Using Async Functions in Python 3.5

Justus Perlwitz

2015-09-17

Contents

With the new async syntax in Python 3.5, defining asynchronous functions has become a lot simpler. In this article, I will demonstrate a simple example for this new feature.

It will involve pulling a set of homepages of popular websites and displaying the first 10 characters of every HTTP response. The example will utilize the awesome aiohttp library. Make sure that your machine has aiohttp and Python 3.5 installed.

1 A Traditional, Synchronous Approach

First, let's take a look at how this would have been solved in a naive, synchronous fashion. Let's define our set of URLs that we want to retrieve.

```
sites = [
    'https://www.google.com',
    'https://www.yahoo.com',
    'https://www.bbc.co.uk',
    'https://en.wikipedia.org',
    'https://de.wikipedia.org',
    'https://news.ycombinator.com',
    'https://www.tagesschau.de',
]
FORMAT_STRING ="{site.url:<30.30}: {site.text:.10} in {site.elapsed}s"
```

We also need some logic to retrieve the pages.

```
from requests import get

def get_site_snippet(site):
    return FORMAT_STRING.format(site=get(site))

def main():
    for site in sites:
        print(get_site_snippet(site))

main()
```

Once we run our example, we will see the following response

```
https://www.google.com/       : <!doctype  in 0:00:00.798940s
https://www.yahoo.com/        : <?xml vers in 0:00:00.883083s
http://www.bbc.co.uk/         : <!DOCTYPE  in 0:00:01.479646s
https://en.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.172367s
https://de.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.166793s
https://news.ycombinator.com/ : <html op=" in 0:00:01.464526s
https://www.tagesschau.de/    : <!DOCTYPE  in 0:00:00.991291s
```

So far so good. But knowing how this kind of code works, we quickly realize that a lot of time is being spent waiting for and being blocked by external resources. Usually, HTTP servers take their time to respond, and depending on a multitude of factors, results are almost always not being delivered instantly.

Since one HTTP request has to be finished in order to execute the next HTTP request, our program will lose a considerable amount of time being blocked by an external resource. This is also known as an I/O bound computation.

By interleaving the execution of several function calls at once, we will be able to save a considerable amount of time. In our example, this would mean that while one function call is busy retrieving HTTP results, another function call can already get the next site name and initiate the next HTTP request.

2 Async All The Things

We are going to execute the same task, but with a twist. Instead of synchronously executing the `get_site_snippet` function, we are going to asynchronously get all website results and join the results in the end. Let's take a look at how to achieve that.

```
from asyncio.client import get

async def async_get_site_snippet(site):
    response = await get(site)
    content = await response.read()
    return FORMAT_STRING.format(site=content)
```

A keen observer will immediately notice the usage of `async` and `await`. I won't try to get too much into the details of how these are being handled in CPython internally. Let's just say that this means that the functions we're calling will not return the desired result immediately. Instead, calling an `async` function, will return a promise to eventually calculate a result. To be more precise, calling `async_get_site_snippet('http://www.google.com')` will return a coroutine object. The same applies for the two `await` calls, as `get(site)` returns a promise, as well as `response.read()`.

Now, by itself the coroutine object will not do anything. Getting to the result of every coroutine call involves some extra functionality that we are going to add now.

First, a list of tasks needed to be created, containing all the coroutine objects that need to be run.

```
from asyncio import get_event_loop, wait

def async_main():
    tasks = [async_get_site_snippet(site) for site in sites]
```

Then, we create a BaseEventLoop object, that runs all our tasks until they all have completed. In order to execute all coroutines concurrently, the list of tasks needs to be wrapped with `asyncio.wait`. The `BaseEventLoop` object can then run the wrapped tasks until all results are returned.

```
    loop = get_event_loop()
    # We can safely discard pending futures
    result, _ = loop.run_until_complete(wait(tasks))
    loop.close()
```

The results of `loop.run_until_complete` are now contained in a list and are ready to be retrieved.

```
    for task in result:
        print(task.result())

async_main()
```

Pretty neat!

The benefits immediately become apparent when we compare execution times:

```
print("Running synchronous example")
start = time()
main()
duration = time() - start

print("Running asynchronous example")
async_start = time()
async_main()
async_duration = time() - async_start

print("Synchronous example took {} seconds".format(duration))
print("Asynchronous example took {} seconds".format(async_duration))
```

This outputs the following on my trusty laptop:

```
Running synchronous example
https://www.google.com/       : <!doctype  in 0:00:00.798940s
https://www.yahoo.com/        : <?xml vers in 0:00:00.883083s
http://www.bbc.co.uk/         : <!DOCTYPE  in 0:00:01.479646s
https://en.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.172367s
https://de.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.166793s
https://news.ycombinator.com/ : <html op=" in 0:00:01.464526s
https://www.tagesschau.de/    : <!DOCTYPE  in 0:00:00.991291s
Running asynchronous example
https://www.google.com/       : <!doctype  in 0:00:00.618827s
http://www.bbc.co.uk/         : <!DOCTYPE  in 0:00:00.501347s
https://en.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.169479s
https://www.yahoo.com/        : <?xml vers in 0:00:00.762460s
https://news.ycombinator.com/ : <html op=" in 0:00:00.711696s
https://www.tagesschau.de/    : <!DOCTYPE  in 0:00:00.645607s
https://de.wikipedia.org/wiki/: <!DOCTYPE  in 0:00:00.167020s
Synchronous example took 12.025413990020752 seconds
Asynchronous example took 6.950876951217651 seconds
```

While the result will vary slightly depending on the network and server load it becomes clear that we can shave off a few seconds of execution time by not letting HTTP requests block us. In this run the execution time was halved!

3 Further reading

If you want to read more about Python's `asyncio` refer to the following sources:

3.1 Python Documentation

- PEP 0492 - Coroutines with async and await syntax
- asyncio - Asynchronous I/O, event loop, coroutines and tasks

3.2 Tutorials

- PYTHON ASYNCIO FROM THE INSIDE OUT
- Fast scraping in python with asyncio