Sync vs. Async Python: What is the Difference? (/post/sync-vs-async-python-what-is-the-difference)

September 8 2020

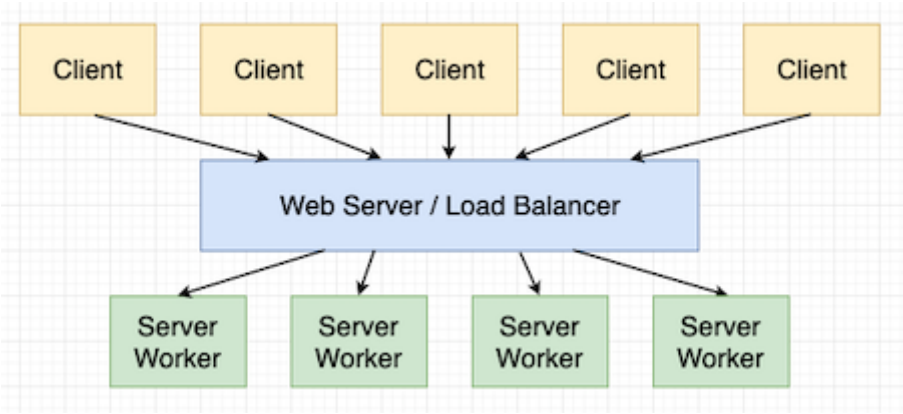
Posted by **Miguel Grinberg (/author/Miguel Grinberg)** under **Python (/category/Python)**.

Have you heard people say that async Python code is faster than "normal" (or sync) Python code? How can that be? In this article I'm going to try to explain what async is and how it differs from normal Python code.

What Do "Sync" and "Async" Mean?

Web applications often have to deal with many requests, all arriving from different clients and within a short period of time. To avoid processing delays it is considered a must that they should be able to handle several requests in parallel, something commonly known as *concurrency*. I will continue to use web applications as an example throughout this article, but keep in mind that there are other types of applications that also benefit from having multiple tasks done concurrently, so this discussion isn't specific to the web.

The terms "sync" and "async" refer to two ways in which to write applications that use concurrency. The so called "sync" servers use the underlying operating system support of threads and processes to implement this concurrency. Here is a diagram of how a sync deployment might look:



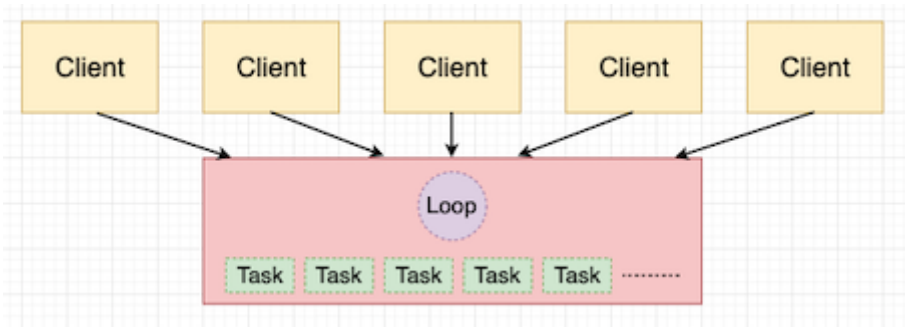
In this situation we have five clients, all sending requests to the application. The public access point for this application is a *web server* that acts as a *load balancer* by distributing the requests among a pool of *server workers*, which might be implemented as processes, threads or a

combination of both. The workers execute requests as they are assigned to them by the load balancer. The application logic, which you may write using a web application framework such as Flask or Django, lives in these workers.

This type of solution is great for servers that have multiple CPUs, because you can configure the number of workers to be a multiple of the number of CPUs, and with this you can achieve an even utilization of your cores, something that a single Python process cannot do due to the limitations imposed by the Global Interpreter Lock (GIL) (https://en.wikipedia.org/wiki/Global_interpreter_lock).

In terms of disadvantages, the diagram above clearly shows what the main limitation of this approach is. We have five clients, but only four workers. If these five clients send their requests all at the same time, then the load balancer will be able to dispatch all but one to workers, and the request that lost the race will have to remain in a queue while it waits for a worker to become available. So four of the five clients will receive their responses timely, but one of them will have to wait longer for it. The key in making the server perform well is in choosing the appropriate number of workers to prevent or minimize blocked requests given the expected load.

An asynchronous server setup is harder to draw, but here is my best take:



This type of server runs in a single process that is controlled by a *loop*. The loop is a very efficient task manager and scheduler that creates tasks to execute the requests that are sent by clients. Unlike server workers, which are long lived, an async task is created by the loop to handle a specific request, and when that request is completed the task is destroyed. At any given time an async server may have hundreds or even thousands of active tasks, all doing their own work while being managed by the loop.

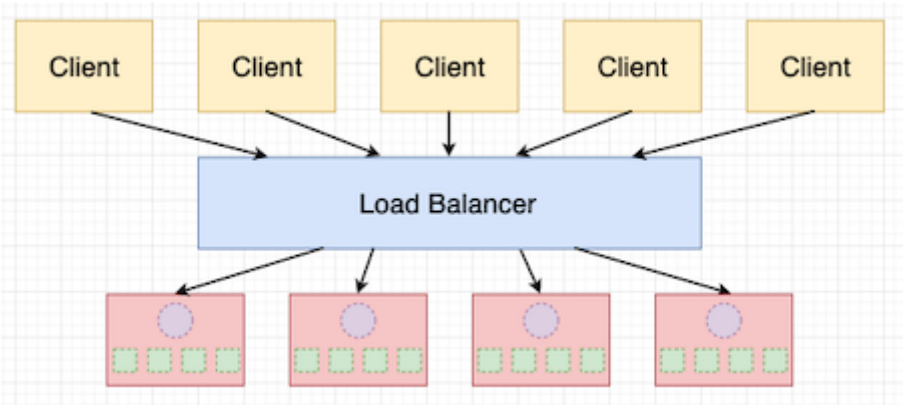
You may be wondering how is the parallelism between async tasks achieved. This is the interesting part, because an async application relies exclusively on cooperative multitasking (https://en.wikipedia.org/wiki/Cooperative_multitasking) for this. What does this mean? When a task needs to wait for an external event, like for example, a response from a database server, instead of just waiting like a sync worker would do, it tells the loop what it needs to wait for and

then returns control to it. The loop then is able to find another task that is ready to run while this task is blocked by the database. Eventually the database will send a response, and at that point the loop will consider that first task ready to run again, and will resume it as soon as possible.

This ability for an async task to suspend and resume execution may be difficult to understand in the abstract. To help you apply this to things that you may already know, consider that in Python, one way to implement this is with the `await` or `yield` keywords, but these aren't the only ways as you will see later.

An async application runs entirely in a single process and a single thread, which is nothing short of amazing. Of course this type of concurrency takes some discipline, since you can't have a task that holds on to the CPU for too long or else the remaining tasks starve. For async to work, all tasks need to voluntarily suspend and return control to the loop in a timely manner. To benefit from the async style, an application needs to have tasks that are often blocked by I/O and don't have too much CPU work. Web applications are normally a very good fit, in particular if they need to handle large amounts of client requests.

To maximize the utilization of multiple CPUs when using an async server, it is common to create a hybrid solution that adds a load balancer and runs an async server on each CPU, as shown in the following diagram:



Two Ways to Do Async in Python

I'm sure you know that to write an async application in Python you can use the asyncio package (https://docs.python.org/3/library/asyncio.html), which builds on top of *coroutines* to implement the suspend and resume features that all asynchronous application require. The `yield` keyword, along with the newer `async` and `await`, are the foundation on which the async capabilities of `asyncio` are built. To paint a complete picture, there are other coroutine-based async solutions in the Python ecosystem, such as Trio (https://trio.readthedocs.io/en/stable/),

and Curio (https://curio.readthedocs.io/en/latest/). There is also Twisted (https://twistedmatrix.com/trac/), which is the oldest coroutine framework of all, even predating `asyncio`.

If you are interested in writing an async web application, there are a number of async frameworks based on coroutines to choose from, including aiohttp (https://docs.aiohttp.org/en/stable/), sanic (https://sanic.readthedocs.io/en/latest/), FastAPI (https://fastapi.tiangolo.com/) and Tornado (https://www.tornadoweb.org/en/stable/).

What a lot people don't know, is that coroutines is just one of the two methods available in Python to write asynchronous code. The second way is based on a package called greenlet (https://greenlet.readthedocs.io/en/latest/) that you can install with pip. Greenlets are similar to coroutines in that they also allow a Python function to suspend execution and resume it at a later time, but the way in which they achieve this is completely different, which means that the async ecosystem in Python is fractured in two big groups.

The interesting difference between coroutines and greenlets for async development is that the former requires specific keywords and features of the Python language to work, while the latter does not. What I mean by this is that coroutine-based applications need to be written using a very specific syntax, while greenlet-based applications look exactly like normal Python code. This is very cool, because under certain conditions it enables sync code to be executed asynchronously, something the coroutine-based solutions such as `asyncio` cannot do.

So what is the equivalent of `asyncio` on the greenlet side? I know of three async packages based on greenlets: Gevent (http://www.gevent.org/), Eventlet (https://eventlet.net/) and Meinheld (https://meinheld.org/), though the last one is more a web server than a general purpose async library. All have their own implementation of an async loop, and they provide an interesting "monkey-patching" feature that replaces the blocking functions in the Python standard library, such as those that do networking and threading, with equivalent non-blocking versions implemented on top of greenlets. If you have a piece of sync code that you want to run asynchronously, there is a good chance these packages will let you do it.

You are going to be surprised by this. To my knowledge, the only web framework that has explicit support for greenlets is no other than Flask (https://flask.palletsprojects.com/). This framework automatically detects when you are running on a greenlet web server and adjusts itself accordingly, without any need for configuration. When doing this, you need to be careful to not call blocking functions, or if you do, then use monkey-patching to "fix" those blocking functions.

But Flask isn't the only framework that can benefit from greenlets. Other web frameworks such as Django (https://www.djangoproject.com/) and Bottle (https://bottlepy.org/docs/dev/), which have no knowledge of greenlets, can also function asynchronously when paired with a greenlet web server and blocking functions are monkey-patched.

Is Async Faster Than Sync?

There is a widely spread misconception with regards to the performance of sync and async applications. The belief is that async applications are significantly faster than their sync counterparts.

Let me clarify this so that we are all on the same page. Python code runs at exactly the same speed whether it is written in sync or async style. Aside from the code, there are two factors that can influence the performance of a concurrent application: context-switching and scalability.

Context-Switching

The effort that is required to share the CPUs fairly among all the running tasks, which is called context-switching, can affect the performance of the application. In the case of sync applications, this work is done by the operating system and is basically a black box with no configuration or fine tuning options. For async applications, context-switching is done by the loop.

The default loop implementation provided by `asyncio`, which is written in Python, is not considered to be very efficient. The uvloop (https://github.com/MagicStack/uvloop) package provides an alternative loop that is partly implemented in C code to achieve better performance. The event loops used by Gevent and Meinheld are also written in C code. Eventlet uses a loop written in Python.

A highly optimized async loop is likely more efficient in doing context-switching than the operating system, but in my experience, to be able to see a tangible performance gain you would have to be running at really high levels of concurrency. For most applications, I do not believe the performance difference between sync and async context switches amount to anything significant.

Scalability

What I believe is the source of the myth that async is faster is that async applications often lead to a more efficient use of the CPUs, due to their ability to scale much better and in a more flexible way than sync.

Consider what would happen to the sync server shown in the diagram above if it were to receive one hundred requests all at the same time. This server cannot handle more than four requests at a time, so most of those requests will be in a queue for a while before they can get a worker assigned.

Contrast that with the async server, which would immediately create one hundred tasks (or 25 in each of the four async workers if using the hybrid model). With an async server, all requests would begin processing without having to wait (though to be fair, there may be other bottlenecks

down the road that slow things down, such as a limit on the number of active database connections).

If these hundred tasks make heavy use of the CPU, then the sync and async solutions would have similar performance, since the speed at which the CPU runs is fixed, Python's speed of executing code is always the same and the work to be done by the application is also equal. But if the tasks need to do a lot of I/O operations, then the sync server may not be able to achieve high CPU utilization with just four concurrent requests. The async server, on the other side, will certainly be better at keeping the CPUs busy because it runs all hundred requests in parallel.

You may be wondering why can't you run one hundred sync workers, so that the two servers have the same concurrency. Consider that each worker needs to have its own Python interpreter with all the resources associated with it, plus a separate copy of the application with its own resources. The sizes of your server and your application will determine how many worker instances you can run, but in general this number isn't very high. Async tasks, on the other side, are extremely lightweight and all run in the context of a single worker process, so they have a clear advantage.

Keeping all of this in mind, we can say that async could be faster than sync for a given scenario only when:

- There is high load (without high load there is no advantage in having access to high concurrency)
- The tasks are I/O bound (if the tasks are CPU bound, then concurrency above the number of CPUs does not help)
- You look at average number of requests handled per unit of time. If you look at individual request handling times you will not see a big difference, and async may even be slightly slower due to having more concurrent tasks competing for the CPU(s).

Conclusion

I hope this article clears some of the confusion and misunderstandings regarding async code. The two important takeaways that I hope you remember are:

- An async application will only do better than a sync equivalent under high load.
- Thanks to greenlets, it is possible to benefit from async even if you write normal code and use traditional frameworks such as Flask or Django.

If you'd like to understand more in detail how asynchronous systems work, check out my PyCon presentation Asynchronous Python for the Complete Beginner (https://www.youtube.com/watch?v=iG6fr81xHKA) on YouTube.

Do you have any lingering questions regarding differences between sync and async? Let me know below in the comments!
