Threaded Asynchronous Magic and How to Wield It

Originally published by Cristian Medina on October 4th 2016

@tryexceptpass

Cristian Medina

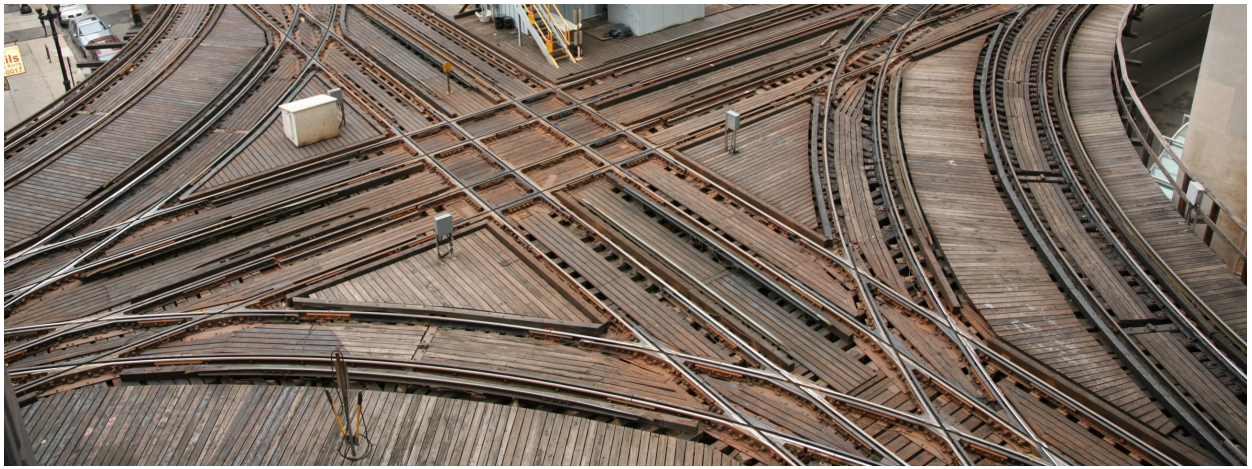


Photo Credit: Daniel Schwen via Wikipedia

A dive into Python's asyncio tasks and event loops

Ok let's face it. Clock speeds no longer govern the pace at which computer processors improve. Instead we see increased transistor density and higher

core counts. Translating to software terms, this means that code won't run faster, but more of it can run in parallel.

Although making good use of our new-found silicon real estate requires

improvements in software, a lot of programming languages have already started down this path by adding features that help with parallel execution. In fact, they've been there for years waiting for us to take advantage.

So why don't we? A good engineer always has an ear to the ground, listening for the latest trends in his industry, so let's take a look at what Python is building for us.

What do we have so far?

Python enables parallelism through both the threading and the multiprocessing libraries. Yet it wasn't until the 3.4 branch that it gave us the *asyncio* library to help with single-threaded concurrency. This addition was key in providing a more convincing final push to start swapping over from version 2.

The asyncio package allows us to define coroutines. These are code blocks that have the ability of yielding execution to other blocks. They run inside an *event loop* which iterates through the scheduled tasks and executes them one by one. A task switch occurs when it reaches an `await` statement or when

the current task completes.

Task execution itself happens the same as in a single-threaded system. Meaning, this is not an implementation of parallelism, it's actually closer to

multithreading. We can perceive the concurrency in situations where a block of code depends on external actions.

This illusion is possible because the block can yield execution while it waits, making anything that depends on external IO, like network or disk storage, a great candidate. When the IO completes, the coroutine receives an interrupt and can proceed with execution. In the meantime, other tasks execute.

The asyncio event loop can also serve as a task scheduler. Both asynchronous and blocking functions can queue up their execution as needed.

Tasks

A `Task` represents callable blocks of code designed for asynchronous execution within event loops. They execute single-threaded, but can run in parallel through loops on different threads.

Prefixing a function definition with the `async` keyword turns it into an asynchronous coroutine. Though the task itself will not exist until it's added to

a loop. This is usually implicit when calling most loop methods, but `asyncio.ensure_future(your_coroutine)` is the more direct mechanism.

To denote an operation or instruction that can yield execution, we use the `await` keyword. Although it's only available within a coroutine block and

causes a syntax error if used anywhere else.

Please note that the async keyword was not implemented until Python version 3.5. So when working with older versions, use the `@asyncio.coroutine` decorator and `yield from` keywords instead.

Scheduling

In order to execute a task, we need a reference to the event loop in which to run it. Using `loop = asyncio.get_event_loop()` gives us the current loop in our execution thread. Now it's a matter of calling `loop.run_until_complete(*your_coroutine*)` or `loop.run_forever()` to have it do some work.

Let's look at a short example to illustrate a few points. I strongly encourage you to open an interpreter and follow along:

```
import time
import asyncio
```

```
async def do_some_work(x):
    print("Waiting " + str(x))
    await asyncio.sleep(x)
```

```
loop = asyncio.get_event_loop()
loop.run_until_complete(do_some_work(5))
```

Here we defined `do_some_work()` as a coroutine that waits on the results of external workload. The workload is simulated through `asyncio.sleep`.

Running the code may be surprising. Did you expect `run_until_complete` to be a *blocking* call? Remember that we're using the event loop from the **current thread** to execute the task. We'll discuss alternatives in more detail later. So for now, the important part is to understand that while execution blocks, the `await` keyword still enables concurrency.

For a better picture, let's change our test code a bit and look at executing tasks in batches:

```
tasks = [asyncio.ensure_future(do_some_work(2)),
         asyncio.ensure_future(do_some_work(5))]
```

```
loop.run_until_complete(asyncio.gather(*tasks))
```

Introducing the `asyncio.gather()` function enables results aggregation. It waits for several tasks in the same thread to complete and puts the results in a list.

The main observation here is that both function calls did not execute in sequence. It did not wait 2 seconds, then 5, for a total of 7 seconds. Instead it *started* to wait 2s, then moved on to the next item which *started* to wait 5s, returning when the longer task completed, for a total of 5s. Feel free to add more print statements to the base function if it helps visualize.

This means that we can put long running tasks with *awaitable* code in an execution batch, then ask Python to run them in parallel and wait until they all complete. If you plan it right, this will be faster than running in sequence.

complete. If you plan it right, this will be faster than running in sequence.

Think of it as an alternative to the `threading` package where after spinning up a number of `Threads`, we wait for them to complete with `.join()`. The major difference is that there's less overhead incurred than creating a new thread for each function.

Of course, it's always good to point out that your millage may vary based on the task at hand. If you're doing compute-heavy work, with little or no time waiting, then the only benefit you get is the grouping of code into logical batches.

Running a loop in a different thread

What if instead of doing everything in the current thread, we spawn a separate Thread to do the work for us.

```
from threading import Thread
import asyncio
```

```
def start_loop(loop):
```

```
    asyncio.set_event_loop(loop)
    loop.run_forever()
```

```
new_loop = asyncio.new_event_loop()
t = Thread(target=start_loop, args=(new_loop,))
t.start()
```

Notice that this time we created a new event loop through `asyncio.new_event_loop()`. The idea is to spawn a new thread, pass it that new loop and then call thread-safe functions (discussed later) to schedule work.

The advantage of this method is that work executed by the other event loop will not block execution in the current thread. Thereby allowing the main thread to manage the work, and enabling a new category of execution mechanisms.

Queuing work in a different thread

Using the thread and event loop from the previous code block, we can easily get work done with the `call_soon()`,`call_later()` or

get work done with the `call_soon()`, `call_later()` & `call_at()` methods. They are able to run regular function code blocks (those not defined as coroutines) in an event loop.

However, it's best to use their `_threadsafe` alternatives. Let's see how that looks:

```
import time
```

```
def more_work(x):
  print("More work %s" % x)
  time.sleep(x)
  print("Finished more work %s" % x)
```

```
new_loop.call_soon_threadsafe(more_work, 6)
new_loop.call_soon_threadsafe(more_work, 3)
```

Now we're talking! Executing this code does not block the main interpreter,

allowing us to give it more work. Since the work executes in order, we now essentially have a task queue.

We just went to multi-threaded execution of single-threaded code, but isn't concurrency part of what we get with asyncio? Sure it is! That loop on the worker thread is still async, so let's enable parallelism by giving it awaitable coroutines.

Doing so is a matter of using `asyncio.run_coroutine_threadsafe()`, as seen bellow:

```
new_loop.call_soon_threadsafe(more_work, 20)
asyncio.run_coroutine_threadsafe(do_some_work(5), new_loop)
asyncio.run_coroutine_threadsafe(do_some_work(10), new_loop)
```

These instructions illustrate how python is going about execution. The first call to `more_work` blocks for 20 seconds, while the calls to `do_some_work` execute in parallel immediately after `more_work` finishes.

Real World Example #1—Sending Notifications

A common situation these days is to send notifications as a result of a task or

event. This is usually simple, but talking to an email server to submit a new message can take time, and so can crafting the email itself.

There are many scenarios where we don't have the luxury of waiting around for tasks to complete. Where doing so provides no benefit to the end user. A prime example being a request for a password reset, or a webhook event that triggers repository builds and emails the results.

The recommended practice so far has been to use a task queuing system like `celery`, on top of a message queue server like `rabbitmq` to schedule the work. I'm here to tell you that for small things that can easily execute from another thread of your main application, it's not a bad idea to just use asyncio. The pattern being fairly similar to the code examples we've seen so far:

```
import asyncio
import smtplib
from threading import Thread
```

```
def send_notification(email):
    """Generate and send the notification email"""
```

```
# Do some work to get email body
message = ...

# Connect to the server
server = smtplib.SMTP("smtp.gmail.com:587")
server.ehlo()


server.starttls()
server.login(username, password)
```

```
# Send the email
server.sendmail(from_addr, email, message)
```

```
server.quit()
```

```
def start_email_worker(loop):
    """Switch to new event loop and run forever"""

    asyncio.set_event_loop(loop)
    loop.run_forever()
```

```
# Create the new loop and worker thread
worker_loop = asyncio.new_event_loop()

worker = Thread(target=star_email_worker, args=(worker_loop,))
```

```
# Start the thread
worker.start()
```

```
# Assume a Flask restful interface endpoint
@app.route("/notify")
def notify(email):
    """Request notification email"""
```

```
worker_loop.call_soon_threadsafe(send_notification, email)
```

Here we assume a Flask web API with an endpoint mounted at `/notify` in which to request a notification email of some sort.

Notice that `send_notification` is not a coroutine, so each email will be a

blocking call. The worker thread's event loop will serve as the queue in which to track the outgoing emails.

Why are the SMTP calls synchronous you wonder? Well, while this is a good example of what *should* be awaitable IO, I'm not aware of an asynchronous SMTP library at the moment. Feel free to substitute with an `async def`, `await` and `run_coroutine_threadsafe`, if you do find one.

Real World Example #2—Parallel Web Requests

Here's an example of batching HTTP requests that run concurrently to several servers, while waiting for responses before processing. I expect it to be useful for those of you that do a lot of scraping, as well as a quick intro to the `aiohttp` module.

`aiohttp` module.

```
import asyncio
import aiohttp
```

```
async def fetch(url):
    """Perform an HTTP GET to the URL and print the response"""
```

```
    response = await aiohttp.request('GET', url)
    return await response.text()
```

```
# Get a reference to the event loop
loop = asyncio.get_event_loop()
```

```
# Create the batch of requests we wish to execute
requests = [asyncio.ensure_future(fetch("https://github.com")),
            asyncio.ensure_future(fetch("https://google.com"))]
```

```
# Run the batch
responses = loop.run_until_complete(asyncio.gather(*requests))
```

```
# Examine responses
for resp in responses:
  print(resp)
```

Fairly straightforward, it's a matter of grouping the work in a list of tasks and using `run_until_complete` to get the responses back. This can easily change to use a separate thread in which to make requests, where it would be simple to add all the URLs through the thread-safe methods described previously.

I want to note that the `requests` library has asynchronous support through `gevent`, but I haven't done the work to figure out how that can tie into

`asyncio`. In contrast, I'm not aware of `asyncio` plans for the popular scraping framework `scrapy`, but I assume they're working on it.

Stopping the loop

If at any point you find yourself wanting to stop an infinite event loop, or want to cancel tasks that haven't completed, I tend to use a `KeyboardInterrupt` exception clause to trigger cancellation as shown below. Although the same can be accomplished by using the `signal` module and registering a handler for `signal.SIGINT`.

```
try:
    loop.run_forever()
```

```
except KeyboardInterrupt:
    # Canceling pending tasks and stopping the loop
    asyncio.gather(*asyncio.Task.all_tasks()).cancel()
```

```
# Stopping the loop
loop.stop()
```

```
# Received Ctrl+C
loop.close()
```

This time we're introducing the use of `Task.all_tasks()` to generate a list

of all currently running or scheduled tasks. When coupled with `gather()` we can send the `cancel()` command to each one and have them all stop executing or remove them from the queue.

Please note that due to signaling deficiencies in Windows, if the loop is empty, the keyboard interrupt is never triggered. A workaround for this situation is to queue a task that sleeps for several seconds. This guarantees that if the interrupt arrives while the task sleeps, the loop will notice when it wakes.

Asynchronous programming can be very confusing. I must confess that I started with some base assumptions that turned out to be wrong. It wasn't until I dove deeper into it that I realized what's really going on.

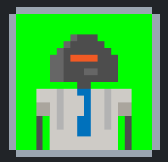
I hope this served as a good introduction to asyncio event loops and tasks, as

well as their possible uses. I know there are plenty of other articles out there, but I wanted to make something that tied things to some real world examples. If you have any questions or comments, feel free to drop them below and I'll help as best I can.

Follow @tryexceptpass

If you liked this article and want to keep up with what I'm working on, please recommend it, visit tryexecptpass.org for more topics and follow me on Twitter.

Share this story



@tryexceptpass

Cristian Medina

Read my stories

Related

Discover, triage, and prioritize Python errors in

Introduction to Scripting Languages

Scraping Tweet Replies with Python and Tweepy Twitter API

real-time

#scripting

@RohitLakh
Rohit Lakhotia
06/22/20

[A Step-b...

#python

@nich
Nicholas Resendez
06/21/20

Tags

#asyncio #python #multithreading