asyncio and friends

A gentle introduction to the wild world of async programming in Python

---

By Travis Hathaway

Who am I?

Name and Websites	**Travis Hathaway** https://travishathaway.com https://github.com/travishathaway
Hats I wear 	• Python Programmer • Musician/Guitar Player • Social Science Researcher
Interests	Music, Social Science Research, Software, Fitness Traveling

What are we going to cover today?

Basics of async in Python

Important theoretical concepts and designs

Learn *when* and *when not* to use async

(Hopefully) useful real-world examples

Why does asynchronous programming exist?

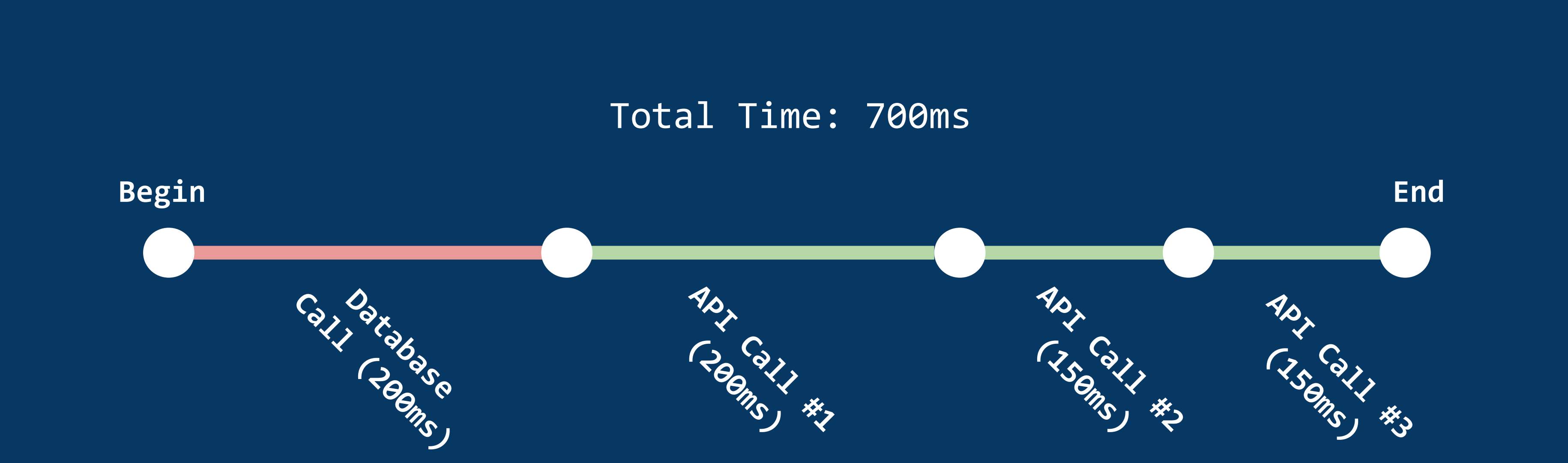
In a nutshell

efficiency and performance

Wordier answer:

It helps our programs deal with events which occur **independent** of the **main program flow**. These methods help us minimize time spent **blocking** and waiting for results caused by these events.

Synchronous Control Flow



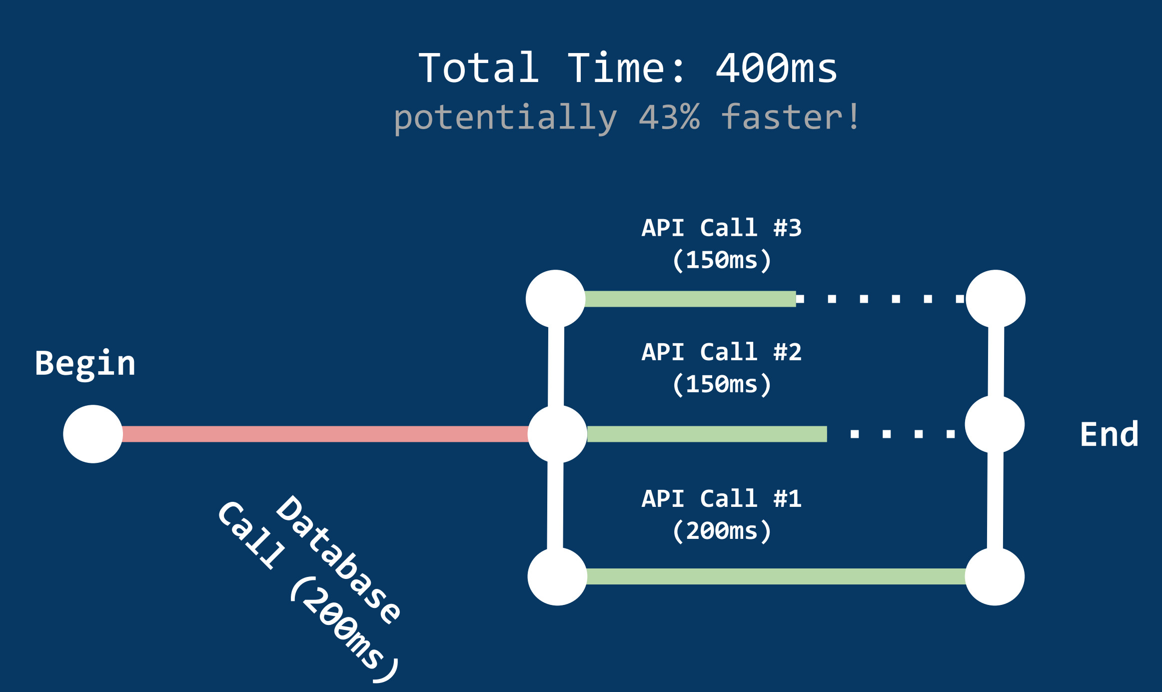
Blocking

This is what occurs when one process waits for another process to finish.

Sometimes this is necessary due to **temporal dependencies.**

Sometimes this waiting is unnecessary because these operations can be completed **in tandem**.

Asynchronous Control Flow



Let’s check out some examples

async “hello world!”

```
import asyncio


async def main_async():
    await asyncio.sleep(0.5)
    print('Hello Python Users Berlin!')

asyncio.run(main_async())

---
Output: Hello Python Users Berlin!!!
```

async "hello world!"

```python
import asyncio

async def main_async():
    await asyncio.sleep(0.5)
    print('Hello Python Users Berlin!')

asyncio.run(main_async())

---
Output: Hello Python Users Berlin!!!
```

All async programs in Python need to run inside an event loop. The asyncio module gives us the ability to create an event loop to run our async code in.

async “hello world!”

```
import asyncio


async def main_async():
    await asyncio.sleep(0.5)
    print('Hello Python Users Berlin!')

asyncio.run(main_async())

---
Output: Hello Python Users Berlin!!!
```

With this keyword, you specify that this function returns a `coroutine` type.

To actually run the function, we either use the await keyword or pass it in to our event loop…

async “hello world!”

```
import asyncio


async def main_async():
    await asyncio.sleep(0.5)
    print('Hello Python Users Berlin!')

asyncio.run(main_async())

---
Output: Hello Python Users Berlin!!!
```

Here is where we define our event loop. By passing in the coroutine that `main_async()` returns, our function is executed.

async “hello world!”

```
import asyncio


async def main_async():
    await asyncio.sleep(0.5)
    print('Hello Python Users Berlin!')

asyncio.run(main_async())

---
Output: Hello Python Users Berlin!!!
```

Our example also shows how we await other coroutines such as the `asyncio.sleep` function.

When we call await we yield control of our program to other tasks in the event loop. If multiple tasks are running, then they get to execute while we wait for this call to return.

This is where concurrency happens

Coroutine

Coroutines are a more generalized form of subroutines.

Subroutines are entered at one point and exited at another point.

Coroutines can be entered, exited, and resumed at many different points.

They can be implemented with the `async def` statement.

Source: https://docs.python.org/3/glossary.html#term-coroutine

Where else have we seen *coroutines*?

```
def get_top_customer_details(limit: int = 10) -> Generator:
    top_customer_ids = get_top_customers(limit)

    for customer_id in top_customer_ids:
        yield get_customer_details(customer_id)


for customer in get_top_customer_details(10):
    # Send some marketing spam
    send_big_deal_notification(customer)
    send_more_marketing_stuff(customer)

    # Add to a VIP list for upcoming features
    add_to_vip_list(customer)
```

Where else have we seen *coroutines*?

```
def get_top_customer_details(limit: int = 10) -> Generator:
    top_customer_ids = get_top_customers(limit)

    for customer_id in top_customer_ids:
        yield get_customer_details(customer_id)


for customer in get_top_customer_details(10):
    # Send some marketing spam
    send_big_deal_notification(customer)
    send_more_marketing_stuff(customer)

    # Add to a VIP list for upcoming features
    add_to_vip_list(customer)
```

Generators are a type of coroutine (simpler).

Using a yield statement allows these two for loops to cooperate with each other by passing the execution back and forth.

Time for a more complex example…

GET 20 files synchronously

```python
def sync_download():
    tile_server_url = 'https://tile-a.openstreetmap.fr/hot/13/1300/'
    start = 2920
    stop = 2941

    for id_ in range(start, stop):
        url = f'{tile_server_url}{id_}.png'
        resp = requests.get(url)
        # do something with response…
```

GET 20 files asynchronously

```python
def async_download():
    tile_server_url = 'https://tile-a.openstreetmap.fr/hot/13/1300/'
    start = 2920
    stop = 2941
    urls = tuple(
        f'{tile_server_url}{id_}.png'
        for id_ in range(start, stop)
    )

    async def main():
        async with aiohttp.ClientSession() as session:
            async def _get(url):
                resp = await session.get(url)
                # do something with response…

            await asyncio.gather(*(_get(url) for url in urls))

    asyncio.run(main())
```

GET 20 files asynchronously

```
def async_download():
    tile_server_url = 'https://tile-a.openstreetmap.fr/hot/13/1300/'
    start = 2920
    stop = 2941
    urls = tuple(
        f'{tile_server_url}{id_}.png'
        for id_ in range(start, stop)
    )

    async def main():
        async with aiohttp.ClientSession() as session:
            async def _get(url):
                resp = await session.get(url)
                # do something with response…

            await asyncio.gather(*(_get(url) for url in urls))

    asyncio.run(main())
```

With this example, we add true concurrency.

This is accomplished with `asyncio.gather` which accepts a sequence of coroutine objects.

These all get scheduled for running in our main event loop.

Here's some actual performance statistics

```
$ simple_http sync
Avg over 5 attempts: 3.545234


$ simple_http async
Avg over 5 attempts: 0.233422 # 17x faster!
```

When should I not use async?

You have nothing that can sensibly be run concurrently

You have something that could be run concurrently, but it is CPU bound. **Use multi-processing instead.**

Your code cannot be feasibly refactored to convert all *synchronous, blocking* calls to *asynchronous, non-blocking* calls. **Use threads instead**

Is there anything else???

How about some real world examples?

Increasing concurrency via async can lead to downstream problems…

If you're not careful, you could **DoS** your own systems via too many requests.

How do we address this in our code?

```
async def limited_download(urls: tuple[str], limit: int = 10):

    async with aiohttp.ClientSession() as session:
        sem = asyncio.Semaphore(limit)

        async def _download_url(url):
            async with sem:
                await download_url(session, url)

        tasks = tuple(
            _download_url(session, url)
            for url in urls
        )
        await asyncio.gather(*tasks)
```

How do we address this in our code?

```python
async def limited_download(urls: tuple[str], limit: int = 10):

    async with aiohttp.ClientSession() as session:
        sem = asyncio.Semaphore(limit)

        async def _download_url(url):
            async with sem:
                await download_url(session, url)

        tasks = tuple(
            _download_url(session, url)
            for url in urls
        )
        await asyncio.gather(*tasks)
```

The Semaphore object can be used as an async context manager.

This effectively slows down our code as the Semaphore object doesn’t allow more than the provide limit to be running at a single time.

How do we build more complicated workflows?

One way to better organize your async code is by using an **asyncio.Queue**

When using queues it becomes fairly easy to use the Pattern (i.e. pub/sub)

```python
async def main() -> None:
    points = (
        Point(lat=54.305902, lon=10.123282, label='Kiel'),
        Point(lat=52.521021, lon=13.381268, label='Berlin'),
        Point(lat=48.144049, lon=11.575928, label='München'),
    )

    queue = asyncio.Queue()

    async def produce(point: Point) -> None:
        while True:
            async with aiohttp.ClientSession() as session:
                weather_data = await get_weather_data(session, point)
                await queue.put(weather_data)
            await asyncio.sleep(5)

    async def consume():
        while True:
            data = await queue.get()
            print(f'{data.point.label}: {data.temperature} :: {data.description}')
            queue.task_done()

    asyncio.create_task(consume())

    await asyncio.gather(*(produce(point) for point in points))
```

```python
async def main() -> None:
    points = (
        Point(lat=54.305902, lon=10.123282, label='Kiel'),
        Point(lat=52.521021, lon=13.381268, label='Berlin'),
        Point(lat=48.144049, lon=11.575928, label='München'),
    )

    queue = asyncio.Queue()

    async def produce(point: Point) -> None:
        while True:
            async with aiohttp.ClientSession() as session:
                weather_data = await get_weather_data(session, point)
                await queue.put(weather_data)
            await asyncio.sleep(5)

    async def consume():
        while True:
            data = await queue.get()
            print(f'{data.point.label}: {data.temperature} :: {data.description}')
            queue.task_done()

    asyncio.create_task(consume())

    await asyncio.gather(*(produce(point) for point in points))
```

The asyncio library provides its own implementation of a Queue data structure.

```python
async def main() -> None:
    points = (
        Point(lat=54.305902, lon=10.123282, label='Kiel'),
        Point(lat=52.521021, lon=13.381268, label='Berlin'),
        Point(lat=48.144049, lon=11.575928, label='München'),
    )

    queue = asyncio.Queue()

    async def produce(point: Point) -> None:
        while True:
            async with aiohttp.ClientSession() as session:
                weather_data = await get_weather_data(session, point)
                await queue.put(weather_data)
            await asyncio.sleep(5)

    async def consume():
        while True:
            data = await queue.get()
            print(f'{data.point.label}: {data.temperature} :: {data.description}')
            queue.task_done()

    asyncio.create_task(consume())

    await asyncio.gather(*(produce(point) for point in points))
```

We can add objects to the queue with `queue.put`

```python
async def main() -> None:
    points = (
        Point(lat=54.305902, lon=10.123282, label='Kiel'),
        Point(lat=52.521021, lon=13.381268, label='Berlin'),
        Point(lat=48.144049, lon=11.575928, label='München'),
    )

    queue = asyncio.Queue()

    async def produce(point: Point) -> None:
        while True:
            async with aiohttp.ClientSession() as session:
                weather_data = await get_weather_data(session, point)
                await queue.put(weather_data)
            await asyncio.sleep(5)

    async def consume():
        while True:
            data = await queue.get()
            print(f'{data.point.label}: {data.temperature} :: {data.description}')
            queue.task_done()

    asyncio.create_task(consume())

    await asyncio.gather(*(produce(point) for point in points))
```

We can retrieve these objects with a call to `queue.get`

```python
async def main() -> None:
    points = (
        Point(lat=54.305902, lon=10.123282, label='Kiel'),
        Point(lat=52.521021, lon=13.381268, label='Berlin'),
        Point(lat=48.144049, lon=11.575928, label='München'),
    )

    queue = asyncio.Queue()

    async def produce(point: Point) -> None:
        while True:
            async with aiohttp.ClientSession() as session:
                weather_data = await get_weather_data(session, point)
                await queue.put(weather_data)
            await asyncio.sleep(5)

    async def consume():
        while True:
            data = await queue.get()
            print(f'{data.point.label}: {data.temperature} :: {data.description}')
            queue.task_done()

    asyncio.create_task(consume())

    await asyncio.gather(*(produce(point) for point in points))
```

We can retrieve these objects with a call to `queue.get`

Final Thoughts

Async programming is complex (*async/await syntax just tries to make this complexity easier to deal with!*)

Ensure you have the right use case before starting down the async path (*make sure other options do not work better*)

Be aware of how your async program fits in with your environment, will it overburden other systems?

Further Resources

- [Demystifying Python's Async and Await Keywords]
- [Lynn Root - Advanced asyncio: Solving Real-world Production Problems - PyCon 2019 (YouTube)]
- https://fastapi.tiangolo.com/async/ (great explanation of async programming)
- https://realpython.com/async-io-python/ (packed with tons of useful information)