the tested API is usable and useful by unplanned clients.

There are many tools for benchmarking. It is important to do, but not very informative without also doing profiling. These days, *wrk* and *siege* appear to be common choices. It is important to test both lots of concurrent connections and well as lots of serial connections. Testing each of these scenarios to failure will reveal different issues.

For profiling, it is important to narrow the scope of the profile data so that the Web server and any middleware do not have an impact. The way I have done this is to use a profiling middleware ([werkzeug has one](#)) as the last middleware, so only the actual WSGI application is being profiled. This makes it possible to make one request, create a profile of just that request, and do the required analysis.

Of course, more often than not the time-honored tradition of outputting `time.time()` at critical sections of the code can be very useful to get a rough overview of where things are slow.

---

[6] Or any other method that could modify an object on the server.

Chapter 10. Deploying on *PaaS*

Deploying applications and managing their processes can be tedious. As we have seen in [Section 9.1, "The WSGI Protocol"](#), you need to setup and configure a WSGI-compliant Web server to serve your code. Managing servers to serve your application can be hazardous and tiresome, especially if it is not your specialty. I am not even mentioning the security concern and the overall maintenance you will have to take care of.

Platform as a service, or *PaaS* for short, is a cloud computing service that provides hosting for your application. It allows managing your deployment without the complexity of building and maintaining the needed infrastructure. Those platforms usually offer interesting features, such a relational database system management and auto-scaling.

They are exciting alternatives with the classical server of virtual-machine deployment, because if you are only a small team of developers with no expertise (yet?) in infrastructure, this can help you save a lot of time.

There are numbers of platforms as a service out there – it would not be practical to cover them all in this book. However, as I think it is interesting to demonstrate what kind of ease they can provide, I will talk about a few of them in the following sections.

Tip

Most of those platforms are not free, but offer a free-tier that allows you to test them.

10.1. Heroku

[*Heroku*](#) is one of the first cloud platforms out there, starting up 2007. It provides a platform which supports several languages, among them Python. It also has delightfully good taste in that it also to provides [PostgreSQL](#) support – a marvelous RDBMS.

Heroku is simple to use for shipping any kind of application in production. They have extensive and complete [documentation online](#) that covers most use cases.

Once you have completed your registration online via their Web site, the `heroku` command line tool allows you push and deploy your application.

Heroku uses *git* to manage the versioning and deployment of applications. In order to deploy an application, a simple *git push* is needed.

Heroku provides a process model, where your application can be made up of one or several processes. The usual default processes is a WSGI application served by a WSGI server such as *gunicorn*. Those processes can be configured in a *Procfile* file, which looks like this:

Example 10.1. `Profile` for Heroku

```
web: gunicorn hello.wsgi --log-file -
```

With this simple file it is possible to run an WSGI application from the `hello.wsgi` module.

Example 10.2. Deploying a Heroku application

```
$ ls -R
Procfile          app.json          hello          re

./hello:
__init__.py wsgi.py
$ heroku login
Enter your Heroku credentials.
Email: python@example.com
Password:
Logged in as python@example.com
$ heroku create
Creating app... done, ● fierce-savannah-40050
https://fierce-savannah-40050.herokuapp.com/ | https
```

```
$ git init
$ git add .
$ git commit -m 'Initial import'
$ git remote add heroku https://git.heroku.com/fierce
$ git push heroku master
Counting objects: 280, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (128/128), done.
Writing objects: 100% (280/280), 43.45 KiB | 0 bytes,
Total 280 (delta 134), reused 274 (delta 133)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-2.7.13
remote:        $ pip install -r requirements.txt
remote:           Collecting gunicorn==19.6.0 (from -r ,
remote:             Downloading gunicorn-19.6.0-py2.py3-
remote:           Installing collected packages: gunicor
remote:           Successfully installed gunicorn-19.6.(
remote:
remote: -----> Discovering process types
remote:           Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:           Done: 36.3M
remote: -----> Launching...
remote:           Released v3
remote:           https://fierce-savannah-40050.herokuap
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/fierce-savannah-40050.git
 * [new branch]      master -> master
$ curl https://fierce-savannah-40050.herokuapp.com/
Hello world!
```

As [Example 10.2, "Deploying a Heroku application"](#) illustrates, it is quite

straightforward to deploy a WSGI application to Heroku's platform. Once the source code is ready and committed, a simple push to Heroku git repository deploys the application: Python is installed, the dependencies listed in `requirements.txt` are likewise installed and the processes listed in the `Procfile` as executed.

If your application needs more processing power, it is then possible to upgrade the amount of *dyno* [7] based on your needs:

```
$ heroku ps:scale web=5
Scaling dynos... done, now running web at 5:Standard-
```

Note

Some features, such as automatically scaling your number of *dyno* based on the usage of your application are in the works as of this writing.

The `Procfile` can specify workers other than a WSGI server, such as a daemon handling a queue or running regular tasks. This can be done by simply editing your `Procfile` and adding such a line:

```
worker: python myherokuapp/myprogram.py
```

This also can be scaled up and be spread on multiple *dynos*. The reference implementation for background jobs when using Python on Heroku is to use *RQ* – good thing *RQ* was discussed in Section 5.1, "RQ"!

Heroku takes full advantage of the technics we discussed so far. Having stateless Web servers and stateless queue workers make it straightforward to scale up your application.

Figure 10.1. Heroku Web dashboard
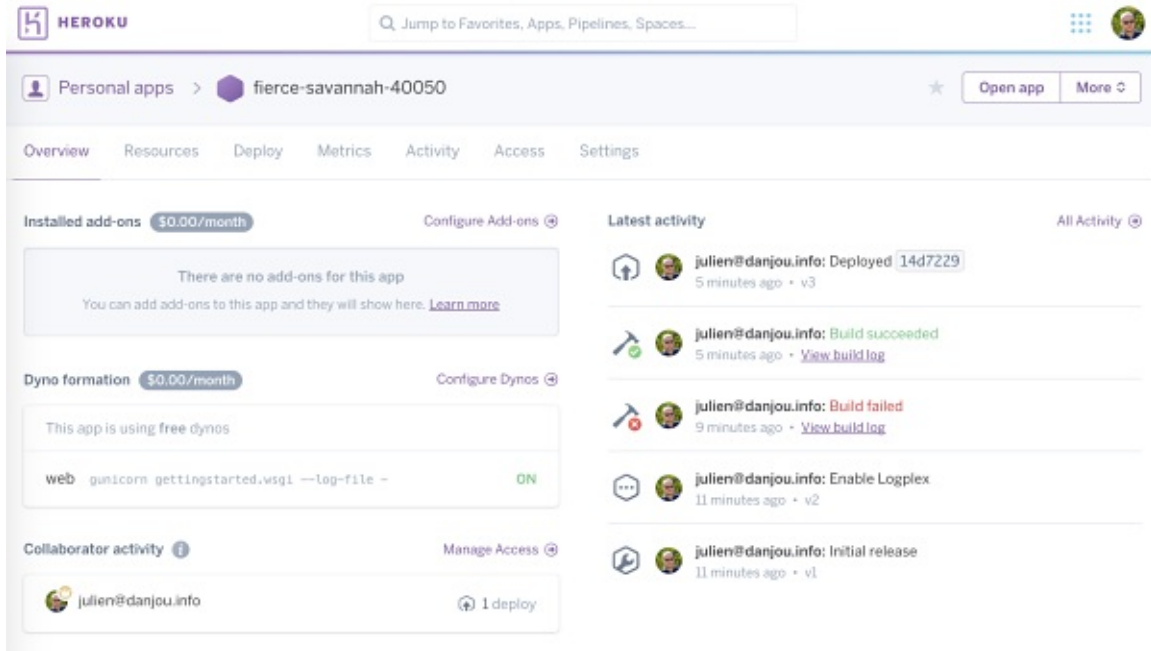
10.2. Amazon Beanstalk

[Amazon Web Services](#) provides a platform to execute code directly, which supports Python, called [AWS Elastic Beanstalk](#). It makes it very easy to run any Python application in the Amazon cloud. Elastic Beanstalk is basically a service that manages virtual machines instances for you (as provided by Amazon EC2 service).

To use this service, Amazon provides a Python command line tool named `eb` that is provided by the *awsebcli* Python package. You can install it using `pip install awsebcli`.

Once your WSGI application is ready to be deployed, you can call `eb init` to initialize the project and then `eb create` to create an environment.

Example 10.3. Deploying to Ealstic Beanstalk with `eb`

```
$ eb init scapytest
$ eb create
Enter Environment Name
```

```
(default is scapytest-dev):
Enter DNS CNAME prefix
(default is scapytest-dev):

Select a load balancer type
1) classic
2) application
(default is 1): 1

Type "view" to see the policy, or just press ENTER to
Creating application version archive "app-4bd9-170119
Uploading scapytest/app-4bd9-170119_180332.zip to S3
Upload Complete.
Environment details for: scapytest-dev
  Application name: scapytest
  Region: us-west-2
  Deployed Version: app-4bd9-170119_180332
  Environment ID: e-x4uchhqxmk
  Platform: 64bit Amazon Linux 2016.09 v2.3.0 running
  Tier: WebServer-Standard
  CNAME: scapytest-dev.us-west-2.elasticbeanstalk.com
  Updated: 2017-01-19 17:03:37.409000+00:00
[…]
INFO: Environment health has transitioned from Pending
INFO: Added instance [i-0a3c269030a13566c] to your en
INFO: Successfully launched environment: scapytest-de
$ curl -v http://scapytest-dev.us-west-2.elasticbeans
*   Trying 54.186.46.232...
* TCP_NODELAY set
* Connected to scapytest-dev.us-west-2.elasticbeansta
> GET / HTTP/1.1
> Host: scapytest-dev.us-west-2.elasticbeanstalk.com
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=UTF-8
```

```
< Date: Thu, 19 Jan 2017 17:16:57 GMT
< Server: Apache/2.4.23 (Amazon) mod_wsgi/3.5 Python,
< Content-Length: 13
< Connection: keep-alive
<
Hello world!
* Curl_http_done: called premature == 0
* Connection #0 to host scapytest-dev.us-west-2.elast
```

As shown in the transcript [Example 10.3, "Deploying to Ealstic Beanstalk with eb"](#), the command line tool `eb` provided by *awsebcli* makes it very easy to deploy a Python application in a few commands. And like all cloud providers, AWS provides autoscaling features based on several different metrics (CPU usage, network bandwidth, etc.).

Since Elastic Beanstalk is based on other Amazon services such as EC2, it can be handy to use and you can alsoleverage other services.

Figure 10.2. AWS Elastic Beanstalk dashboard
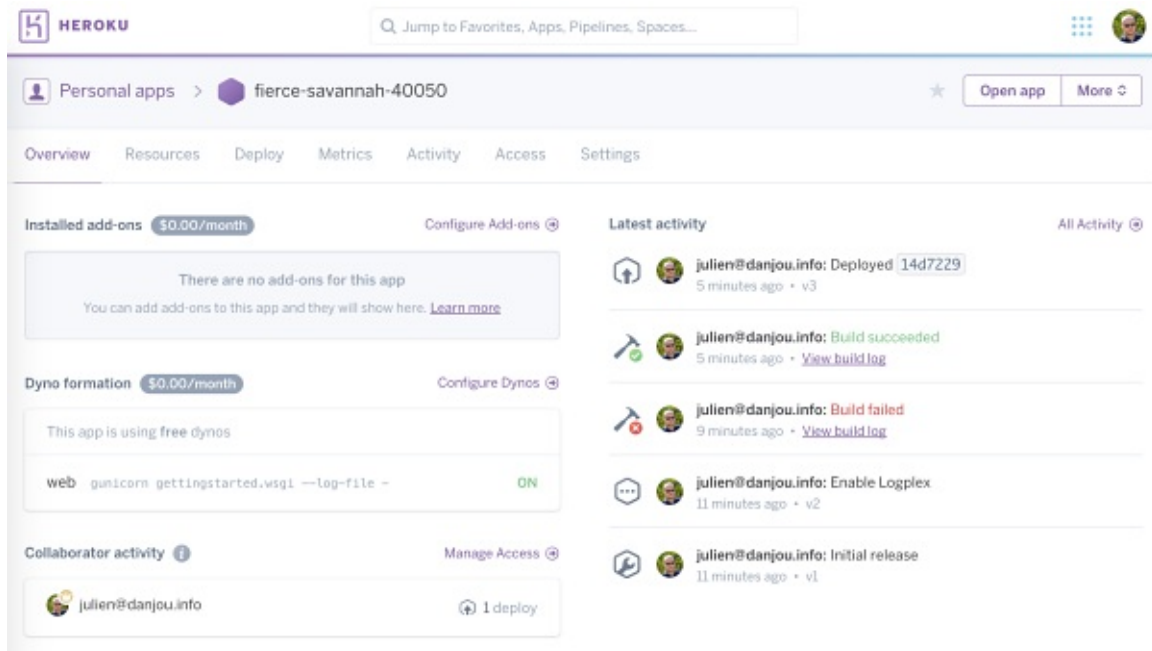
10.3. Google App Engine

[Google](#) also has its cloud platform that offers a large selection of features, from virtual machines to application hosting. [Google App Engine](#) is the platform that can host and execute your code directly, including Python and many other languages.

Google provides a command-line tool, like everyone else, and this one is called `gcloud`. It is rather straightforward to use and the online documentation is pretty complete in terms of how to deploy your WSGI application.

There are only two files needed to deploy a WSGI Python application in the Google App Engine:

- `app.yaml` which contains the application metadata and the address of the main module to run

- At least one Python file that contains the code to run – obviously, it can be an entire Python module with subdirectories.

Example 10.4. `app.yaml` for Google App Engine

```
runtime: python27
api_version: 1
threadsafe: true

handlers:
  - url: /.*
    script: main.app
```

Example 10.5. `main.py` for Google App Engine

```
def app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain'
    start_response(status, response_headers)
    return ['Hello world!\n']
```

Once those files exist, you can directly deploy your WSGI application on Google App Engine and access it as Example 10.6, "Deploying a WSGI application on Google Cloud App" shows.

Example 10.6. Deploying a WSGI application on Google Cloud App

```
$ gcloud app deploy
You are about to deploy the following services:
 - scapytest/default/20170120t170002 (from [/Users/jc
     Deploying to URL: [https://scapytest.appspot.com

Do you want to continue (Y/n)?  y

Beginning deployment of service [default]...
File upload done.
Updating service [default]...done.
Deployed service [default] to [https://scapytest.apps

You can read logs from the command line by running:
  $ gcloud app logs read -s default

To view your application in the web browser run:
  $ gcloud app browse
abydos examples/gae-app ➜ curl https://scapytest.apps
Hello world!
```

Figure 10.3. Google App Engine dashboard
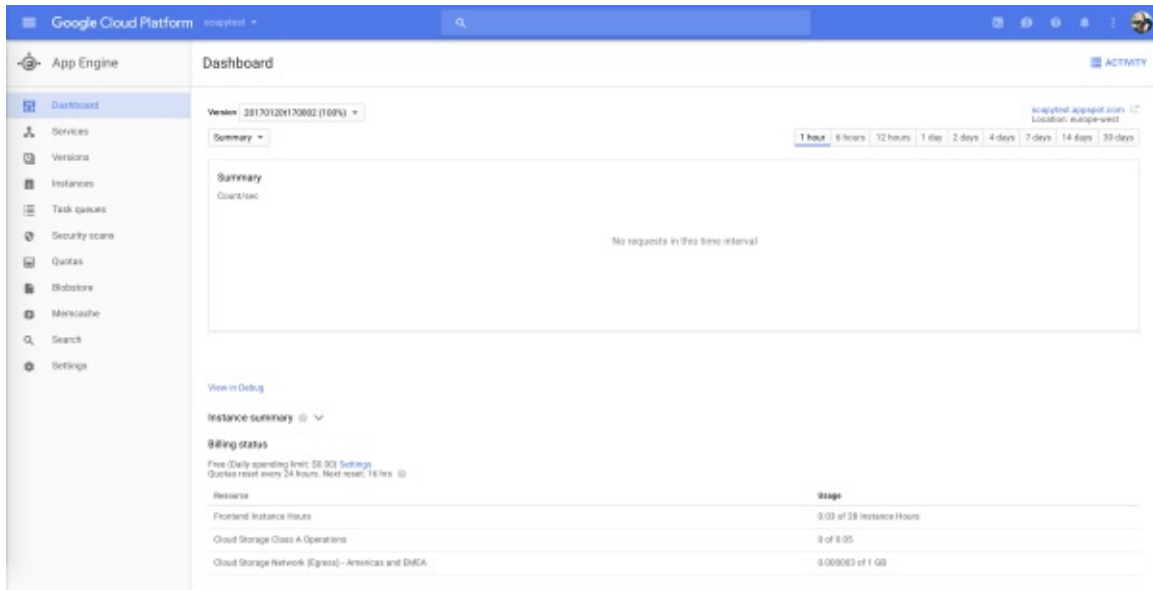
Like many of the others platforms, Google App Engine provides scalability features. In this case, it is automatic and there is nothing that has to be done if the traffic to the application increases.

The main downside of Google App Engine is that it only supports Python 2.7 in its standard environment as of this writing – Python 3 is however available in the flexible environment.

10.4. OpenShift

OpenShift is a platform as a service software edited by Red Hat. It offers support for a number of different programming language, among them Python. It goes beyond what Heroku can offer as it has a high level of customization, being based on Kubernetes and Docker. That means it is easy to support more scenarios than just a Python application as you can provide your own Docker images and Kubernetes templates.

OpenShift is available online through **OpenShift Online** as a public cloud that is operated by Red Hat, through **OpenShift Dedicated** to run and manage your own service on other cloud providers (Amazon, Google Compute Engine, etc.) and through **OpenShift Container Platform** to be hosted on your own hardware and data center. This

allows you to pick the best solution for your projects, as you can internalize or externalize what you want, from nothing to everything.

The ability to run your own PaaS platform can be pretty important as it prevents being in a situation in which you are locked in to a vendor.

It is also pretty straightforward to run **OpenShift Origin**, the free edition of OpenShift, on your own hardware. For example, I was able to deploy OpenShift on my laptop in a few minutes. Once Docker is installed on your machine and OpenShift tools are deployed, creating a new cluster is easy enough:

```
$ oc cluster up
-- Checking OpenShift client ... OK
-- Checking Docker client ... OK
-- Checking Docker version ... OK
-- Checking for existing OpenShift container ... OK
-- Checking for openshift/origin:v1.3.2 image ...
   Pulling image openshift/origin:v1.3.2
   Pulled 0/3 layers, 6% complete
   Pulled 1/3 layers, 70% complete
   Pulled 2/3 layers, 85% complete
   Pulled 3/3 layers, 100% complete
   Extracting
   Image pull complete
-- Checking Docker daemon configuration ... OK
-- Checking for available ports ... OK
-- Checking type of volume mount ...
   Using Docker shared volumes for OpenShift volumes
-- Creating host directories ... OK
-- Finding server IP ...
   Using 192.168.64.3 as the server IP
-- Starting OpenShift container ...
   Creating initial OpenShift configuration
   Starting OpenShift using container 'origin'
   Waiting for API server to start listening
   OpenShift server started
```

```
-- Installing registry ... OK
-- Installing router ... OK
-- Importing image streams ... OK
-- Importing templates ... OK
-- Login to server ... OK
-- Creating initial project "myproject" ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
       https://192.168.64.3:8443

   You are logged in as:
       User:     developer
       Password: developer

   To login as administrator:
       oc login -u system:admin

$ oc login -u system:admin
Logged into "https://192.168.64.3:8443" as "system:ac

You have access to the following projects and can sw:

    default
    kube-system
  * myproject
    openshift
    openshift-infra

Using project "myproject".
```

Using OpenShift to deploy a Python application is straightforward. The Web dashboard offers access to all the features. It only requires providing a git URL to your project repository to run your favorite WSGI application.

```
$ oc new-app python~https://github.com/OpenShiftDemos
--> Found image 09a1531 (40 hours old) in image strea

    Python 3.5
    ----------
    Platform for building and running Python 3.5 appl

    Tags: builder, python, python35, rh-python35

    * A source build using source code from https://g
      * The resulting image will be pushed to image s
      * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment confi
    * Port 8080/tcp will be load balanced by service
      * Other containers can access this service thro

--> Creating resources with label app=scaling-python-
    imagestream "scaling-python-test" created
    buildconfig "scaling-python-test" created
    deploymentconfig "scaling-python-test" created
    service "scaling-python-test" created
--> Success
    Build scheduled, use 'oc logs -f bc/scaling-pytho
    Run 'oc status' to view your app.
$ oc expose svc scaling-python-test
route "scaling-python-test" exposed
$ oc status
In project My Project (myproject) on server https://1

http://scaling-python-test-myproject.192.168.64.3.xip
  dc/scaling-python-test deploys istag/scaling-python
    bc/scaling-python-test source builds https://gith
    deployment #1 deployed 44 minutes ago - 1 pod
$ curl -v http://scaling-python-test-myproject.192.16
*   Trying 192.168.64.3...
* TCP_NODELAY set
* Connected to scaling-python-test-myproject.192.168.
```

```
> GET / HTTP/1.1
> Host: scaling-python-test-myproject.192.168.64.3.x:
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: gunicorn/19.6.0
< Date: Thu, 19 Jan 2017 14:32:16 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 12
< Set-Cookie: 438d0cfdb6e5a6a330e8b3c011196054=1c135(
< Cache-control: private
<
* Curl_http_done: called premature == 0
* Connection #0 to host scaling-python-test-myproject
Hello World!
```

As you can see, it is quite comfortable to deploy a Python application to an OpenShift instance. OpenShift takes care of cloning the git repository and deploying it in a Python container.

There are numerous other features provided by OpenShift that allow for completing your application. It can also help deploying and managing network services (RDBMS, REST API, key/value stores, etc.).

On the scaling side, OpenShift makes it very easy to scale your application to several *pods* automatically based on CPU consumption. It monitors the application health for you, and the API or dashboard allows you to define your scaling rules with a few click or commands.

Figure 10.4. OpenShift Web dashboard
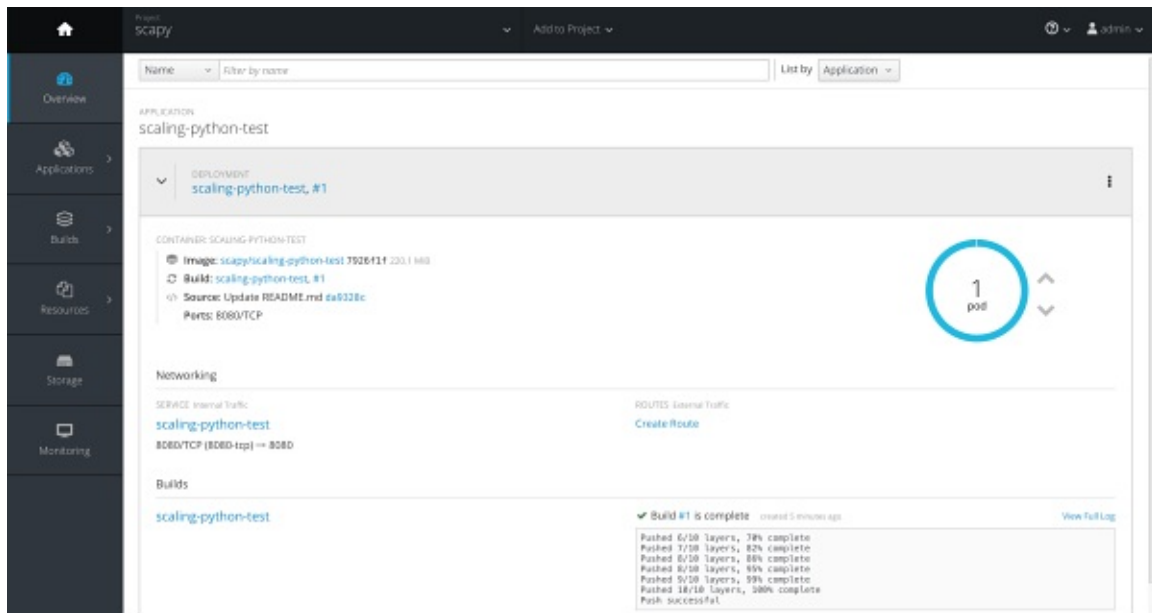
10.5. Beyond PaaS

As we have just seen, there is a long list of hosting platform that you can use and leverage to deploy your application. The above list of cloud platforms is not exhaustive; there is a lot more of them out there. They all have different offerings, while the foundation is usually common: running Python code.

Many of those platforms include more than just running your Python code: database access, message queues, object storage. These might be necessary building blocks of your application. Most of those services are scalable by design, and the performance of those components is the responsibility of the provider – both an advantage and a disadvantage, it is up to you to decide. However, studying those services, what they provide and how they do it, can teach you a thing or two about scalability and the trade-offs you might have to choose.

Very few of those extra services are standardized, which means that your application, which is based on an open source language, might be subject to vendor lock-in. Migrating your Python application from one platform to another could therefore be a daunting task or even impossible: the APIs are different, the services are disparate, etc. In cases of prolonged

service outages or a change of policy, being tied to one particular provider is rarely pleasant. This can be mitigated by using a platform that provides an open API that you could run on your own, such as *OpenShift*.

While Python hosting platforms are usually highly scalable and fault-tolerant, they are not resilient. They can crash or fail, suffer server downtime or have network issues. Your application should be prepared for this eventuality – wherever it runs. Using multiple distribution sites if available or multiple providers might be a solution to think about, depending on the level of service that you need to provide!

---

[7] *dyno* is the name Heroku gives to the containers running your application.

Chapter 11. Testing Distributed Systems

Among the types of tests that are needed to write and maintain proper software, unit tests are not affected by scalability. Whatever your code base size, the way to write and execute those tests stays the same. Such checks are not tied to anything other than simple units of code, and they should not be dependent of the global application architecture.

However, most of the time, this cannot be said of functional tests. They can depend on other services if the application has been built using a micro-services architecture for example. The same goes if an application depends on an external system, storage, relational database, etc.

Writing micro-services is an excellent solution for distributing your application among several nodes. The problem is that testing the multiple parts together might be tricky.

There are entire projects that build their functional and integration testing orchestration around shell scripts, which are run by Travis or Jenkins. While that might work for any application, in the next chapter we will discuss using a more Python-oriented approach and go over some tools that are known to most Pythonistas.

The importance of the functional and integration test tool-chain setup is being fast, repeatable, and consistent. This makes sure that all developers can run (most of) the tests in their own environment while making sure there are no false-negatives in the tests results.

11.1. Setting Up Environments with *tox*

As you might already know, *tox* is a tool that automates building virtual

Python environments. Those environments are isolated and straightforward to rebuild while having support for multiple Python versions at the same time in different environments.

Most of the time, *tox* is used to directly run unit tests of a program, with a file as simple as:

```
[testenv]
deps=nose
commands=nosetests
```

Such a file makes sure that the tests are run in a clean Python environment with the right dependencies installed, as specified by your packaging tools.

Since it is possible to run any command by setting the `commands` parameter, it is easy to envision a system where you could start other services before running your tests. Therefore, a simplistic technique to run integration tests with *memcached*, for example, would be:

```
[testenv]
commands=memcached -p 12345 &
        nosetests
```

The application would then have to set itself up to use *memcached* on TCP port 12345, and the tests could run using that instance of *memcached*.

Unfortunately, this approach is overly naive: first, there's no way to know when *memcached* is ready to accept connections. That could make the first tests fail for no good reason. This is exactly the kind of false negative result that you want to avoid at all costs, as they undermine the usefulness of the test suite.

Moreover, after the tests are done, *memcached* would still be running. The second run of *tox* will spawn a new *memcached* instance, which

would fail to start because the TCP port is already being used. Also, the second test run would connect to the previous *memcached* daemon, possibly using old entry and keys stored in it – again yielding potential false negatives.

This short example demonstrates that unfortunately, the orchestration of external dependency for integration testing is much more complicated than one might hope. It does not mean that it is impossible to orchestrate a whole deployment of resources before running your tests, but to do that you need to take care of the following before running the tests:

1. Prepare any directory or resources needed by external services

2. Launch those external services

3. Make sure those external services are ready to be used

Then, after the tests are run you need to:

1. Shut down the external service (correctly, if possible)

2. Clean everything the service might have left behind (e.g. temporary data)

Taking care of this is not trivial, but once done well, it makes sure that you can comfortably run *tox* and run integration testing once and for all with a simple command line `tox -e py36-integration`:

```
[testenv]
deps = nose
commands=nosetests

[testenv:py36-integration]
commands={toxinidir}/setup-memcached.sh nosetests
```

Example 11.1. Tests using *memcached* if available

```
import os
import socket
```

```
import unittest


class TestWithMemcached(unittest.TestCase):
    def setUp(self):
        super(TestWithMemcached, self).setUp()
        if not os.getenv("MEMCACHED_PID"):
            self.skipTest("Memcached is not running")

    def test_connect(self):
        s = socket.socket()
        s.connect(("localhost", 4526))
```

Example 11.2. `setup-memcached.sh`

```sh
#!/bin/sh

clean_on_exit () {
    test -n "$MEMCACHED_DIR" && rm -rf "$MEMCACHED_DI
}

trap clean_on_exit EXIT

wait_for_line () {
    while read line
    do
        echo "$line" | grep -q "$1" && break
    done < "$2"
    # Read the fifo for ever otherwise process would
    cat "$2" >/dev/null &
}


MEMCACHED_DIR=`mktemp -d`
mkfifo ${MEMCACHED_DIR}/out
memcached -p 4526 -vv > ${MEMCACHED_DIR}/out 2>&1 &
export MEMCACHED_PID=$!
```

```
wait_for_line "server listening" ${MEMCACHED_DIR}/out

$*

kill ${MEMCACHED_PID}
```

When run normally, the test is skipped because *memcached* is not detected:

```
$ tox -e py36
GLOB sdist-make: /Users/jd/Source/scaling-python/exam
py36 create: /Users/jd/Source/scaling-python/examples
py36 installdeps: nose
py36 inst: /Users/jd/Source/scaling-python/examples/t
py36 installed: nose==1.3.7,testmemcached==0.0.1.dev2
py36 runtests: PYTHONHASHSEED='3256577142'
py36 runtests: commands[0] | nosetests
S

Ran 1 test in 0.004s

OK (SKIP=1)
  py36: commands succeeded
  congratulations :)
```

But when run with the `py36-integration` target, *memcached* is set-up, and therefore the test can be executed:

```
tox -e py36-integration
GLOB sdist-make: /Users/jd/Source/scaling-python/exam
py36-integration create: /Users/jd/Source/scaling-pyt
py36-integration installdeps: nose
py36-integration inst: /Users/jd/Source/scaling-pytho
py36-integration installed: nose==1.3.7,testmemcached
py36-integration runtests: PYTHONHASHSEED='201362488!
py36-integration runtests: commands[0] | /Users/jd/Sc
```

```
.

Ran 1 test in 0.006s

OK


  py36-integration: commands succeeded
  congratulations :)
```

This is a good strategy for making sure that your tests can run everywhere smoothly, and that every engineer working on the project can run them with their own system. Having concentrated minutes or hours to spend on setting-up a test environment is not a genuine way to scale your application testing beyond just doing unit testing.

11.2. Manage External Services with *pifpaf*

If writing shell scripts is not your favorite pastime, rest assured you are normal. The error handling in this kind of script is often wonky, the verbosity painful, and the tools available to help to build your workflow are very restricted. So while shell scripts are handy to use and write, they are a pain to maintain. They can become way too convoluted to handle.

Therefore this chapter will include some shameless promotion of a tool I started to write a few months ago and that I use in many of my applications nowadays to facilitate integration testing. It is called *pifpaf*.

The problem that *pifpaf* solves is precisely the one covered in Section 11.1, "Setting Up Environments with *tox*": how to start and stop systems needed to test integration with your applications.

pifpaf is a command line tool that allows starting any daemon from the command line, without having to set it up on your system, nor be an administrator. As long as the software is already installed, *pifpaf* is able

to start it. For example, to start *PostgreSQL*, you can just run `pifpaf` `run postgresql` as shown in [Example 11.3, "*pifpaf* launching](#) [*PostgreSQL*"](#).

Example 11.3. *pifpaf* launching *PostgreSQL*

```
$ pifpaf run postgresql $SHELL
$ echo $PIFPAF_URL
postgresql://localhost/postgres?host=/var/folders/7k,
$ psql
Expanded display is used automatically.
Line style is unicode.
SET
psql (9.6.1)
Type "help" for help.

postgres=# \l
                              List of databases
    Name     │ Owner │ Encoding │   Collate    │   Ctyp
─────────────┼───────┼──────────┼──────────────┼──────────
 postgres    │ jd    │   UTF8   │ en_US.UTF-8  │ en_US.U
 template0   │ jd    │   UTF8   │ en_US.UTF-8  │ en_US.U
             │       │          │              │
 template1   │ jd    │   UTF8   │ en_US.UTF-8  │ en_US.U
             │       │          │              │
(3 rows)

postgres=# create table foobar ();
CREATE TABLE
postgres=# \d foobar
    Table "public.foobar"
 Column │ Type │ Modifiers
────────┼──────┼───────────


postgres=# \q
$ exit
$
```

By default, *pifpaf* runs whatever command is given after the `run <daemon>` argument. In Example 11.3, "*pifpaf* launching *PostgreSQL*", the argument is `$SHELL` which is replaced by the shell running (e.g. /bin/bash). This new shell has a few environment variables exported, such as `$PIFPAF_URL` which contains the address of the PostgreSQL server. This URL format is well understood by for example *SQLALchemy*, one of the standard SQL manipulation libraries in Python. In the case of PostgreSQL, *pifpaf* also sets the expected variables for *psql* to work out-of-the-box. Once the application is exited – in this case by typing exit in the subshell – *pifpaf* takes care of stopping PostgreSQL and deleting the temporary data that has been created.

It is then therefore easy to imagine starting a test suite rather than a subshell, and taking advantage of what *pifpaf* sets up for us. Rewriting the examples from Section 11.1, "Setting Up Environments with *tox*" using *pifpaf* is as straightforward as replacing the custom shell script with *pifpaf*:

```
[testenv]
deps = nose
       pifpaf
commands=nosetests


[testenv:py36-integration]
commands=pifpaf run memcached --port 19842 -- nosetes
```

Example 11.4. Tests using *memcached* if available with *pifpaf*

```
import os
import socket
import unittest


class TestWithMemcached(unittest.TestCase):
    def setUp(self):
        super(TestWithMemcached, self).setUp()
```

```
        if not os.getenv("PIFPAF_MEMCACHED_URL"):
            self.skipTest("Memcached is not running")

    def test_connect(self):
        s = socket.socket()
        s.connect(("localhost", int(os.getenv("PIFPAF
```

By using variables exported by *pifpaf*, it is easy to dynamically detect whether *memcached* is available for integration testing and then connect to it in order to execute some tests.

Tip

pifpaf offers support for a large number of daemons: *PostgreSQL, MySQL, memcached, etcd, Redis, Ceph, RabbitMQ, Consul, CouchDB, MongoDB, Gnocchi*, etc. It is written in Python and is easily extensible to support your own applications. Try it!

It is also possible to combine several daemons by using the `--` command separator and specifying several *pifpaf* commands:

```
$ pifpaf run redis -- pifpaf -e PIFPAF2 run memcached
$ env | grep PIFPAF_
PIFPAF_REDIS_URL=redis://localhost:6379
PIFPAF_PID=68205
PIFPAF_DAEMON=redis
PIFPAF_URLS=redis://localhost:6379;memcached://localh
PIFPAF_DATA=/var/folders/7k/pwdhb_mj2cv4zyr0kyrlzjx40
PIFPAF_REDIS_PORT=6379
PIFPAF_URL=redis://localhost:6379
$ env | grep PIFPAF2_
PIFPAF2_MEMCACHED_PORT=11212
PIFPAF2_DAEMON=memcached
PIFPAF2_MEMCACHED_URL=memcached://localhost:11212
PIFPAF2_PID=68207
PIFPAF2_URL=memcached://localhost:11212
PIFPAF2_DATA=/var/folders/7k/pwdhb_mj2cv4zyr0kyrlzjx4
```

```
$ ps
  PID TTY           TIME CMD
68206 ttys001    0:00.05 redis-server *:6379
68208 ttys001    0:00.04 memcached -vv -p 11212
```

There are two servers started by *pifpaf*, with their own set of URLs. The
URLs are also available, separated by a `;` in the global `PIFPAF_URLS`
variable. It is also possible to start several instances of the same daemon
by using different listening ports or sockets.

pifpaf is also able to be used in any shell script by using the same variable
export system used by *ssh-agent*, for example. Calling *pifpaf* without a
command to execute makes it behave in this way:

```
$ pifpaf run memcached
export PIFPAF_DATA="/var/folders/7k/pwdhb_mj2cv4zyr0
export PIFPAF_MEMCACHED_PORT="11212";
export PIFPAF_URL="memcached://localhost:11212";
export PIFPAF_PID=68318;
export PIFPAF_DAEMON="memcached";
export PIFPAF_MEMCACHED_URL="memcached://localhost:1
export PIFPAF_URLS="memcached://localhost:11212";
pifpaf_stop () { if test -z "$PIFPAF_PID"; then echo
$ kill 68318
```

This should indeed by called with an `eval` command if it is to be really
useful:

```
$ eval `pifpaf run memcached`
$ echo $PIFPAF_URL
memcached://localhost:11212
$ pifpaf_stop
```

This way of running *pifpaf* has the advantage of not starting a new
program or a subshell. In the context of shell scripts, for example, this
might be handy.

11.3. Using Fixtures with *pifpaf*

While it is possible to run *pifpaf* globally around a test suite, it is also possible to use it inside tests and orchestrate some of its behavior.

pifpaf exports its drivers as test fixtures. In unit testing, fixtures represent components that are set up before a test and cleaned up after the test is finished. It is usually a good idea to build a specific kind of component for them, as they are reused in a lot of different places. In this case, *pifpaf* exports objects that represent the daemon launched. The object is initialized before each test and reset to its default values when the test is completed.

In Example 11.5, "Tests using *memcached* fixtures", the test sets up a *memcached* instance before running each test. While this is slower and more expensive than using a *memcached* for all the tests, it makes sure that each test runs on top of a clean and fresh *memcached* instance, avoiding side effect that could result from having several tests running one after another the same instance.

Example 11.5. Tests using *memcached* fixtures

```python
import socket

import fixtures
from pifpaf.drivers import memcached


class TestWithMemcached(fixtures.TestWithFixtures):
    def setUp(self):
        super(TestWithMemcached, self).setUp()
        self.memcached = self.useFixture(memcached.Me

    def test_connect(self):
        s = socket.socket()
        s.connect(("localhost", self.memcached.port))
```

This test does not need the *pifpaf* daemon to run and it does not need any setup to be done by *tox*, in contrast to what we have seen previously in [Section 11.2, "Manage External Services with *pifpaf*"](#).

Note

The approach used in [Example 11.5, "Tests using *memcached* fixtures"](#) only works if the tests are run serially, as *memcached* uses the same port for listening for each test. Running tests concurrently would fail, as several *memcached* instances would try to bind to the same TCP/UDP ports. Using an incremental port number for each test or tracking ports used would solve this issue.

While this test is great and allows testing the case in which everything works like a charm, it is even more interesting to test when things could go wrong. *pifpaf* allows us to manipulate its fixture object and do just that.

For example, an application could test what happens when the connection to *memcached* is broken.

Example 11.6. Application using *memcached* and fixtures

```
import fixtures
from pifpaf.drivers import memcached
from pymemcache import client


class AppException(Exception):
    pass


class Application(object):
    def __init__(self, memcached=("localhost", 11211)
        self.memcache = client.Client(memcached)

    def store_settings(self, settings):
```

```
        self.memcache.set("appsettings", settings)

    def retrieve_settings(self):
        return self.memcache.get("appsettings")


class TestWithMemcached(fixtures.TestWithFixtures):
    def test_store_and_retrieve_settings(self):
        self.memcached = self.useFixture(memcached.Me
        self.app = Application(("localhost", self.men
        self.app.store_settings(b"foobar")
        self.assertEqual(b"foobar", self.app.retrieve
```

In Example 11.6, "Application using *memcached* and fixtures", an
imaginary application uses *memcached* to store and retrieve its settings.
The canonical test case is `test_store_and_retrieve_settings`,
which starts *memcached* and tests whether everything works correctly.
This is the case that most people write and end up being happy with.

However, if *memcached* is stopped at store or retrieve time, or it is
restarted, there is no way to know what happens. The application does
not use any `try/except` around its interaction with *memcached*. If
anything wrong happens, the user of the application will probably get an
exception from *pymemcache*. It is unclear until it is tested.

So let's do that and test it.

**Example 11.7. Application using *memcached* and fixtures, testing all
scenarios**

```
import fixtures
from pifpaf.drivers import memcached
from pymemcache import client
from pymemcache import exceptions


class AppException(Exception):
```

```python
        pass


class Application(object):
    def __init__(self, memcached=("localhost", 11211)
        self.memcache = client.Client(memcached)

    def store_settings(self, settings):
        try:
            self.memcache.set("appsettings", settings
        except (exceptions.MemcacheError,
                ConnectionRefusedError,
                ConnectionResetError):
            raise AppException

    def retrieve_settings(self):
        try:
            return self.memcache.get("appsettings")
        except (exceptions.MemcacheError,
                ConnectionRefusedError,
                ConnectionResetError):
            raise AppException


class TestWithMemcached(fixtures.TestWithFixtures):
    def test_store_and_retrieve_settings(self):
        self.memcached = self.useFixture(memcached.Me
        self.app = Application(("localhost", self.mem
        self.app.store_settings(b"foobar")
        self.assertEqual(b"foobar", self.app.retrieve

    def test_connect_fail_on_store(self):
        self.app = Application(("localhost", 123))
        self.assertRaises(AppException,
                          self.app.store_settings,
                          b"foobar")
```

```
    def test_connect_fail_on_retrieve(self):
        self.memcached = memcached.MemcachedDriver(po
        self.memcached.setUp()
        self.app = Application(("localhost", self.men
        self.app.store_settings(b"foobar")
        self.memcached.cleanUp()
        self.assertRaises(AppException,
                          self.app.retrieve_settings)

    def test_memcached_restarted(self):
        self.memcached = memcached.MemcachedDriver(po
        self.memcached.setUp()
        self.app = Application(("localhost", self.men
        self.app.store_settings(b"foobar")
        self.memcached.reset()
        self.addCleanup(self.memcached.cleanUp)
        self.assertRaises(AppException,
                          self.app.retrieve_settings)
```

In Example 11.7, "Application using *memcached* and fixtures, testing all scenarios", this time all the scenarios are tested. The test `test_connect_fail_on_store` tests what happens if *memcached* is not started at all when the application is being used. The test `test_connect_fail_on_retrieve` tests what happens if *memcached* works at the beginning but suddenly is shut down and stops working. Finally, the test `test_memcached_restarted` tests what happens if *memcached* is restarted.

Writing all those new tests and scenarios forced the application to handle new exceptions, such as `ConnectionRefusedError`, whereas it did not do that before. This example raises an `AppException` exception, but an attractive alternative would be to retry using *tenacity*, as described in Section 6.2, "Retrying with *Tenacity*".

Even if *pifpaf* might not suit your needs or be the best tool, the approach described in the last sections should still be kept in mind when writing tests for applications that leverage another component.

Remember: never forget to cover failure scenarios.

Chapter 12. Caching

Caching is an essential component when scaling applications to large proportions. It can solve various problems:

- High cost of computing data: caching allows reusing already-computed results, so rather than computing the same result over and over again, a database of results can be queried rather than compute the whole result again. This is the basic principle behind *memoization*.

- High-latency access to data: some data might be accessible but the latency to access it is too high. If the data is retrieved enough times, storing it in a closer (in terms of latency) data store will improve performance. This introduces the well-known problem of *cache invalidity*.

Caching represents a good solution to those problems and the different techniques described next can be combined to bring about very high performances for your applications.

12.1. Local Caching

Local caching has the advantage of being (very) fast, as it does not require access to a remote cache over the network. Usually, caching works by storing cached data under a key that identifies it. This technique makes Python dictionary the most obvious data structure for implementing a caching mechanism.

Example 12.1. A basic caching example

```
>>> cache = {}
>>> cache['key'] = 'value'
>>> cache = {}
>>> def compute_length_or_read_in_cache(s):
...     try:
...             return cache[s]
```

```
...       except KeyError:
...           cache[s] = len(s)
...           return cache[s]
...
>>> compute_length_or_read_in_cache("foobar")
6
>>> cache
{'foobar': 6}
>>> compute_length_or_read_in_cache("foobar")
6
>>> compute_length_or_read_in_cache("babaz")
5
>>> cache
{'foobar': 6, 'babaz': 5}
```

Obviously, such a simple cache has a few drawbacks. First, its size is unbound, which means it can grow to a substantial size that can fill up the entire system memory. That would result in the death of either the process, or even the whole operating system in a worst-case scenario.

Therefore, a policy must be implemented to expire some items out of any cache, in order to be sure that the data store does not grow out of control. There are a few algorithms that can be found that are pretty simple to implement such as the following:

- Least recently used (LRU) removes the least recently used items first. This means the last access time for each item must also be stored.

- Least Frequently Used (LFU), which removes the least frequently used items first. This means the number of accesses of each item must be stored.

- Time-to-live (TTL) based removes any entry that is older than a certain period of time. This has the benefit of automatically invalidating the cache after a certain amount of time, whereas the LRU and LFU policies are only access-based.

Methods like LRU and LFU makes more sense for memoization (see

Section 12.2, "Memoization"). Other methods, such as TTL, are more commonly used for a locally stored copy of remote data.

The *cachetools* Python package provides implementations of all those algorithms and it is pretty easy to use as shown in Example 12.2, "Using cachetools". There should be no need to implement this kind of cache yourself.

Example 12.2. Using `cachetools`

```
>>> import cachetools
>>> cache = cachetools.LRUCache(maxsize=3)
>>> cache['foo'] = 1
>>> cache['bar'] = 42
>>> cache
LRUCache([('foo', 1), ('bar', 42)], maxsize=3, currsi
>>> cache['bar']
42
>>> cache['foo']
1
>>> cache[12]
Traceback (most recent call last):
[...]
KeyError: 12
>>> cache['baz'] = 33
>>> cache['babar'] = 32
>>> cache
LRUCache([('baz', 33), ('foo', 1), ('babar', 32)], ma
```

The `cachetools.LRUCache` class provides an implementation of a LRU cache mechanism. The maximum size is set to three in this example, so as soon as a fourth item is added to the cache, the least recently used one is discarded.

Example 12.3. Using *cachetools* to cache Web pages

```
import time
```

```
import cachetools
import requests


cache = cachetools.TTLCache(maxsize=5, ttl=5)
URL = "http://httpbin.org/uuid"
while True:
    try:
        print(cache[URL])
    except KeyError:
        print("Paged not cached, fetching")
        cache[URL] = page = requests.get("http://http
        print(page)
    time.sleep(1)
```

In Example 12.3, "Using *cachetools* to cache Web pages", a demo program uses *cachetools* to cache a Web page for five seconds, wherein up to five pages are cached for five seconds. When it run, this program prints the page every second but it only refreshes it every five seconds.

The `TTLCache` class accepts a different definition of time; if needed, you can customize it to not count the time in seconds but in any other time unit that you would like (iteration, pings, requests, etc.).

12.2. Memoization

Memoization is a technique used to speed up function calls by caching their results. The results can be cached only if the function is pure – meaning that it has no side effects or outputs and that it does not depend on any global state.

A trivial function that can be memoized is the sine function `sin`.

Example 12.4. A basic memoization technique

```
>>> import math
>>> _SIN_MEMOIZED_VALUES = {}
>>> def memoized_sin(x):
...     if x not in _SIN_MEMOIZED_VALUES:
...         _SIN_MEMOIZED_VALUES[x] = math.sin(x)
...     return _SIN_MEMOIZED_VALUES[x]
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965}
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin(2)
0.9092974268256817
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
```

The first time that `memoized_sin` is called with an argument that is not stored in `_SIN_MEMOIZED_VALUES`, the value will be computed and stored in this dictionary. Later on, if we call the function with the same value again, the result will be retrieved from the dictionary rather than being computed another time. While `sin` is a function that computes very quickly, this may not be true for some advanced functions involving more complicated computations.

Usage of memoization can be simplified in Python by using a decorator. PyPI lists a few implementations of memoization through decorators, from very simple cases to the most complex and comprehensive.

Starting with Python 3.3, the *functools* module provides a LRU (least recently used) cache decorator. This decorator provides the same functionality as the memoization described here, but with the benefit that

it limits the number of entries in the cache, removing the least recently used one when the cache size reaches its maximum.

The module also provides statistics on cache hits, misses, etc. In my opinion, these are a must-haves when implementing such a cache. There's no point in using memoization – or any caching technique – if you are unable to meter its usage and usefulness.

In Example 12.5, "Using `functools.lru_cache`", you can see an example of the example `memoized_sin` function, using `functools.lru_cache`.

Example 12.5. Using `functools.lru_cache`

```
>>> import functools
>>> import math
>>> @functools.lru_cache(maxsize=2)
... def memoized_sin(x):
...     return math.sin(x)
...
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=2, maxsize=2, currsize=2)
>>> memoized_sin(4)
-0.7568024953079282
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=3, maxsize=2, currsize=2)
>>> memoized_sin(3)
```

```
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=2, misses=3, maxsize=2, currsize=2)
>>> memoized_sin.cache_clear()
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

If an older version of Python is used, or if a different algorithm is desired, the *cachetools* package as seen previously provides a useful `cachetools` module that can be imported with a wide variety of cache types as shown in Example 12.6, "Using `cachetools` for memoization".

Example 12.6. Using `cachetools` for memoization

```
>>> import cachetools.func
>>> import math
>>> import time
>>> memoized_sin = cachetools.func.ttl_cache(ttl=5)(r
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=128, currsize=0)
>>> time.sleep(5)
>>> memoized_sin.cache_info()
>>> CacheInfo(hits=1, misses=1, maxsize=128, currsize
```

12.3. Distributed Caching

Caching systems such as those provided by *cachetools* or `functools.lru_cache` (discussed in Section 12.2, "Memoization") present a big flaw in relation to distributed system: their data store is not distributed. As those functions usually save data into a Python dictionary,

they do not offer a scalable and shared cache data store which is needed for large applications.

When a system is distributed across a network, it also needs a cache that is distributed across a network. Nowadays, there are plenty of network servers that offer caching capability, such as *memcached*, *Redis* and many others.

The simplest one to use is probably *memcached*, which once installed, can simply be launched by calling the `memcached` command. My preferred Python library for interacting with *memcached* is *pymemcache*. I recommend using it. The example in Example 12.7, "Connecting to *memcached*" shows how you can connect to *memcached* and use it as a network-distributed cache across your applications.

Example 12.7. Connecting to *memcached*

```python
from pymemcache.client import base

# Don't forget to run `memcached' before running
client = base.Client(('localhost', 11211))
client.set('some_key', 'some_value')
result = client.get('some_key')
print(result)  # some_value
```

While straightforward enough, this example allows storing key/value tuples across the network and accessing them through multiple, distributed, nodes. This is simplistic yet powerful and it is a first step that might be a cheap enough means to optimize your application.

When storing data into *memcached*, it is possible to set an expiration time – a maximum number of seconds for *memcached* to keep the key and value around. After that delay, *memcached* removes the key from its cache. There is no magic number for this delay, and it will entirely depend on the type of data and application that you are working with. It could be a few seconds, or it might be a few hours.

Cache invalidation, which defines when to remove the cache because it is

out of sync with the current data, is also something that your application will have to handle if presenting data that is too old is to be avoided. Here again, there is no magical recipe; it depends on the type of application you work on.

However, there are several outlying cases that should be handled – which are not handled in Example 12.7, "Connecting to *memcached*".

First, a caching server cannot grow infinitely. This comes up whenever it might be holding too many keys and needs to flush some out. Some keys might also be expired because they reach their time-to-live delay. In those cases, the data is lost, and the canonical source of the data must be queried again. This can be handled simply as done in Example 12.8, "Handling missing keys in *memcached*".

Example 12.8. Handling missing keys in *memcached*

```
from pymemcache.client import base


def do_some_query():
    # Replace with actual querying code to a database
    return 42


# Don't forget to run `memcached' before running
client = base.Client(('localhost', 11211))
result = client.get('some_key')
if result is None:
    # The cache is empty, need to get the value from
    result = do_some_query()
    # Cache the result for next time
    client.set('some_key', result)
print(result)
```

Handling missing keys is mandatory because of normal flush-out operations. It is also obligatory to handle the cold cache scenario, i.e.

when *memcached* has just been started. In that case, the cache will be entirely empty and the cache needs to be fully repopulated, one request at a time.

Some of the cold cache scenarios cannot be prevented, for example a *memcached* crash, but some can, for example migrating to a new *memcached* server. When it is possible to predict that a cold cache scenario will happen, it is better to avoid it. A cache that needs to be refilled means that all of the sudden, the canonical storage of the cached data will be massively hit by all cache users who lack a cache data (also known as a cache miss).

pymemcache provides a class named `FallbackClient` that helps in implementing this scenario as demonstrated in [Example 12.9, "Fallback with *pymemcache*"](#).

Example 12.9. Fallback with *pymemcache*

```
from pymemcache.client import base
from pymemcache import fallback


def do_some_query():
    # Replace with actual querying code to a database
    return 42


# Set `ignore_exc=True` so it is possible to shut dow
# removing its usage from the program, if ever necess
old_cache = base.Client(('localhost', 11211), ignore_
new_cache = base.Client(('localhost', 11212))

client = fallback.FallbackClient((new_cache, old_cach

result = client.get('some_key')
if result is None:
    # The cache is empty, need to get the value from
    result = do_some_query()
```

```
        # Cache the result for next time
        client.set('some_key', result)
    print(result)
```

The `FallbackClient` queries the old cache passed to its constructor, respecting the order. In this case, the new cache server will always be queried first, and in case of a cache miss, the old one will be queried – avoiding a possible return-trip to the primary source of data. If any key is set, it will only be set to the new cache. After some time, the old cache can be decommissioned and the `FallbackClient` can be replaced directed with the `new_cache` client.

When communicating with a remote cache, the usual concurrency problem comes back: there might be several clients trying to access the same key at the same time. *memcached* provides a **check and set** operation, shortened to *CAS*, which helps to solve this problem.

The simplest example is an application that wants to count the number of users it has. Each time a visitor connects, a counter is incremented by 1. Using *memcached*, a simple implementation is provided in Example 12.10, "Counting the number of visitor in *memcached*".

Example 12.10. Counting the number of visitor in *memcached*

```
def on_visit(client):
    result = client.get('visitors')
    if result is None:
        result = 1
    else:
        result += 1
    client.set('visitors', result)
```

However, what happens if two instances of the application try to update this counter at the same time. The first call `client.get('visitors')` will return the same number of visitors for both of them, let's say it's 42. Then both will add 1, compute 43, and set the number of visitors to 43. That number is wrong, and the result should be 44, i.e. 42 + 1 + 1.

To solve this concurrency issue, the CAS operation of *memcached* is handy. [Example 12.11, "Using CAS in *memcached*"](#) implements the correct solution.

Example 12.11. Using CAS in *memcached*

```python
def on_visit(client):
    while True:
        result, cas = client.gets('visitors')
        if result is None:
            result = 1
        else:
            result += 1
        if client.cas('visitors', result, cas):
            break
```

The `gets` method returns the value, just like the `get` method, but it also returns a CAS value. What is in this value is not relevant, but it is used for the next method `cas` call. This method is equivalent to the `set` operation, except that it fails if the value has changed since the `gets` operation. In case of success, the loop is broken. Otherwise, the operation is restarted from the beginning.

In the scenario where two instances of the application try to update the counter at the same time, only one succeeds to move the counter from 42 to 43. The second instance gets a `False` value returned by the `client.cas` call, and have to retry the loop. It will retrieve 43 as value this time, will increment it to 44, and its `cas` call will succeed. Solving our problem.

Tip

Incrementing a counter is interesting as an example to explain how CAS works because it is simplistic. However, *memcached* also provides the `incr` and `decr` methods to increment or decrement an integer in a single request, rather than doing multiple `gets`/`cas` calls. In real-world applications `gets` and

`cas` are used for more complex data type or operations

Most remote caching server and data store provides such a mechanism to prevent concurrency issue. It is critical to be aware of those cases to make a proper use of their features.

12.4. Jason Myers on Databases

Hi Jason! Could you introduce yourself and explain how you did come to Python?

I am Jason Myers, a Python developer at Juice Analytics and an author. I got my start in Python after switching from being an Infrastructure Architect for large mission-critical networks. When I first switched to development, I spent time learning C# and working in PHP. I had written code for years in Perl, PHP, and some C. I got a job doing Python web development and never looked back. I feel in love with the highly readable, flexible language and soon the community around it. I have worked as a cloud OS engineer, a data engineer, and a web developer.

You are the author of a book around SQLAlchemy ([Essential SQLAlchemy, 2nd Edition, O'Reilly Media](#)). With your database and Python experts hat on, what do you think are the more significant mistakes developers do around those technologies that dangerously impact their scalability and performance?

I see a few things here that really can come into place.

First, people fail to use query caching like [dogpile](#) to store query results. Missing this can have a dramatic impact on page load and scalability as some queries become nothing more than a cache hit. I like to use query result caching so that I can have multiple response formatters for different outputs or even different parts of the application leverage the same cache if the query is the same.

Secondly, often people pull back way to more data than they need to answer the question. This can be can because they feel more comfortable processing the data in Python rather than in SQL functions or because they pull back full rows when they only need columns. This can also occur in ETL when people ingest more data during the ETL process than they need to answer the questions asked in their application. All of these factors lead to higher memory usage and slower query time.

Finally, I do not see people leaning on things like [pgbouncer](#) and [pgpool](#) to provide better connection pooling and smoother failovers as much as I think they should. By keeping fewer connections open longer and recycling them less often we can cut down on a lot of the overhead on the server required to service a large number of connections and the connection startup time.

Do you have any specific strategy or Python-ready solution to store results from queries? Alternatively, do you build something from scratch each time you need it?

I really like the *dogpile.cache* library for this. I have used it religiously with *SQLAlchemy*, and I also like the `cached_property`

decorator when working in *Django*. By default in *Django* all noncallable attributes of a model are cached, but with the `cached_property` decorator you can take that even further.

Is there native support for providing better connection pooling with libraries such as *psycopg2* or *SQLAlchemy* for example? Can you use them right out of the box? Is there equivalent mechanisms and tricks that can be used outside PostgreSQL?

pgbouncer and *pgpool* are PostgreSQL proxies you stand up and treat like PostgreSQL servers. *psycopg2* and *SQLAlchemy* do not even know that magic is being done for them. You can also tune the connection pools used by *SQLAlchemy* and *PostgreSQL* to have a size that matches your checkout rate. While there is a ton of words spewed at this topic, I like the groundwork laid out in *saautopool* (about 3ish years ago). It chooses a pool size that reacted to the number of currently checked out connections and the number opened on average in a fixed (5 seconds by default) window.

You recently talked about object versioning. Could you explain what they are and how they are useful?

Version objects are objects that contain an attribute that tells you what version of code they were created under so that we can have a data model within a project independent of external APIs or database schema for the purposes of providing upgrade compatibility across distributed services.

For example, you have an initial users service which is version 1: it sets the version attribute to 1 on the user model. Whenever this

model is input or output say for an API request, that version will be passed with it. This allows the receiving system to know exactly how to handle that object.

That might not sound very useful if you keep all your systems in lockstep. However, in a world of distributed systems and rolling updates, it is. When deploying the version 2 of the users model, we can do many things to separate the "new" users model structure from the old one like version our API endpoints. Though, we could also just have the API endpoint accept multiple versions and handle the differences both in input and output so one container running version 1 and one container that got upgraded to version 2 can still process requests for all the clients on the same endpoints.

This multiple version feels unusual to those who are used to the normal REST API version scheme, but it creates a very natural flow where the same endpoints can be reused and updated and act as a translation boundary not only between services but also between object versions. This method really shines in things like RPC, queues, user-facing APIs, etc, where versions might need to be fluid, but the endpoints are well known and don't lend themselves to immediate changes.

I really like to connect this with the JSONAPI spec (jsonapi.org) and in Python I find myself using marshmallow and marshmallow json-api to define my schemas. We highjack the "type" field in the JSONAPI Object definition to contain our type and a version for example "user:1" vs "user:3" in that field would be handled differently internally, but the API consumer has no idea and doesn't have to care. They request with the type they have and can upgrade to the new type when they are ready.

This concept also grants me some more piece of mind when I am working with document stores instead of an RDBMS.

Thanks Jason!

Chapter 13. Performance

Early optimization is the root of all evil.

-- Donald Knuth

There will come a time when optimizing for better performance will be the right thing to do. This should probably even be before thinking about distributing your application or dividing it up in micro-services.

Many developers focus on the wrong thing when doing optimization, guessing where Python might be slower or faster. Rather than speculating, this chapter will help you understand how to profile your application, so you will **know** what part of your program is slowing things down and where the bottlenecks are.

13.1. Memory and CPU Profiling

Profiling a Python program means doing a dynamic analysis that measures the execution time of the program and everything that involves. That means measuring the time spent in each of its functions. This data gives you info about where your program is spending time, and what area might be worth optimizing.

This is a very interesting exercise. Many people focus on local optimization, such as determining, for example, which of the Python 2 functions `range` or `xrange` is going to be faster. It turns out that knowing which one is faster may never be an issue in your program, and that the time gained by using one of the functions above might not be worth the time you spend researching it xor arguing about it with your colleague.

Trying to blindly optimize a program without measuring where it is actually spending its time is a useless exercise. Following your gut alone is not always sufficient.

There are many types of profiling, as there are many things that you can measure. Here we focus on CPU and memory utilization profiling, meaning the time spent by each function executing instructions or allocating memory.

Since its 2.5 version, Python provides a few built-in tools to help you in achieving that task. The standard one is `cProfile` and it is easy enough to use.

Example 13.1. Basic usage of `cProfile`

```
>>> import cProfile
>>> cProfile.run('2 + 2')
         2 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filenam
        1    0.000    0.000    0.000    0.000 <string
        1    0.000    0.000    0.000    0.000 {methoc
```

You can also use a script as an argument as shown in Example 13.2, "Using the `cProfile` module with a program".

Example 13.2. Using the `cProfile` module with a program

```
$ python -m cProfile myscript.py
         343 function calls (342 primitive calls) in

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filenam
        1    0.000    0.000    0.000    0.000 :0(_get
        1    0.000    0.000    0.000    0.000 :0(len)
      104    0.000    0.000    0.000    0.000 :0(seta
        1    0.000    0.000    0.000    0.000 :0(setp
        1    0.000    0.000    0.000    0.000 :0(star
```

```
   2/1      0.000      0.000      0.000      0.000 <string
     1      0.000      0.000      0.000      0.000 String
     1      0.000      0.000      0.000      0.000 String
```

The results list indicates the number of times each function was called, and the time spent on its execution. You can use the -s option to sort by any fields, e.g. -s time sorts by internal time.

While this is handy, it is a bit rough to use and parse. However, if you have coded in C, you probably already know about the fantastic Valgrind tool, that – among other things – can provide profiling data for C programs. The data that it provides can then be visualized by another great tool called KCacheGrind.

You will be happy to know that the profiling information generated by *cProfile* can easily be converted to a call tree that can be read by *KCacheGrind*. The cProfile module has a -o option that allows you to save the profiling data, and pyprof2calltree can convert from one format to the other.

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

Figure 13.1. *KCacheGrind* example
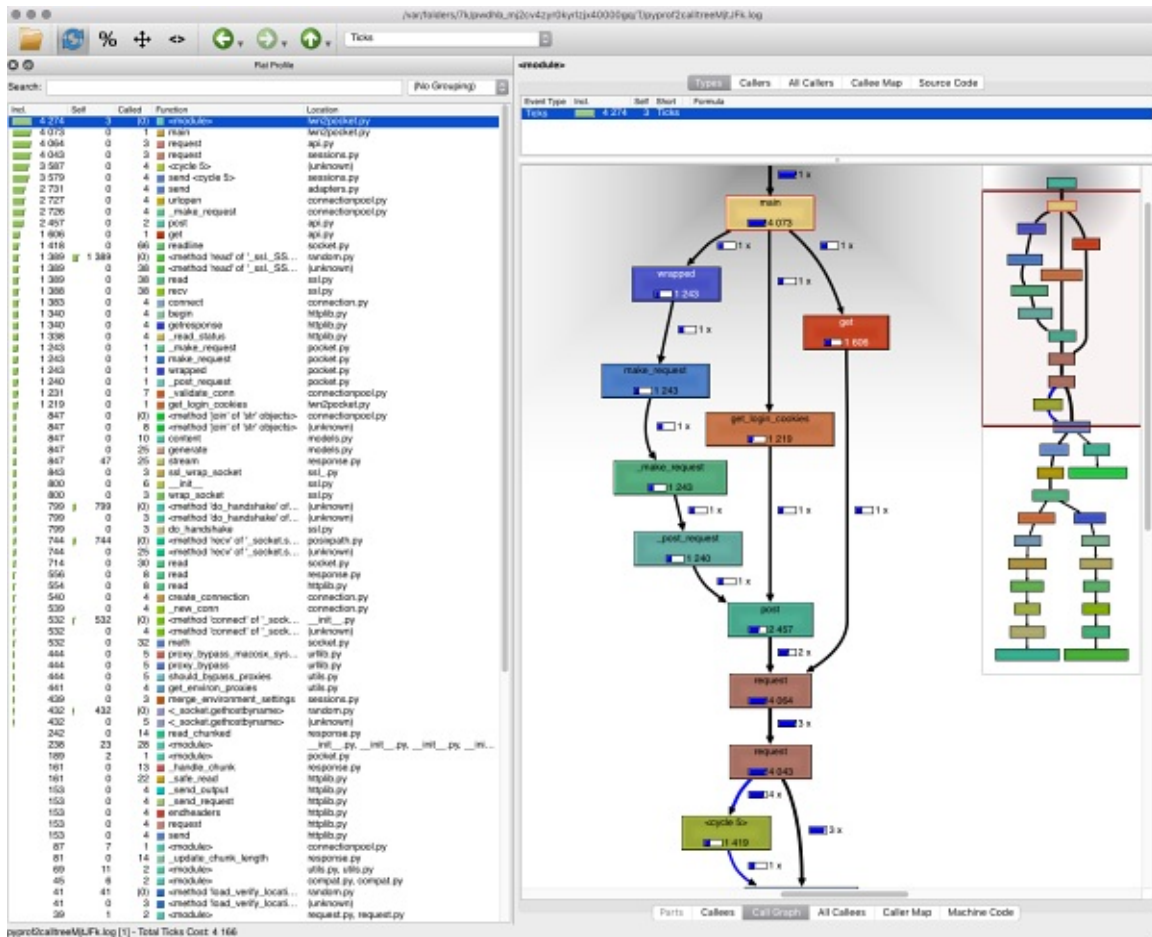
This provides a lot of information that will allow you to determine what part of your program might be consuming too many resources.

There is also another handy tool for visualizing profiling information which is called [RunSnakeRun](#). It is simple to install using `pip install runsnakerun` [8]. Then you can use it without much difficulty:

```
$ python -m cProfile -o myscript.cprof myscript.py
$ runsnake myscript.cprof
```

Figure 13.2. *RunSnake* **example**

RunSnake offers the same kind of functionality as *KCacheGrind*, though it has a little less choice regarding what's displayed. However, it feels a little more interactive, and the navigation through the profiling information might be more accessible.

If only one of them is available on your platform, that is good enough. However, if you can try and use both because, they might complement each other well until you get used to using one or the other.

13.2. Profiling Strategy and a Case

To profile a program, you need to run it. Nonetheless, running the whole program in profiling mode can generate a lot of data that you do not care about, and this adds noise to what you might be trying to understand. A

good strategy to leverage, if your program has unit or functional tests, is to use them to profile your code. They are little scenarios that can be very useful when you want to obtain profiling data.

Using tests is a good strategy for a curious and naive first-pass profiling. Though there's no way to make sure that the hot spots seen in the unit/functional tests are the actual hot spots that an application encounters in production.

This means that a profiling session executed with the same conditions as the production environment and with a scenario that mimics what is seen in this environment is often a necessity if you need to push your program optimization further and want to achieve perceptible and valuable gains.

I wrote a little library called *Carbonara* and doing time series computing. To check its performance and possibly make it faster, I profiled it using one of its unit tests named `test_fetch`. The test is pretty simple: it puts data in a time series object, and then fetch the computed aggregation result.

Figure 13.3. Carbonara profiling information before optimization

				Function	File
53	0	1		_run_core	runtest.py
53	0	(0)		_run_one	runtest.py
53	0	1		_run_prepared_result	runtest.py
52	0	15		_run_user	runtest.py
50	0	1		_run_test_method	testcase.py
50	0	1		test_fetch	test_carbonara.py
44	0	4		set_values	carbonara.py
18	0	2		update	carbonara.py
12	0	3		_first_block_timestamp	carbonara.py
11	0	5		_truncate	carbonara.py
10	0	3		_resample_timestamps	resample.py
10	0	2		_resample	carbonara.py
10	0	8		combine_first	series.py
10	0	3		resample	generic.py
10	0	3		resample	resample.py
7	0	16		reindex	generic.py
7	0	16		reindex	series.py
6	0	3		_get_binner_for_resample	resample.py
6	0	3		_get_time_bins	resample.py
6	4	252		_view_is_safe	_internal.py
6	0	9		groupby	groupby.py
5	0	10		_reindex_axes	generic.py
5	0	9		<cycle 5>	(unknown)
4	0	25		__getitem__	series.py
4	0	21		_get_with	series.py
4	0	4		f	groupby.py
4	0	9		groupby	generic.py
4	0	15		<cycle 4>	(unknown)
3	0	285		<method 'view' of 'numpy.n...	(unknown)
3	1	64		__init__	series.py
3	0	40		__new__ <cycle 1>	index.py, index.py
3	2	252		_check_field_overlap	_internal.py
3	0	3		_generate	index.py
3	0	12		aggregate <cycle 5>	groupby.py
3	2	95		delta	offsets.py
3	0	9		ngroups	groupby.py
3	0	10		reindex	index.py
3	0	9		result_index	groupby.py
3	0	32		<cycle 1>	(unknown)
2	0	2		<pandas.algos.arrmap_obj...	(unknown)
2	0	(0)		<pandas.algos.arrmap_obj...	carbonara.py
2	0	4		__init__	carbonara.py
2	0	27		__init__ <cycle 4>	groupby.py
2	0	9		_cython_agg_general <cycl...	groupby.py
2	0	3		_generate_regular_range	index.py
2	0	9		_get_grouper <cycle 4>	groupby.py
2	0	21		_get_values	series.py
2	0	6		_make_labels	groupby.py
2	0	10		_reindex_with_indexers	generic.py
2	0	20		_round_timestamp	carbonara.py
2	0	74		_simple_new	index.py, index.py
2	0	9		_wrap_aggregated_output	groupby.py
2	0	2		fetch	carbonara.py
2	0	46		get_value	index.py, index.py
2	0	12		group_index	groupby.py
2	0	2		map	base.py
1	0	342		<getattr>	(unknown)
1	0	34		__getitem__	base.py
1	0	30		_convert_slice_indexer <cy...	index.py
1	0	84		_ensure_index	index.py
1	0	8		_get_string_slice	index.py
1	0	12		_isnull_ndarraylike	common.py
1	0	15		_isnull_new	common.py
1	0	1		_run_setup	testcase.py
1	0	33		_sanitize_array	series.py
1	0	10		_slice_take_blocks_ax0	internals.py
1	0	67		asi8	index.py
1	0	6		factorize	algorithms.py
1	0	12		get_indexer	index.py
1	0	36		get_loc	index.py, index.py
1	0	11		get_slice_bound	index.py
1	0	21		get_slice	internals.py
1	0	9		infer_freq	frequencies.py
1	0	9		inferred_freq	base.py
1	0	6		is_in_axis	groupby.py
1	0	6		is_in_obj	groupby.py
1	0	15		isnull	common.py
1	0	10		reindex_indexer	internals.py
1	0	1		setUp	base.py
1	0	20		slice_indexer	index.py, index.py
1	0	11		slice_locs	index.py
1	0	10		take_nd	common.py

The list of calls in [Figure 13.3, "Carbonara profiling information before optimization"](#) shows that 88% of the ticks are spent in `set_values` (44 ticks over 50) – a tick is a computing cycle unit. This function is used to insert values into the time series, and not to fetch the values. That means that it is really slow to insert data, and pretty fast to retrieve it.

Reading the rest of the list reveals that several functions share the rest of the ticks, update, `_first_block_timestamp`, `_truncate`, `_resample`, etc. Some of the functions in the list are not part of *Carbonara*, so there's no point in looking to optimize them (yet). The only thing that can sometimes be optimized is the number of times they are called.

Figure 13.4. Carbonara call graph before optimization
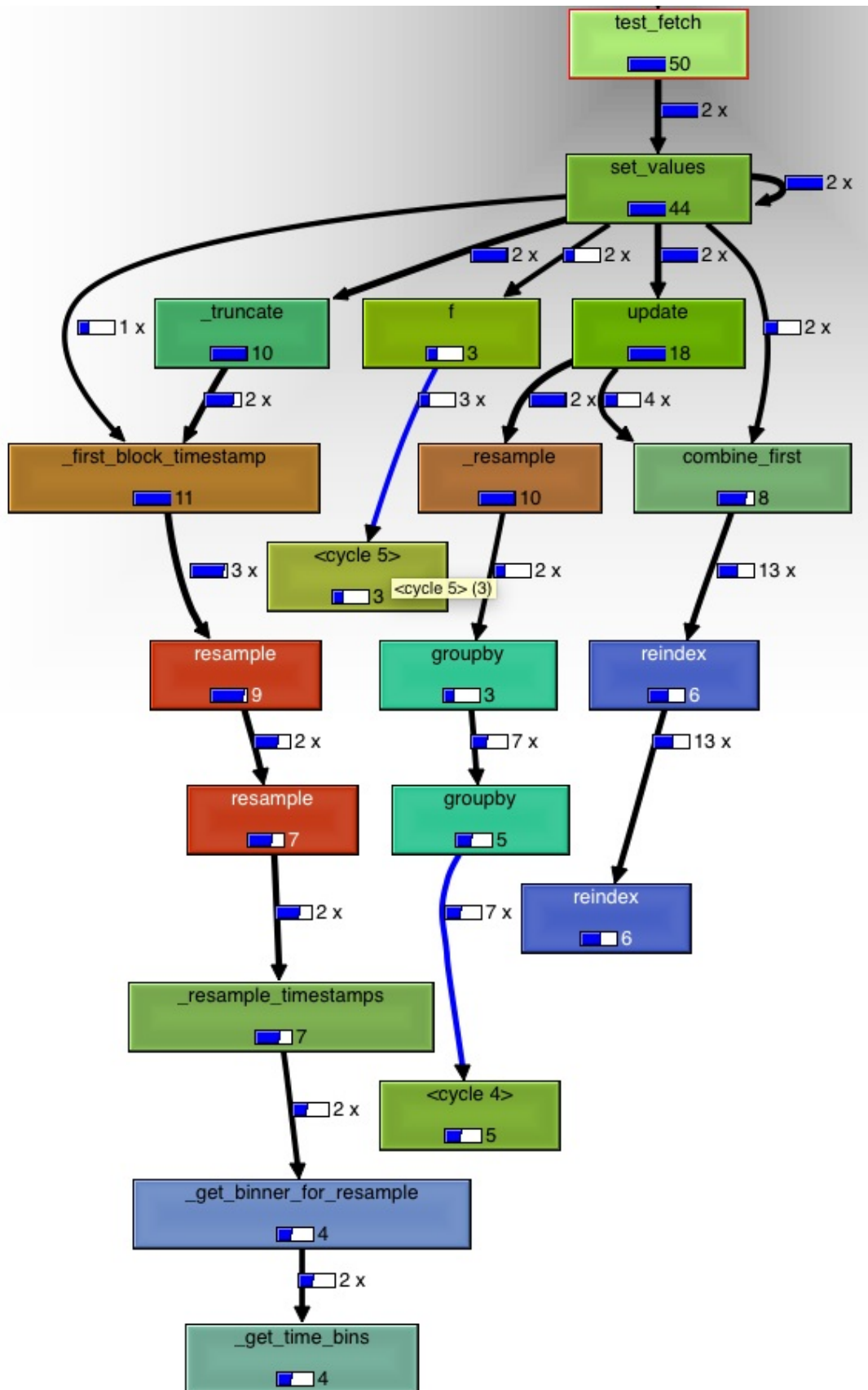
The call graph in [Figure 13.4, "Carbonara call graph before optimization"](#) gives a bit more insight into what's going on. Using knowledge about how Carbonara works internally, it does not seem like the whole stack on the left for `_first_block_timestamp` makes much sense. This function is supposed to find the first timestamp for a particular time series – for example if a time series starts with a timestamp of 13:34:45 and has a period of 5 minutes, the function should return 13:30:00. The way it works currently is by calling the resample function from Pandas [9] on a time series with only one element. However, that operation seems to be very slow: it represents 25% of the time spent by `set_values` (11 ticks on 44).

By implementing my own simpler version of that rounding algorithm [10] named `_round_timestamp`, it is possible to rewrite the code this way:

```
 def _first_block_timestamp(self):
-         ts = self.ts[-1:].resample(self.block_size)
-         return (ts.index[-1] - (self.block_size * se
+         rounded = self._round_timestamp(self.ts.inde
+         return rounded - (self.block_size * self.bac
```

Figure 13.5. Carbonara profiling information after optimization

					Function	File
	58	0	1		_run_core	runtest.py
	58	0	(0)		_run_one	runtest.py
	58	0	1		_run_prepared_result	runtest.py
	57	0	15		_run_user	runtest.py
	53	0	1		_run_test_method	testcase.py
	53	0	1		test_fetch	test_carbonara.py
	38	0	4		set_values	carbonara.py
	19	0	2		update	carbonara.py
	12	0	8		combine_first	series.py
	11	0	2		_resample	carbonara.py
	9	0	2		fetch	carbonara.py
	9	0	6		groupby	groupby.py
	9	0	16		reindex	generic.py
	9	0	16		reindex	series.py
	8	0	6		groupby	generic.py
	7	0	10		_reindex_axes	generic.py
	7	0	6		<cycle 4>	(unknown)
	6	0	22		__getitem__	series.py
	6	4	71		delta	offsets.py
	5	0	6		_cython_agg_general	groupby.py
	5	0	18		_get_with	series.py
	5	2	201		_view_is_safe	_internal.py
	4	0	4		__init__	carbonara.py
	4	0	6		_get_grouper <cycle 4>	groupby.py
	4	0	1		_run_setup	testcase.py
	4	0	6		aggregate	groupby.py
	4	0	23		nanos	offsets.py
	4	0	17		sampling	carbonara.py
	4	0	1		setUp	base.py
	3	0	18		__init__ <cycle 4>	groupby.py
	3	0	8		_get_string_slice	index.py
	3	0	6		_make_labels	groupby.py
	3	0	10		_reindex_with_indexers	generic.py
	3	0	3		_setUp <cycle 5>	tempdir.py
	3	0	4		f	groupby.py
	3	0	36		get_loc	index.py, index.py
	3	0	40		get_value	index.py, index.py
	3	0	12		group_index	groupby.py
	3	0	2		map	base.py
	3	0	2		mkdtemp	tempfile.py
	3	0	6		ngroups	groupby.py
	3	0	10		reindex	index.py
	3	0	6		result_index	groupby.py
	3	0	5		useFixture	testcase.py
	3	0	10		<cycle 5>	(unknown)
	2	0	297		<getattr>	(unknown)
	2	0	17		<method 'get_loc' of 'panda…	(unknown)
	2	0	2		<pandas.algos.arrmap_obj…	(unknown)
	2	0	(0)		<pandas.algos.arrmap_obj…	carbonara.py
	2	0	2		__delitem__	generic.py
	2	0	2		__init__	random.py
	2	0	58		__init__	series.py
	2	1	201		_check_field_overlap	_internal.py
	2	0	27		_convert_slice_indexer <cy…	index.py
	2	0	18		_get_values	series.py
	2	0	23		_round_timestamp	carbonara.py
	2	0	10		_slice_take_blocks_ax0	internals.py
	2	0	5		_truncate	carbonara.py
	2	0	2		delete	internals.py
	2	0	6		factorize	algorithms.py
	2	0	12		get_indexer	index.py
	2	0	11		get_slice_bound	index.py
	2	0	6		infer_freq	frequencies.py
	2	0	6		inferred_freq	base.py
	2	0	6		is_in_axis	groupby.py
	2	0	6		is_in_obj	groupby.py
	2	0	3		next	tempfile.py
	2	0	10		reindex_indexer	internals.py
	2	0	3		rng	tempfile.py
	2	0	2		seed	random.py
	2	0	20		slice_indexer	index.py, index.py
	2	0	11		slice_locs	index.py
	2	0	16		<cycle 2>	(unknown)
	1	0	2		<function seed at 0x10b133…	(unknown)
	1	1	(0)		<function seed at 0x10b133…	real.py
	1	0	402		<method 'append' of 'list' ob…	(unknown)
	1	1	(0)		<method 'append' of 'list' ob…	_weakrefset.py
	1	0	(0)		<method 'get_loc' of 'panda…	os.py
	1	1	(0)		<method 'get_value' of 'pan…	real.py
	1	0	240		<method 'view' of 'numpy.n…	index.py
	1	1	(0)		<range>	real.py
	1	0	25		__getitem__	base.py
	1	0	2		__getitem__	carbonara.py

Running the test again to profile the code again shows a different output. The list of functions called in [Figure 13.5, "Carbonara profiling information after optimization"](#) is different, and the amount of time spent by `set_values` dropped from 88% to 71%.

Figure 13.6. Carbonara call graph after optimization
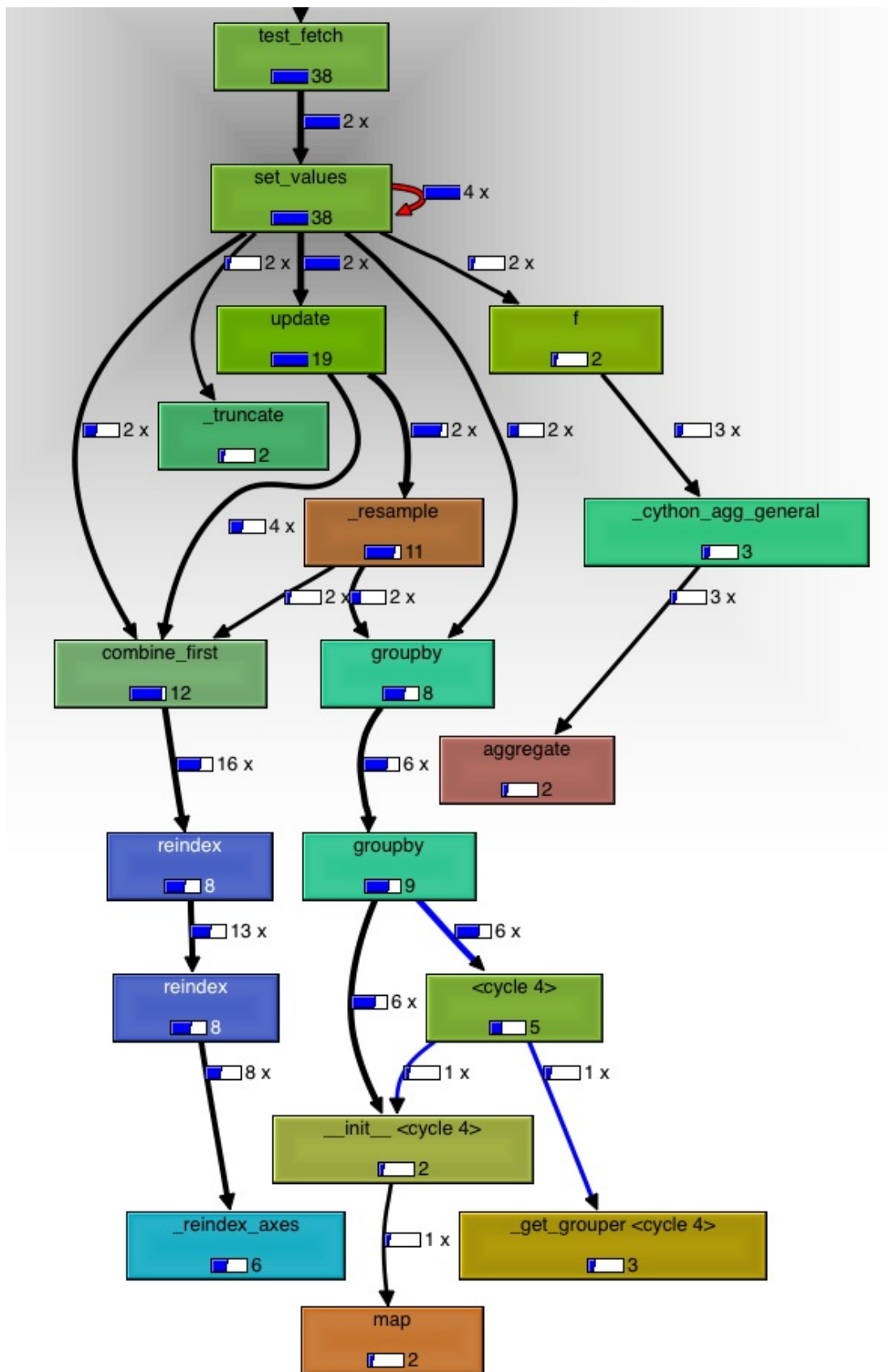
The call stack for `set_values` does not even show the calls to `_first_block_timestamp` function anymore. Indeed, it became so fast that its execution time is now insignificant, so *KCachegrind* no longer displays it.

In that case, in a matter of minutes, it was possible to achieve a 25% performance improvement with just a naive approach and using a code scenario that already exists. There should be no reason that your application cannot do that too – given that it has unit tests.

13.3. Zero-Copy

Often programs have to deal with an enormous amount of data in the form of large arrays of bytes. Handling such a massive amount of data in strings can be very ineffective once you start manipulating it through copying, slicing and modifying.

Let's consider a small program that reads a large file of binary data, and partially copies it into another file. To examine our memory usage, we will use memory_profiler, a nice Python package that allows us to see the memory usage of a program line by line.

Example 13.3. Using *memory_profiler*

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = content[1024:]
    print("Content length: %d, content to write lengt
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

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

We then run the program in [Example 13.3, "Using *memory_profiler*"](#) using *memory_profiler*:

```
$ python -m memory_profiler memoryview-copy.py
Content length: 10240000, content to write length 102
Filename: memoryview/copy.py

Mem usage      Increment    Line Contents
================================================
                            @profile
 9.883 MB      0.000 MB     def read_random():
 9.887 MB      0.004 MB         with open("/dev/urandom'
19.656 MB      9.770 MB             content = source.rea
29.422 MB      9.766 MB             content_to_write = c
29.422 MB      0.000 MB         print("Content length: '
29.434 MB      0.012 MB               (len(content), len
29.434 MB      0.000 MB         with open("/dev/null", '
29.434 MB      0.000 MB             target.write(content
```

(1) We are reading 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB of memory to store this data as a string.

(2) We copy the entire block of data minus the first kilobyte – because we won't be writing those first 1024 bytes to the target file.

What is interesting in this example is that, as you can see, the memory usage of the program is increased by about 10 MB when building the variable *content_to_write*. In fact, the slice operator is copying the entirety of *content*, minus the first KB, into a new string object.

When dealing with big amount of data, performing this kind of operation on large byte arrays is going to be a disaster. If you happen to have experience writing C code, you know that using `malloc` and `memcpy` has a significant cost, both in terms of memory usage and regarding general performance: allocation and copying memory is slow.

However, as a C programmer, you also know that strings are arrays of characters and that nothing stops you from looking at only part of this array without copying it, through the use of pointer arithmetic – assuming that the entire string is in a contiguous memory area.

This is possible in Python using objects that implement the *buffer protocol*. The buffer protocol is defined in [PEP 3118](#), which explains the C API used to provide this protocol to various types, such as strings.
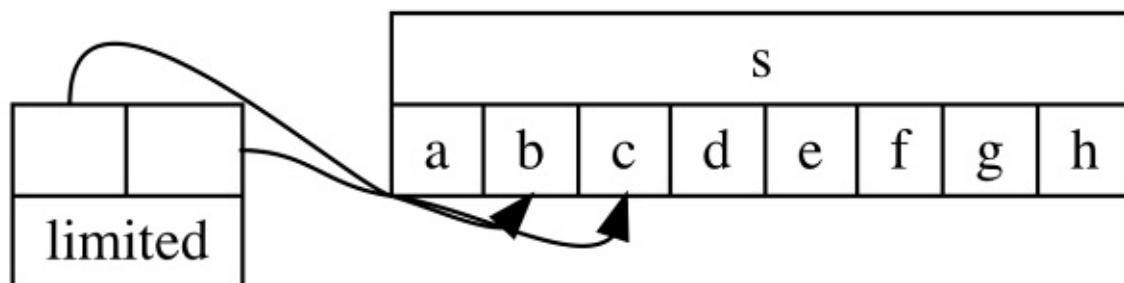
When an object implements this protocol, you can use the `memoryview` class constructor on it to build a new *memoryview* object that references the original object memory.

Here is an example:

```
>>> s = b"abcdefgh"
>>> view = memoryview(s)
>>> view[1]
98 (1)
>>> limited = view[1:3]
<memory at 0x7fca18b8d460>
>>> bytes(view[1:3])
b'bc'
```

(1) This is the ASCII code for the letter *b*.

Example 13.4. Using slice on `memoryview` objects

In this case, we are going to make use of the fact that the `memoryview`

object's slice operator itself returns a `memoryview` object. That means it does *not* copy any data, but merely references a particular slice of it.

With this in mind, we can now rewrite the program, this time referencing the data we want to write using a *memoryview* object.

```python
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = memoryview(content)[1024:]
    print("Content length: %d, content to write lengt
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)


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

And this program will have half the memory usage of the first version:

```
$ python -m memory_profiler memoryview/copy-memoryvie
Content length: 10240000, content to write length 102
Filename: memoryview/copy-memoryview.py

Mem usage      Increment    Line Contents
================================================
                            @profile
 9.887 MB      0.000 MB     def read_random():
 9.891 MB      0.004 MB         with open("/dev/urandom'
19.660 MB      9.770 MB             content = source.rea
19.660 MB      0.000 MB             content_to_write = r
19.660 MB      0.000 MB         print("Content length: %
19.672 MB      0.012 MB             (len(content), ler
19.672 MB      0.000 MB         with open("/dev/null", '
19.672 MB      0.000 MB             target.write(content
```

(1) Read 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB of memory to store this data as a string.

(2) Reference the entire block of data minus the first KB – because we don't write this first kilobyte to the target file. No copying means that no more memory is used!

This kind of trick is especially useful when dealing with sockets. As you may know, when data is sent over a socket, it might not send all the data in a single call. A simple implementation would be to write:

```python
import socket

s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000)  (1)
while data:
    sent = s.send(data)
    data = data[sent:]  (2)
```

(1) Build a bytes object with more than 100 millions times the letter `a`.

(2) Remove the first `sent` bytes sent.

Obviously, using such a mechanism, you are going to copy the data over and over until the socket has sent everything. Using *memoryview*, we can achieve the same functionality without copying data – hence, zero copy:

```python
import socket

s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000)  (1)
mv = memoryview(data)
while mv:
    sent = s.send(mv)
```

```
    mv = mv[sent:] (2)
```

<u>(1)</u> Build a bytes object with more than 100 millions times the letter *a*.

<u>(2)</u> Build a new `memoryview` object pointing to the data that remains to be sent.

This code does not copy anything, and it won't use any more memory than the 100 MB initially needed for our *data* variable.

We have now seen `memoryview` objects used to write data efficiently, but the same method can also be used to **read** data. Most I/O operations in Python know how to deal with objects implementing the buffer protocol. They can read from it, and they can also write to it. In this case, we do not need `memoryview` objects – we can just ask an I/O function to write into our pre-allocateds object:

```
>>> ba = bytearray(8)
>>> ba
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba)
...
8
>>> ba
bytearray(b'`m.z\x8d\x0fp\xa1')
```

With such techniques, it is easy to pre-allocate a buffer (as you would do in C to mitigate the number of calls to `malloc`) and fill it at your convenience. Using *memoryview*, you can even place data at any point in the memory area:

```
>>> ba = bytearray(8)
>>> ba_at_4 = memoryview(ba)[4:] (1)
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba_at_4) (2)
```

```
...
4
>>> ba
bytearray(b'\x00\x00\x00\x00\x0b\x19\xae\xb2')
```

(1) We reference the *bytearray* from offset 4 to its end.

(2) We write the content of /dev/urandom from offset 4 to the end of the *bytearray*, effectively reading 4 bytes only.

Tip

Both the objects in the *array* module and the functions in the *struct* module can handle the buffer protocol correctly, and they can, therefore, perform efficiently when targeting zero copy.

A typical pattern in network-oriented applications is to send files to a remote client. It usually means opening a file, reading its contents and writing this to a socket:

```python
import socket

s = socket.socket(…)
s.connect(…)
with open("file.txt", "r") as f:
    content = f.read()
s.send(content)
```

The downside with such a piece of code is that it needs to allocate and copy a big piece of memory to store the contents of the file in the `content` variable before writing it. That is slow.

Modern operating systems provide a system call named `sendfile` that solves this problem. This system call is not portable, but if you target the right systems then it is good to use it. This system call is exposed as `os.sendfile(out, in, offset, count)` in Python: `out` is the socket to write to, `in` is the file descriptor to read from, `offset` indicates

at which byte number to start reading the file and `count` the number of bytes to copy from it.

A higher-level wrapper is provided as `socket.socket.sendfile`:

```
import socket
import os

s = socket.socket(…)
s.connect(…)
with open("file.txt", "r") as f:
    s.sendfile(f)
```

The `sendfile` method passes the correct file descriptors to the operating system, making sure the Python program does not have any memory to allocate – making it as fast as possible!

Stay aware of how you allocate and copy data when using Python. It is a high-level language that covers the plumbing from the developers, but it often has a hidden cost.

13.4. Disassembling Code

Sometimes, it helps to have a microscopic view of some part of the code. When that is the case, I find it better to rely on the `dis` module to find out what's going on behind the scenes. The `dis` module is a disassembler for Python byte code. It is simple enough to use:

```
>>> def x():
...     return 42
...
>>> import dis
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
```

The `dis.dis` function disassembles the function that you passed on as a parameter, and prints the list of bytecode instructions that are run by the function. It can be useful to understand what's really behind each line of code that you write in order to be able to optimize your code correctly.

The following code defines two functions, each of which does the same thing – concatenates three letters:

```python
abc = ('a', 'b', 'c')

def concat_a_1():
    for letter in abc:
            abc[0] + letter

def concat_a_2():
    a = abc[0]
    for letter in abc:
            a + letter
```

Both appear to do the same thing, but if we disassemble them, we see that the generated bytecode is a bit different:

```
>>> dis.dis(concat_a_1)
  2           0 SETUP_LOOP              26 (to 29)
              3 LOAD_GLOBAL             0 (abc)
              6 GET_ITER
        >>    7 FOR_ITER               18 (to 28)
             10 STORE_FAST              0 (letter)

  3          13 LOAD_GLOBAL             0 (abc)
             16 LOAD_CONST              1 (0)
             19 BINARY_SUBSCR
             20 LOAD_FAST               0 (letter)
             23 BINARY_ADD
             24 POP_TOP
             25 JUMP_ABSOLUTE           7
```

```
          >>    28 POP_BLOCK
          >>    29 LOAD_CONST              0 (None)
                32 RETURN_VALUE
```

```
 >>> dis.dis(concat_a_2)
   2             0 LOAD_GLOBAL             0 (abc)
                 3 LOAD_CONST              1 (0)
                 6 BINARY_SUBSCR
                 7 STORE_FAST              0 (a)

   3            10 SETUP_LOOP             22 (to 35)
                13 LOAD_GLOBAL             0 (abc)
                16 GET_ITER
          >>    17 FOR_ITER               14 (to 34)
                20 STORE_FAST              1 (letter)

   4            23 LOAD_FAST               0 (a)
                26 LOAD_FAST               1 (letter)
                29 BINARY_ADD
                30 POP_TOP
                31 JUMP_ABSOLUTE          17
          >>    34 POP_BLOCK
          >>    35 LOAD_CONST              0 (None)
                38 RETURN_VALUE
```

In the second version, we store `abc[0]` in a temporary variable before running the loop. This variable makes the bytecode executed inside the loop a little smaller, as we avoid having to do the `abc[0]` lookup for each iteration. Measured using `timeit`, the second version is 10% faster than the first one; it takes a whole microsecond less to execute! Obviously, this microsecond is not worth the optimization unless you call this function millions of times – but this is the kind of insight that the `dis` module can provide.

Whether you really need to rely on such "tricks" as storing the value outside the loop is debatable – ultimately, it should be the compiler's job

to optimize this kind of thing. On the other hand, as the Python language is amazingly dynamic, it is difficult for the compiler to be sure that an optimization would not result in contradictory side effects. The takeaway: be careful when writing your code!

Another bad habit I have often encountered when reviewing code is the defining of functions inside functions for no reason. It has a cost, as the function is going to be redundantly redefined over and over again.

Example 13.5. A function defined in a function, disassembled

```
>> import dis
>>> def x():
...       return 42
...
>>> dis.dis(x)
  2             0 LOAD_CONST               1 (42)
                3 RETURN_VALUE
>>> def x():
...       def y():
...               return 42
...       return y()
...
>>> dis.dis(x)
  2             0 LOAD_CONST               1 (<code obj
                3 MAKE_FUNCTION            0
                6 STORE_FAST               0 (y)

  4             9 LOAD_FAST                0 (y)
               12 CALL_FUNCTION            0
               15 RETURN_VALUE
```

We can see in [Example 13.5, "A function defined in a function, disassembled"](#) that the code is needlessly complicated, calling `MAKE_FUNCTION`, `STORE_FAST`, `LOAD_FAST` and `CALL_FUNCTION` instead of just `LOAD_CONST`. That requires more opcodes for no good reason – and function calling in Python is already inefficient.

The only case in which it is required to define a function within a function is when building a function closure, and this is a perfectly defined use case in Python's opcodes.

Example 13.6. Disassembling a closure

```
>>> def x():
...     a = 42
...     def y():
...             return a
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 STORE_DEREF              0 (a)

  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code obj
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (y)

  5          21 LOAD_FAST                0 (y)
             24 CALL_FUNCTION            0
             27 RETURN_VALUE
```

The *dis* module is a great tool for digging into the code of your program. Together with profiling, it is a powerful tool when it comes to understanding why a piece of code might be slow.

13.5. Victor Stinner on Performance

Hi Victor! Could you introduce yourself and explain how you came to Python?

Hi, my name is Victor Stinner, I am working for Red Hat on OpenStack, and I have been a CPython core developer since 2010.

I was always programming. I tried a wide range of programming languages from the lowest level Intel x86 assembler to high-level languages like Javascript and BASIC. Even if I now really enjoy writing C code for best performance, Python fits my requirements better for my daily job. Since it is easy to write Python code, and I am not annoyed by memory management or analyzing crashes, I use the "free" time to write more unit tests, take care of the coding style, and do all the tiny stuff which makes software into a "good software".

After 10 years of professional programming, I can now say that I spent more time on reading "old" code and fixing old twisted corner cases, than writing new code from scratch. Having an extensible test suite makes me relaxed. Having to work under pressure without a net is likely to lead to burnout, or more simply to quit a job.

I completely agree. You spent so much time analyzing performances around Python – how does having a test suite help with that? Are there any other sound habits that can help with profiling or code optimization?

I spent multiple months experimenting with large Python changes to

globally optimize Python, but my work was quickly blocked by the benchmarks. CPython had a benchmark suite called "benchmarks" made up of 44 benchmarks, but there was no documentation on how to run it, especially how to get reproducible and reliable results.

I spent six more months investigating why benchmark results changed for no (apparent) reasons. The list of reasons is quite long: system noise ("jitter") and noisy applications, advanced CPU features like Turbo Boost, code placement (CPU instruction cache), average versus minimum, etc. I just [gave a talk at FOSDEM (Belgium, Brussels) describing these issues](#).

My goal was to provide a benchmark suite usable by everyone, so I implemented a new *perf* module, which provides a simple API to run benchmarks: it spawns multiple worker processes sequentially and computes the average of samples, ignoring warm-up samples.

I rewrote the "benchmarks" project using my new *perf* module. I renamed the project as "performance" to be able to publish it on PyPI (to make it as easy as possible to use it) and moved it from `hg.python.org` (*Mercurial*) to `github.com` (*Git*) since Python has moved to *GitHub*.

I cleaned up the performance code and modified it to make it more and more reliable. In the end, I removed old benchmark results from [speed.python.org](#) and published new results computed with performance and the best practice to get stable results. I am happy because results were very stable for a whole year!

I can trust benchmark results again. It became possible to identify performance regressions.

More generally, when you run benchmarks: don't run them once, run them multiple times, regularly, for example once a week during a month. It is common that results are very stable for 2 weeks and that "suddenly" become 20% slower or 20% faster even if no link is apparent between the code changes and the performance changes.

For profiling, I use a lot the Linux perf tool which is amazing. `perf stat --repeat` computes the average and standard deviation on low-level metrics like CPU caches, memory loads, etc. "perf record" + "perf report" is a lightweight statistical profiler: it collects the stack trace up to 100,000 times per second, and analyzes in which functions your code spent the most time. The tool works on a single process or even system-wide! Sadly, it does not support Python frames yet, and so it only shows C functions on a Python application.

Finally, don't reinvent the wheel: reuse existing well optimized Python built-in types and existing libraries on PyPI.

You have spent a lot of time writing those performance tools to run benchmarks because you try to make Python go faster. What is the greatest improvement you made?

Sadly, it is quite hard to get tremendous improvement in CPython because of the backward compatibility, especially of the C API. This API is very important for a large part of our community (particularly in the sciences area, *SciPy*, *NumPy*, *Pandas*, etc.). I was told recently that without its C API, the Python language would never be as popular as it is. CPython is a super cool glue language, which allows reusing battle-tested code written years ago like BLAS library written in FORTRAN 77.

The CPython C API uses reference counting, a specific implementation of a garbage collector, C structures, memory allocators, the well known GIL (Global Interpreter Lock), etc. In short, these things cannot be changed, so tricks are required to find room for performance improvements.

In Python 3.6, multiple optimizations were implemented.

The bytecode format changed from variable size (1 or 3 bytes) to fixed size (2 bytes units). The bytecode to call functions (CALL_FUNCTION) was also redesigned and optimized. To call a function with keyword arguments, the name of the keywords are now packed into a constant tuple to reduce the number of LOAD_CONST instructions.

I implemented debug hooks to ensure that applications hold the GIL when calling `PyMem_Malloc()`. It is now possible to use debug hooks on a release build using `PYTHONMALLOC=debug` environment variable. Previously, Python had to be rebuilt in debug mode which caused various issues. Thanks to these checks, I was allowed to modify `PyMem_Malloc()` to use Python fast memory allocator optimized for small objects with a short lifetime, rather than using system `malloc()`.

I spent most of my time on FASTCALL: an internal optimization to avoid creating a temporary tuple to pass positional arguments, and also avoiding a temporary dictionary to pass keyword arguments. The temporary tuple only has a cost of 20 nanoseconds: it is not a lot, but many Python functions are implemented in C and take less than 100 nanoseconds: the tuple represented 20% of its runtime!

Where do you see Python performances going to over the next few years?

I hope that someone will succeed in writing a JIT compiler for CPython. Sadly, Pyston and Pyjion, two projects adding a JIT compiler to CPython, are no longer being actively developed :-(

I also hope that Larry Hastings *gilectomy* project to remove the GIL from CPython succeeds in making CPython faster when using multiple CPUs.

At least, PyPy is already much faster than CPython. Its emulation layer for the CPython C API is faster on each release, and they are now actively working on Python 3 support.

Do you have any useful tricks or advice about things to avoid when writing Python code because they are slow or slower?

If you start optimizing a project with "micro-optimizations", you are doing things backward. You should start at the very high level, take a look at the "big picture", at the architecture of the whole project: not on individual components, but more relations between components. For example for a Web server, start by feeding all CPUs with enough worker processes. Use a profiler to identify in which parts of the code the project spends most of its time. Moreover, never forget to run benchmarks. An optimization not validated by a benchmark is worthless. It is common that an "optimization" makes a function slower in practice because of wrong assumptions.

Even if your project spends 90% of its time on a single Python instruction, it does not mean that you must optimize this specific

instruction. It only means that you should start to look at code **around** this instruction. For example, a crazy optimization will never be as efficient as a very simple cache to avoid calling the function. Adding a cache takes a few seconds, so it is easy to experiment, whereas writing a cryptic optimization can take several days and introduce subtle bugs. Later you may want to spend more time on tuning the cache size, cache invalidation, etc.

The Internet is full of bad advice on micro-optimization, which can make the code uglier and more expensive to maintain. Just don't waste your time on micro-optimizations and use faster algorithms and data structures. ;-)

Thanks Victor!

---

[8] Unfortunately, it seems that for now RunSnakeRun only still works with Python 2.

[9] Pandas is a Python data analysis library, see http://pandas.pydata.org.

[10] One in pure Python that does not need Pandas at all.

Epilogue

With this book, I tried to give you a handful of tips and a high-level view of how applications can be made in a distributed and scalable way in Python. As we have seen, there are plenty of architectural decisions that are not specific to Python, such as using a queue-based system to distribute jobs or leveraging a consistent hash ring.

Python itself does not solve all the problems, and as you have noticed, your future application will have to depend on external systems that do a great job with caching or building consensus. Some of those services deserve entire dedicated books, so I only covered them briefly here. It will be up to you to discover such systems more thoroughly and learn the specific characteristics and features.

Nevertheless, I think Python can shine in large-scale applications, and I hope this book makes that clear. There are certainly plenty of traps to avoid falling into, for sure – the chapters you've just read should guide you in the right direction.

If you ever want to learn more about Python development tips and improve your coding skills in general, I would encourage you to read my first book entitled *The Hacker's Guide to Python*!

In the meantime, have fun building distributed and scalable applications in Python. Happy hacking!