Last updated January 18th, 2022

# Speeding Up Python with Concurrency, Parallelism, and asyncio

 Jace Medlin

Twitter Reddit Hacker News Facebook

*What are concurrency and parallelism, and how do they apply to Python?*

There are many reasons your applications can be slow. Sometimes this is due to poor algorithmic design or the wrong choice of data structure. Sometimes, however, it's due to forces outside of our control, such as hardware constraints or the quirks of networking. That's where concurrency and parallelism fit in. They allow your programs to do multiple things at once, either at the same time or by wasting the least possible time waiting on busy tasks.

Whether you're dealing with external web resources, reading from and writing to multiple files, or need to use a calculation-intensive function multiple times with different parameters, this article should help you maximize the efficiency and speed of your code.

First, we'll delve into what concurrency and parallelism are and how they fit into the realm of Python using standard libraries such as threading, multiprocessing, and asyncio. The last portion of this article will compare Python's implementation of `async` / `await` with how other languages have implemented them.

You can find all the code examples from this article in the concurrency-parallelism-and-asyncio repo on GitHub.

> To work through the examples in this article, you should already know how to work with HTTP requests.

## Objectives

By the end of this article, you should be able to answer the following questions:

1. What is concurrency?

Featured Course

Creating an HTTP Load Balancer in Python

In this course, you'll learn how to implement a load balancer in Python using Test-Driven Development.

Buy Now $15

View Course ›

Search all tutorials

TUTORIAL TOPICS

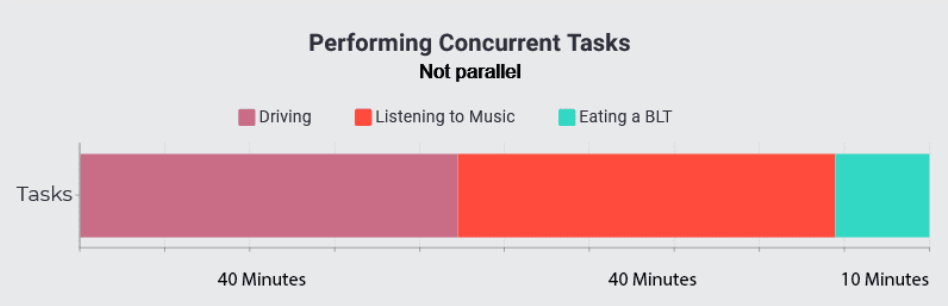
api architecture aws devops django django rest framework docker fastapi flask front-end heroku kubernetes machine learning python react task queue testing vue web scraping

2. What is a thread?
3. What does it mean when something is non-blocking?
4. What is an event loop?
5. What's a callback?
6. Why is the asyncio method always a bit faster than the threading method?
7. When should you use threading, and when should you use asyncio?
8. What is parallelism?
9. What's the difference between concurrency and parallelism?
10. Is it possible to combine asyncio with multiprocessing?
11. When should you use multiprocessing vs asyncio or threading?
12. What's the difference between multiprocessing, asyncio, and concurrency.futures?
13. How can do I test asyncio with pytest?

# Concurrency

*What is concurrency?*

An effective definition for concurrency is "being able to perform multiple tasks at once". This is a bit misleading though, as the tasks may or may not actually be performed at exactly the same time. Instead, a process might start, then once it's waiting on a specific instruction to finish, switch to a new task, only to come back once it's no longer waiting. Once one task is finished, it switches again to an unfinished task until they have all been performed. Tasks start asynchronously, get performed asynchronously, and then finish asynchronously.




If that was confusing to you, let's instead think of an analogy: Say you want to make a BLT. First, you'll want to throw the bacon in a pan on medium-low heat. **While** the bacon's cooking, you can get out your tomatoes and lettuce and start preparing (washing and cutting) them. All the while, you continue checking on and occasionally flipping over your bacon.

At this point, you've started a task, and then started and completed two more in the meantime, all while you're still waiting on the first.

Eventually you put your bread in a toaster. While it's toasting, you continue checking on your bacon. As pieces get finished, you pull them out and place them on a plate. Once your bread is done toasting, you apply to it your sandwich spread of choice, and then you can start layering on your tomatoes, lettuce, and then, once it's done cooking, your bacon. Only once everything is cooked, prepared, and layered can you place the last piece of toast onto your sandwich, slice it (optional), and eat it.

Because it requires you to perform multiple tasks at the same time, making a BLT is inherently a concurrent process, even if you are not giving your full attention to each of those tasks all at once. For all intents and purposes, for the next section, we'll refer to this form of concurrency as just "concurrency." We'll differentiate it later on in this article.

For this reason, concurrency is great for I/O-intensive processes -- tasks that involve waiting on web requests or file read/write operations.

In Python, there are a few different ways to achieve concurrency. The first we'll take a look at is the threading library.

> For our examples in this section, we're going to build a small Python program that grabs a random music genre from Binary Jazz's Genrenator API five times, prints the genre to the screen, and puts each one into its own file.

To work with threading in Python, the only import you'll need is `threading`, but for this example, I've also imported `urllib` to work with HTTP requests, `time` to determine how long the functions take to complete, and `json` to easily convert the json data returned from the Genrenator API.

> You can find the code for this example here.

Let's start with a simple function:

```
def write_genre(file_name):
    """
    Uses genrenator from binaryjazz.us to write a
random genre to the
    name of the given file
    """

    req = Request("https://binaryjazz.us/wp-
json/genrenator/v1/genre/", headers={"User-Agent":
"Mozilla/5.0"})
    genre = json.load(urlopen(req))

    with open(file_name, "w") as new_file:
        print(f"Writing '{genre}' to
'{file_name}'...")
        new_file.write(genre)
```

Examining the code above, we're making a request to the Genrenator API, loading its JSON response (a random music genre), printing it, then writing it to a file.

> Without the "User-Agent" header you will receive a 304.

What we're really interested in is the next section, where the actual threading happens:

```
threads = []

for i in range(5):
    thread = threading.Thread(
        target=write_genre,
        args=[f"./threading/new_file{i}.txt"]
    )
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

We first start with a list. We then proceed to iterate five times, creating a new thread each time. Next, we start each thread, append it to our "threads" list, and then iterate over our list one last time to join each thread.

Explanation: Creating threads in Python is easy.

To create a new thread, use `threading.Thread()`. You can pass into it the kwarg (keyword argument) `target` with a value of whatever function you would like to run on that thread. But only pass in the name of the function, not its value (meaning, for our purposes, `write_genre` and not `write_genre()`). To pass arguments, pass in "kwargs" (which takes a dict of your kwargs) or "args" (which takes an iterable containing your args -- in this case, a list).

Creating a thread is not the same as starting a thread, however. To start your thread, use `{the name of your thread}.start()`. Starting a thread means "starting its execution."

Lastly, when we join threads with `thread.join()`, all we're doing is ensuring the thread has finished before continuing on with our code.

## Threads

*But what exactly is a thread?*

A thread is a way of allowing your computer to break up a single process/program into many lightweight pieces that execute in parallel. Somewhat confusingly, Python's standard implementation of threading limits threads to only being able to execute one at a time due to something called the Global Interpreter Lock (GIL). The GIL is necessary because CPython's (Python's default implementation) memory management is not thread-safe. Because of this limitation, threading in Python is concurrent, but not parallel. To get around this, Python has a separate `multiprocessing` module not limited by the GIL that spins up separate processes, enabling parallel execution of your code. Using the `multiprocessing` module is nearly identical to using the `threading` module.

> More info about Python's GIL and thread safety can be found on Real Python and Python's official docs.

*We'll take a more in-depth look at multiprocessing in Python shortly.*

Before we show the potential speed improvement over non-threaded code, I took the liberty of also creating a non-threaded version of the same program (again, available on GitHub). Instead of creating a new thread and joining each one, it instead calls `write_genre` in a for loop that iterates five times.

To compare speed benchmarks, I also imported the `time` library to time the execution of our scripts:

```
Starting...
Writing "binary indoremix" to
"./sync/new_file0.txt"...
Writing "slavic aggro polka fusion" to
"./sync/new_file1.txt"...
Writing "israeli new wave" to
"./sync/new_file2.txt"...
Writing "byzantine motown" to
"./sync/new_file3.txt"...
Writing "dutch hate industrialtune" to
"./sync/new_file4.txt"...
Time to complete synchronous read/writes: 1.42
seconds
```

Upon running the script, we see that it takes my computer around 1.49 seconds (along with classic music genres such as "dutch hate industrialtune"). Not too bad.

Now let's run the version that uses threading:

```
Starting...
Writing "college k-dubstep" to
"./threading/new_file2.txt"...
Writing "swiss dirt" to
"./threading/new_file0.txt"...
Writing "bop idol alternative" to
"./threading/new_file4.txt"...
Writing "ethertrio" to "./threading/new_file1.txt"...
Writing "beach aust shanty français" to
"./threading/new_file3.txt"...
Time to complete threading read/writes: 0.77 seconds
```

The first thing that might stand out to you is the functions not being completed in order: 2 - 0 - 4 - 1 - 3

This is because of the asynchronous nature of threading: as one function waits, another one begins, and so on. Because we're able to continue performing tasks while we're waiting on others to finish (either due to networking or file I/O operations), you may also have noticed that we cut our time roughly in half: 0.77 seconds. Whereas this might not seem like a lot now, it's easy to imagine the very real case of building a web application that needs to write much more data to a file or interact with much more complex web services.

*So, if threading is so great, why don't we end the article here?*

Because there are even better ways to perform tasks concurrently.

# Asyncio

Let's take a look at an example using asyncio. For this method, we're going to install aiohttp using `pip`. This will allow us to make non-blocking requests and receive responses using the `async` / `await` syntax that will be introduced shortly. It also has the extra benefit of a function that converts a JSON response without needing to import the `json` library. We'll also install and import aiofiles, which allows non-blocking file operations. Other than `aiohttp` and `aiofiles`, import `asyncio`, which comes with the Python standard library.

*"Non-blocking" means a program will allow other threads to continue running while it's waiting. This is opposed to "blocking" code, which stops execution of your program completely. Normal, synchronous I/O operations suffer from this limitation.*

> You can find the code for this example here.

Once we have our imports in place, let's take a look at the asynchronous version of the `write_genre` function from our asyncio example:

```
async def write_genre(file_name):
    """
    Uses genrenator from binaryjazz.us to write a
random genre to the
    name of the given file
    """

    async with aiohttp.ClientSession() as session:
        async with
session.get("https://binaryjazz.us/wp-
json/genrenator/v1/genre/") as response:
            genre = await response.json()

    async with aiofiles.open(file_name, "w") as
new_file:
        print(f'Writing "{genre}" to "
{file_name}"...')
        await new_file.write(genre)
```

For those not familiar with the `async` / `await` syntax that can be found in many other modern languages, `async` declares that a function, `for` loop, or `with` statement **must** be used asynchronously. To call an async function, you must either use the `await` keyword from another async function or call `create_task()` directly from the event loop, which can be grabbed from `asyncio.get_event_loop()` -- i.e., `loop = asyncio.get_event_loop()`.

Additionally:

1. `async with` allows awaiting async responses and file operations.
2. `async for` (not used here) iterates over an asynchronous stream.

## The Event Loop

Event loops are constructs inherent to asynchronous programming that allow performing tasks asynchronously. As you're reading this article, I can safely assume you're probably not too familiar with the concept. However, even if you've never written an async application, you have experience with event loops every time you use a computer. Whether your computer is listening for keyboard input, you're playing online multiplayer games, or you're browsing Reddit while you have files copying in the background, an event loop is the driving force that keeps everything working smoothly and efficiently. In its purest essence, an event loop is a process that waits around for triggers and then performs specific (programmed) actions once those triggers are met. They often return a "promise" (JavaScript syntax) or "future" (Python syntax) of some sort to denote that a task has been added. Once the task is finished, the promise or future returns a value passed back from the called function (assuming the function does return a value).

The idea of performing a function in response to another function is called a "callback."

> For another take on callbacks and events, here's a great answer on Stack Overflow.

Here's a walkthrough of our function:

We're using `async with` to open our client session asynchronously. The `aiohttp.ClientSession()` class is what allows us to make HTTP requests and remain connected to a source without blocking the execution of our code. We then make an async request to the Genrenator API and await the JSON response (a random music genre). In the next line, we use `async with` again with the `aiofiles` library to asynchronously open a new file to write our new genre to. We print the genre, then write it to the file.

Unlike regular Python scripts, programming with asyncio pretty much enforces* using some sort of "main" function.

> *Unless you're using the deprecated "yield" syntax with the @asyncio.coroutine decorator, which will be removed in Python 3.10.




Featured Course

Creating an HTTP Load Balancer in Python

Buy Now $15

View Course >

This is because you need to use the "async" keyword in order to use the "await" syntax, and the "await" syntax is the only way to actually run other async functions.

Here's our main function:

```
async def main():
    tasks = []

    for i in range(5):

tasks.append(write_genre(f"./async/new_file{i}.txt")
)

    await asyncio.gather(*tasks)
```

As you can see, we've declared it with "async." We then create an empty list called "tasks" to house our async tasks (calls to Genrenator and our file I/O). We append our tasks to our list, but they are *not* actually run yet. The calls don't actually get made until we schedule them with `await asyncio.gather(*tasks)`. This runs all of the tasks in our list and waits for them to finish before continuing with the rest of our program. Lastly, we use `asyncio.run(main())` to run our "main" function. The `.run()` function is the entry point for our program, and it should *generally* only be called once *per process*.

> For those not familiar, the `*` in front of tasks is called "argument unpacking." Just as it sounds, it unpacks our list into a series of arguments for our function. Our function is `asyncio.gather()` in this case.

And that's all we need to do. Now, running our program (the source of which includes the same timing functionality of the synchronous and threading examples)...

```
Writing "albuquerque fiddlehaus" to
"./async/new_file1.txt"...
Writing "euroreggaebop" to "./async/new_file2.txt"...
Writing "shoedisco" to "./async/new_file0.txt"...
Writing "russiagaze" to "./async/new_file4.txt"...
Writing "alternative xylophone" to
"./async/new_file3.txt"...
Time to complete asyncio read/writes: 0.71 seconds
```

...we see it's even faster still. And, in general, the asyncio method will always be a bit faster than the threading method. This is because when we use the "await" syntax, we essentially tell our program "hold on, I'll be right

back," but our program keeps track of how long it takes us to finish what we're doing. Once we're done, our program will know, and will pick back up as soon as it's able. Threading in Python allows asynchronicity, but our program could theoretically skip around different threads that may not yet be ready, wasting time if there are threads ready to continue running.

*So when should I use threading, and when should I use asyncio?*

When you're writing new code, use asyncio. If you need to interface with older libraries or those that don't support asyncio, you might be better off with threading.

## Testing asyncio with pytest

It turns out testing async functions with pytest is as easy as testing synchronous functions. Just install the pytest-asyncio package with `pip`, mark your tests with the `async` keyword, and apply a decorator that lets `pytest` know it's asynchronous: `@pytest.mark.asyncio`. Let's look at an example.

First, let's write an arbitrary async function in a file called *hello_asyncio.py*:

```
import asyncio


async def say_hello(name: str):
    """ Sleeps for two seconds, then prints 'Hello,
{{ name }}!' """
    try:
        if type(name) != str:
            raise TypeError("'name' must be a
string")
        if name == "":
            raise ValueError("'name' cannot be
empty")
    except (TypeError, ValueError):
        raise

    print("Sleeping...")
    await asyncio.sleep(2)
    print(f"Hello, {name}!")
```

The function takes a single string argument: `name`. Upon ensuring that `name` is a string with a length greater than one, our function asynchronously sleeps for two seconds, then prints `"Hello, {name}!"` to the console.

> The difference between `asyncio.sleep()` and `time.sleep()` is that `asyncio.sleep()` is non-blocking.

Now let's test it with pytest. In the same directory as *hello_asyncio.py*, create a file called *test_hello_asyncio.py*, then open it in your favorite text editor.

Let's start with our imports:

```
import pytest # Note: pytest-asyncio does not
require a separate import

from hello_asyncio import say_hello
```

Then we'll create a test with proper input:

```
@pytest.mark.parametrize("name", [
    "Robert Paulson",
    "Seven of Nine",
    "x Æ a-12"
])
@pytest.mark.asyncio
async def test_say_hello(name):
    await say_hello(name)
```

Things to note:

- The `@pytest.mark.asyncio` decorator lets pytest work asynchronously
- Our test uses the `async` syntax
- We're `await`ing our async function as we would if we were running it outside of a test

Now let's run our test with the verbose `-v` option:

```
pytest -v
...
collected 3 items

test_hello_asyncio.py::test_say_hello[Robert Paulson]
PASSED    [ 33%]
test_hello_asyncio.py::test_say_hello[Seven of Nine]
PASSED     [ 66%]
test_hello_asyncio.py::test_say_hello[x \xc6 a-12]
PASSED       [100%]
```

Looks good. Next we'll write a couple of tests with bad input. Back inside of *test_hello_asyncio.py*, let's create a class called `TestSayHelloThrowsExceptions`:

```
class TestSayHelloThrowsExceptions:
    @pytest.mark.parametrize("name", [
        "",
    ])
    @pytest.mark.asyncio
    async def test_say_hello_value_error(self,
name):
        with pytest.raises(ValueError):
            await say_hello(name)

    @pytest.mark.parametrize("name", [
        19,
        {"name", "Diane"},
        []
    ])
    @pytest.mark.asyncio
    async def test_say_hello_type_error(self, name):
        with pytest.raises(TypeError):
            await say_hello(name)
```

Again, we decorate our tests with `@pytest.mark.asyncio`, mark our tests with the `async` syntax, then call our function with `await`.

Run the tests again:

```
pytest -v
...
collected 7 items

test_hello_asyncio.py::test_say_hello[Robert Paulson]
PASSED                                      [ 14%]
test_hello_asyncio.py::test_say_hello[Seven of Nine]
PASSED                                       [ 28%]
test_hello_asyncio.py::test_say_hello[x \xc6 a-12]
PASSED                                         [ 42%]
test_hello_asyncio.py::TestSayHelloThrowsExceptions::
test_say_hello_value_error[] PASSED          [ 57%]
test_hello_asyncio.py::TestSayHelloThrowsExceptions::
test_say_hello_type_error[19] PASSED         [ 71%]
test_hello_asyncio.py::TestSayHelloThrowsExceptions::
test_say_hello_type_error[name1] PASSED      [ 85%]
test_hello_asyncio.py::TestSayHelloThrowsExceptions::
test_say_hello_type_error[name2] PASSED      [100%]
```

### Without pytest-asyncio

Alternatively to pytest-asyncio, you can create a pytest fixture that yields an asyncio event loop:

```
import asyncio
import pytest

from hello_asyncio import say_hello


@pytest.fixture
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
```

Then, rather than using the `async` / `await` syntax, you create your tests as you would normal, synchronous tests:

```
@pytest.mark.parametrize("name", [
    "Robert Paulson",
    "Seven of Nine",
    "x Æ a-12"
])
def test_say_hello(event_loop, name):
    event_loop.run_until_complete(say_hello(name))


class TestSayHelloThrowsExceptions:
    @pytest.mark.parametrize("name", [
        "",
    ])
    def test_say_hello_value_error(self, event_loop,
name):
        with pytest.raises(ValueError):

event_loop.run_until_complete(say_hello(name))

    @pytest.mark.parametrize("name", [
        19,
        {"name", "Diane"},
        []
    ])
    def test_say_hello_type_error(self, event_loop,
name):
        with pytest.raises(TypeError):

event_loop.run_until_complete(say_hello(name))
```

> If you're interested, here's a more advanced tutorial on asyncio testing.

### Further Reading

If you want to learn more about what distinguishes Python's implementation of threading vs asyncio, here's a great article from Medium.

For even better examples and explanations of threading in Python, here's a video by Corey Schafer that goes more in-depth, including using the `concurrent.futures` library.

Lastly, for a massive deep-dive into asyncio itself, here's an article from Real Python completely dedicated to it.

*Bonus*: One more library you might be interested in is called Unsync, especially if you want to easily convert your current synchronous code into asynchronous code. To use it, you install the library with pip, import it with `from unsync import unsync`, then decorate whatever currently synchronous function with `@unsync` to make it asynchronous. To await it and get its return value (which you can do anywhere -- it doesn't have to be in an async/unsync function), just call `.result()` after the function call.

## Parallelism

*What is parallelism?*

Parallelism is very-much related to concurrency. In fact, parallelism is a subset of concurrency: whereas a concurrent process performs multiple tasks at the same time whether they're being diverted total attention or not, a parallel process is physically performing multiple tasks all at the same time. A good example would be driving, listening to music, and eating the BLT we made in the last section at the same time.

Because they don't require a lot of intensive effort, you can do them all at once without having to wait on anything or divert your attention away.

Now let's take a look at how to implement this in Python. We could use the `multiprocessing` library, but let's use the `concurrent.futures` library instead -- it eliminates the need to manage the number of process manually. Because the major benefit of multiprocessing happens when you perform multiple cpu-heavy tasks, we're going to compute the squares of 1 million (1000000) to 1 million and 16 (1000016).

> You can find the code for this example here.

The only import we'll need is `concurrent.futures`:

```
import concurrent.futures
import time


if __name__ == "__main__":
    pow_list = [i for i in range(1000000, 1000016)]

    print("Starting...")
    start = time.time()

    with concurrent.futures.ProcessPoolExecutor() as
executor:
        futures = [executor.submit(pow, i, i) for i
in pow_list]

    for f in
concurrent.futures.as_completed(futures):
        print("okay")

    end = time.time()
    print(f"Time to complete: {round(end - start,
2)}")
```

> Because I'm developing on a Windows machine, I'm using `if __name__ == "main"`. This is necessary because Windows does not have the `fork` system call inherent to Unix systems. Because Windows doesn't have this capability, it resorts to launching a new interpreter with each process that tries to import the main module. If the main module doesn't exist, it reruns your entire program, causing recursive chaos to ensue.

So taking a look at our main function, we use a list comprehension to create a list from 1 million to 1 million and 16, we open a ProcessPoolExecutor with concurrent.futures, and we use list comprehension and `ProcessPoolExecutor().submit()` to start executing our processes and throwing them into a list called "futures."

> We could also use `ThreadPoolExecutor()` if we wanted to use threads instead -- concurrent.futures is versatile.

And this is where the asynchronicity comes in: The "results" list does not actually contain the results from running our functions. Instead, it contains "futures" which are similar to the JavaScript idea of "promises." In order to allow our program to continue running, we get back these futures that represent a placeholder for a value. If we try to print the future, depending on whether it's finished running or not, we'll either get back a state of "pending" or "finished." Once it's finished we can get the return value (assuming there is one) using `var.result()`. In this case, our var will be "result."

We then iterate through our list of futures, but instead of printing our values, we're simply printing out "okay." This is just because of how massive the resulting calculations come out to be.

Just as before, I built a comparison script that does this synchronously. And, just as before, you can find it on GitHub.

Running our control program, which also includes functionality for timing our program, we get:

```
Starting...
okay
...
okay
Time to complete: 54.64
```

Wow. 54.64 seconds is quite a long time. Let's see if our version with multiprocessing does any better:

```
Starting...
okay
...
okay
Time to complete: 6.24
```

Our time has been *significantly* reduced. We're at about 1/9th of our original time.

*So what would happen if we used threading for this instead?*

I'm sure you can guess -- it wouldn't be much faster than doing it synchronously. In fact, it might be slower because it still takes a little time and effort to spin up new threads. But don't take my word for it, here's what we get when we replace `ProcessPoolExecutor()` with `ThreadPoolExecutor()`:

```
Starting...
okay
...
okay
Time to complete: 53.83
```

As I mentioned earlier, threading allows your applications to focus on new tasks while others are waiting. In this case, we're never sitting idly by. Multiprocessing, on the other hand, spins up totally new services, usually on separate CPU cores, ready to do whatever you ask it completely in tandem with whatever else your script is doing. This is why the multiprocessing version taking roughly 1/9th of the time makes sense -- I have 8 cores in my CPU.

Now that we've talked about concurrency and parallelism in Python, we can finally set the terms straight. If you're having trouble distinguishing between the terms, you can safely and accurately think of our previous definitions of "parallelism" and "concurrency" as "parallel concurrency" and "non-parallel concurrency" respectively.

### Further Reading

Real Python has a great article on concurrency vs parallelism.

Engineer Man has a good video comparison of threading vs multiprocessing.

Corey Schafer also has a good video on multiprocessing in the same spirit as his threading video.

If you only watch one video, watch this excellent talk by Raymond Hettinger. He does an amazing job explaining the differences between multiprocessing, threading, and asyncio.

## Combining Asyncio with Multiprocessing

*What if I need to combine many I/O operations with heavy calculations?*

We can do that too. Say you need to scrape 100 web pages for a specific piece of information, and then you need to save that piece of info in a file for later. We can separate the compute power across each of our computer's cores by making each process scrape a fraction of the pages.

For this script, let's install Beautiful Soup to help us easily scrape our pages: `pip install beautifulsoup4`. This time we actually have quite a few imports. Here they are, and here's why we're using them:

```
import asyncio                          # Gives us
async/await
import concurrent.futures               # Allows
creating new processes
import time
from math import floor                  # Helps
divide up our requests evenly across our CPU cores
from multiprocessing import cpu_count   # Returns
our number of CPU cores

import aiofiles                         # For
asynchronously performing file I/O operations
import aiohttp                          # For
asynchronously making HTTP requests
from bs4 import BeautifulSoup           # For easy
webpage scraping
```

> You can find the code for this example here.

First, we're going to create an async function that makes a request to Wikipedia to get back random pages. We'll scrape each page we get back for its title using `BeautifulSoup`, and then we'll append it to a given file; we'll separate each title with a tab. The function will take two arguments:

1. num_pages - Number of pages to request and scrape for titles
2. output_file - The file to append our titles to

```
async def get_and_scrape_pages(num_pages: int,
output_file: str):
    """
    Makes {{ num_pages }} requests to Wikipedia to
receive {{ num_pages }} random
    articles, then scrapes each page for its title
and appends it to {{ output_file }},
    separating each title with a tab: "\\t"

    #### Arguments
    ---
    num_pages: int -
        Number of random Wikipedia pages to request
and scrape

    output_file: str -
        File to append titles to
    """
    async with \
    aiohttp.ClientSession() as client, \
    aiofiles.open(output_file, "a+", encoding="utf-
8") as f:

        for _ in range(num_pages):
            async with
client.get("https://en.wikipedia.org/wiki/Special:Ran
dom") as response:
                if response.status > 399:
                    # I was getting a 429 Too Many
Requests at a higher volume of requests
                    response.raise_for_status()

                page = await response.text()
                soup = BeautifulSoup(page,
features="html.parser")
                title = soup.find("h1").text

                await f.write(title + "\t")

        await f.write("\n")
```

We're both asynchronously opening an aiohttp `ClientSession` and our output file. The mode, `a+`, means append to the file and create it if it doesn't already exist. Encoding our strings as utf-8 ensures we don't get an error if our titles contain international characters. If we get an error response, we'll raise it instead of continuing (at high request volumes I was

getting a 429 Too Many Requests). We asynchronously get the text from our response, then we parse the title and asynchronously and append it to our file. After we append all of our titles, we append a new line: "\n".

Our next function is the function we'll start with each new process to allow running it asynchronously:

```
def start_scraping(num_pages: int, output_file: str,
i: int):
    """ Starts an async process for requesting and
scraping Wikipedia pages """
    print(f"Process {i} starting...")
    asyncio.run(get_and_scrape_pages(num_pages,
output_file))
    print(f"Process {i} finished.")
```

Now for our main function. Let's start with some constants (and our function declaration):

```
def main():
    NUM_PAGES = 100 # Number of pages to scrape
altogether
    NUM_CORES = cpu_count() # Our number of CPU cores
(including logical cores)
    OUTPUT_FILE = "./wiki_titles.tsv" # File to
append our scraped titles to

    PAGES_PER_CORE = floor(NUM_PAGES / NUM_CORES)
    PAGES_FOR_FINAL_CORE = PAGES_PER_CORE + NUM_PAGES
% PAGES_PER_CORE # For our final core
```

And now the logic:

```
    futures = []

    with
concurrent.futures.ProcessPoolExecutor(NUM_CORES) as
executor:
        for i in range(NUM_CORES - 1):
            new_future = executor.submit(
                start_scraping, # Function to perform
                # v Arguments v
                num_pages=PAGES_PER_CORE,
                output_file=OUTPUT_FILE,
                i=i
            )
            futures.append(new_future)

        futures.append(
            executor.submit(
                start_scraping,
                PAGES_FOR_FINAL_CORE, OUTPUT_FILE,
NUM_CORES-1
            )
        )

    concurrent.futures.wait(futures)
```

We create an array to store our futures, then we create a `ProcessPoolExecutor`, setting its `max_workers` equal to our number of cores. We iterate over a range equal to our number of cores minus 1, running a new process with our `start_scraping` function. We then append it our futures list. Our final core will potentially have extra work to do as it will scrape a number of pages equal to each of our other cores, but will additionally scrape a number of pages equal to the remainder that we got when dividing our total number of pages to scrape by our total number of cpu cores.

Make sure to actually run your main function:

```
if __name__ == "__main__":
    start = time.time()
    main()
    print(f"Time to complete: {round(time.time() -
start, 2)} seconds.")
```

After running the program with my 8-core CPU (along with benchmarking code):

This version (asyncio with multiprocessing):

```
Time to complete: 5.65 seconds.
```

Multiprocessing only:

```
Time to complete: 8.87 seconds.
```

asyncio only:

```
Time to complete: 47.92 seconds.
```

Completely synchronous:

```
Time to complete: 88.86 seconds.
```

I'm actually quite surprised to see that the improvement of asyncio with multiprocessing over just multiprocessing wasn't as great as I thought it would be.

## Recap: When to use multiprocessing vs asyncio or threading

1. Use multiprocessing when you need to do many heavy calculations and you can split them up.
2. Use asyncio or threading when you're performing I/O operations -- communicating with external resources or reading/writing from/to files.
3. Multiprocessing and asyncio can be used together, but a good rule of thumb is to fork a process before you thread/use asyncio instead of the other way around -- threads are relatively cheap compared to processes.

## Async/Await in Other Languages

`async` / `await` and similar syntax also exist in other languages, and in some of those languages, its implementation can differ drastically.

### .NET: F# to C

The first programming language (back in 2007) to use the `async` syntax was Microsoft's F#. Whereas it doesn't exactly use `await` to wait on a function call, it uses specific syntax like `let!` and `do!` along with proprietary `Async` functions included in the `System` module.

> You can find more about async programming in F# on Microsoft's F# docs.

Their C# team then built upon this concept, and that's where the `async` / `await` keywords that we're now familiar with were born:

```
using System;

// Allows the "Task" return type
using System.Threading.Tasks;

public class Program
{
    // Declare an async function with "async"
    private static async Task<string> ReturnHello()
    {
        return "hello world";
    }

    // Main can be async -- no problem
    public static async Task Main()
    {
        // await an async string
        string result = await ReturnHello();

        // Print the string we got asynchronously
        Console.WriteLine(result);
    }
}
```

*Run it on .NETFiddle*

We ensure that we're `using System.Threading.Tasks` as it includes the `Task` type, and, in general, the `Task` type is needed for an async function to be awaited. The cool thing about C# is that you can make your main function asynchronous just by declaring it with `async`, and you won't have any issues.

> If you're interested in learning more about `async` / `await` in C#, Microsoft's C# docs have a good page on it.

## JavaScript

First introduced in ES6, the `async` / `await` syntax is essentially an abstraction over JavaScript promises (which are similar to Python futures). Unlike Python, however, so long as you're not awaiting, you can call an

async function normally without a specific function like Python's `asyncio.start()`:

```
// Declare a function with async
async function returnHello(){
    return "hello world";
}

async function printSomething(){
    // await an async string
    const result = await returnHello();

    // print the string we got asynchronously
    console.log(result);
}

// Run our async code
printSomething();
```

*Run it on JSFiddle*

> See MDN for more info on `async` / `await` in JavaScript.

## Rust

Rust now allows the use of the `async` / `await` syntax as well, and it works similarly to Python, C#, and JavaScript:

```rust
// Allows blocking synchronous code to run async code
use futures::executor::block_on;

// Declare an async function with "async"
async fn return_hello() -> String {
    "hello world".to_string()
}

// Code that awaits must also be declared with
"async"
async fn print_something(){
    // await an async String
    let result: String = return_hello().await;

    // Print the string we got asynchronously
    println!("{0}", result);
}

fn main() {
    // Block the current synchronous execution to run
our async code
    block_on(print_something());
}
```

[Run it on Rust Play]

In order to use async functions, we must first add `futures = "0.3"` to our *Cargo.toml*. We then import the `block_on` function with `use futures::executor::block_on` -- `block_on` is necessary for running our async function from our synchronous `main` function.

> You can find more info on `async` / `await` in Rust in the Rust docs.

## Go

Rather than the traditional `async` / `await` syntax inherent to all of the previous languages we've covered, Go uses "goroutines" and "channels." You can think of a channel as being similar to a Python future. In Go, you generally send a channel as an argument to a function, then use `go` to run the function concurrently. Whenever you need to ensure the function has finished completing, you use the `<-` syntax, which you can think of as the more common `await` syntax. If your goroutine (the function you're running asynchronously) has a return value, it can be grabbed this way.

```go
package main

import "fmt"

// "chan" makes the return value a string channel
instead of a string
func returnHello(result chan string){
    // Gives our channel a value
    result <- "hello world"
}

func main() {
    // Creates a string channel
    result := make(chan string)

    // Starts execution of our goroutine
    go returnHello(result)

    // Awaits and prints our string
    fmt.Println(<- result)
}
```

*Run it in the Go Playground*

> For more info on concurrency in Go, see An Introduction to Programming in Go by Caleb Doxsey.

### Ruby

Similarly to Python, Ruby also has the Global Interpreter Lock limitation. What it doesn't have is concurrency built-in to the language. However, there is a community-created gem that allows concurrency in Ruby, and you can find its source on GitHub.

### Java

Like Ruby, Java doesn't have the `async` / `await` syntax built-in, but it does have concurrency capabilities using the `java.util.concurrent` module. However, Electronic Arts wrote an Async library that allows using `await` as a method. It's not exactly the same as Python/C#/JavaScript/Rust, but it's worth looking into if you're a Java developer and are interested in this sort of functionality.

### C++

Although C++ also doesn't have the `async` / `await` syntax, it does have the ability to use futures to run code concurrently using the `futures` module:

```
#include <iostream>
#include <string>

// Necessary for futures
#include <future>

// No async declaration needed
std::string return_hello() {
    return "hello world";
}

int main ()
{
    // Declares a string future
    std::future<std::string> fut =
std::async(return_hello);

    // Awaits the result of the future
    std::string result = fut.get();

    // Prints the string we got asynchronously
    std::cout << result << '\n';
}
```

*Run it on C++ Shell*

There's no need to declare a function with any keyword to denote whether or not it can and should be run asynchronously. Instead, you declare your initial future whenever you need it with `std::future<{{ function return type }}>` and set it equal to `std::async()`, including the name of the function you want to perform asynchronously along with any arguments it takes -- i.e., `std::async(do_something, 1, 2, "string")`. To await the value of the future, use the `.get()` syntax on it.

> You can find documentation for async in C++ on cplusplus.com.

## Summary

Whether you're working with asynchronous network or file operations or you're performing numerous complex calculations, there are a few different ways to maximize your code's efficiency.

If you're using Python, you can use `asyncio` or `threading` to make the most out of I/O operations or the `multiprocessing` module for CPU-intensive code.

> Also remember that the `concurrent.futures` module can be used in place of either `threading` or `multiprocessing`.

If you're using another programming language, chances are there's an implementation of `async` / `await` for it too.

> Want to see more examples of parallelism, concurrency, and asyncio? Check out the Parallelism, Concurrency, and AsyncIO in Python - by example article.

python

## Jace Medlin

Jace is a US-based software engineer with interests in Python, Vue, and Rust. When he's not building web apps across the stack, he can be found recording music, collecting VHS tapes, philosophizing about the universe, and experimenting with robotics.

### SHARE THIS TUTORIAL

Twitter Reddit Hacker News Facebook

▶ **Revision History**

### RECOMMENDED TUTORIALS

## Parallelism, Concurrency, and AsyncIO in Python - by example

 Amal Shaji

Jul 5th, 2022

This tutorial looks at how to speed up CPU-bound and IO-bound operations with multiprocessing, threading, and AsyncIO.

python

## Developing an Asynchronous Task Queue in Python

 Michael Herman

Jun 21st, 2023

How to implement several asynchronous task queues using the Python multiprocessing library and Redis.

python task queue

## Concurrent Web Scraping with Selenium Grid and Docker Swarm

 Michael Herman

Mar 31st, 2022

Run a Python and Selenium-based web scraper in parallel with Selenium Grid and Docker Swarm.

devops docker web scraping

## Stay Sharp with Course Updates

Join our mailing list to be notified about updates and new releases.

Enter your email Subscribe



TestDriven.io is a proud supporter of open source

**10% of profits** from each of our FastAPI courses and our Flask Web Development course will be donated to the FastAPI and Flask teams, respectively.

Follow our contributions ›

© Copyright 2017 - 2024 TestDriven Labs.
Developed by Michael Herman.

testdriven.io

Follow @testdrivenio