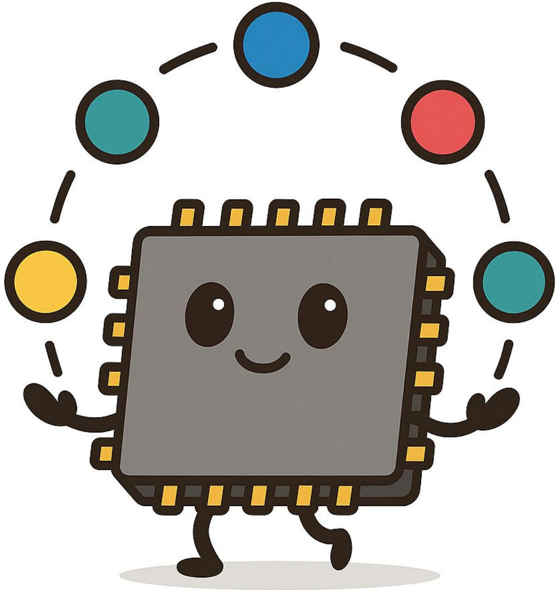
Asynchronous Programming with Python

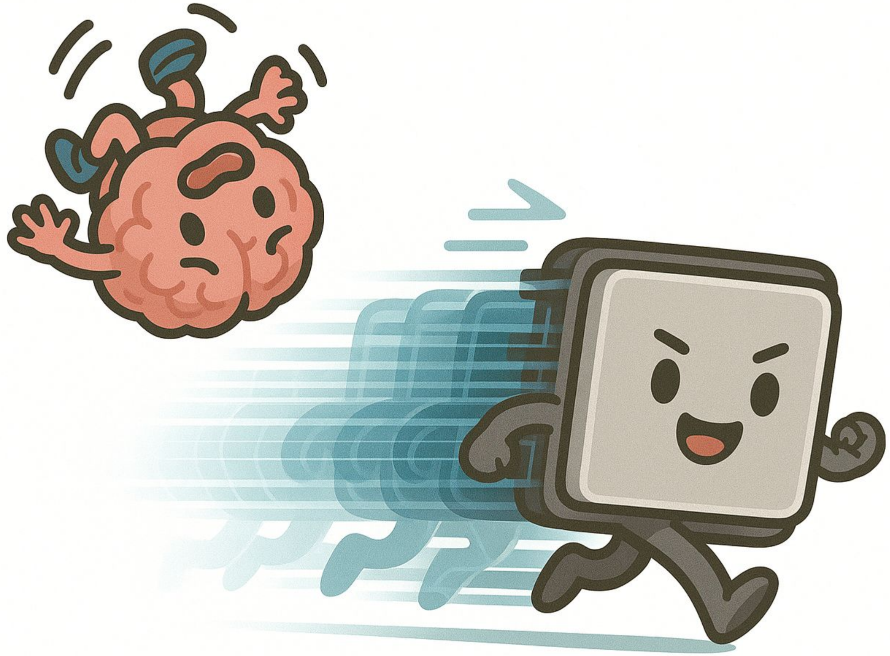


Improving CPU utilisation with asyncio task juggling

Owen Lamont

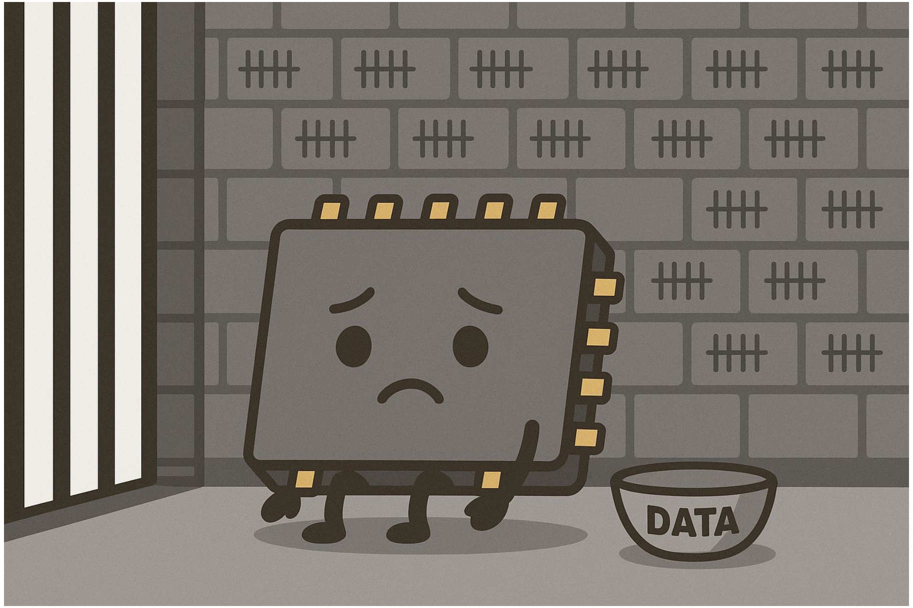
CPUs tick really fast compared to our brains

- Neurons tick at 100-200Hz (max).
- CPUs tick between 1 - 4 GHz.
- Try to imagine watching the world through a 10,000,000X high speed camera.
    - 100 milliseconds takes over 10 days to watch.
    - A 5 minute coffee break takes 95 years to watch.
    - Humans move at about the speed grass grows at.



CPUs are fast - but it’s easy to waste their time waiting

- CPUs process data super-humanly fast - but often have nothing to do - this means they are Input/Output (I/O) bound.
- CPUs have to wait different amounts of time to get data to work with.

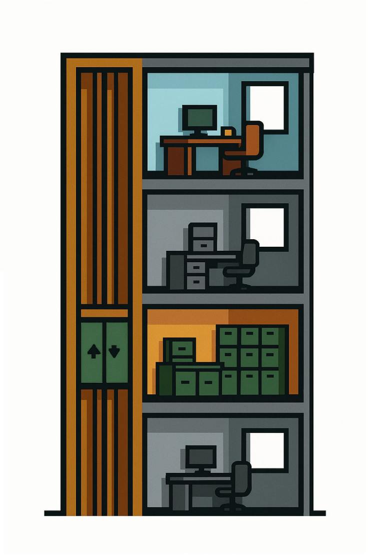


The first ten million years were the worst. And the second ten million, they were the worst too. The third ten million I didn’t enjoy at all. After that I went into a bit of a decline. - Marvin the Android, in The Restaurant at the End of the Universe

CPU Input/Output Latency

A metaphor for CPU data latency is like an office worker accessing info:

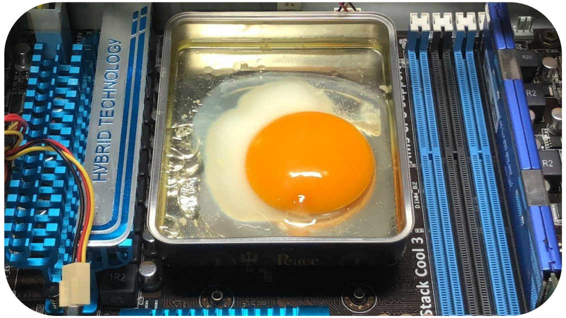
- Registers - top of the desk
- L1 Cache - top desk draw
- L2 Cache - middle desk draw
- L3 Cache - bottom desk draw
- RAM - filing room on a different floor
- Solid State Drive - local library
- External APIs - inter-library loan



Why care about keeping the CPU waiting?

- We're often renting CPUs by time, not by utilisation.
- CPUs are energy intensive mini electric heaters.
- The value of outputs for many applications decreases with latency.

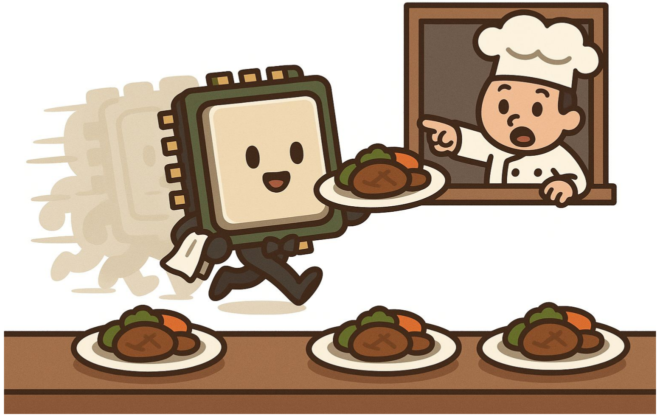
How did you end up in poverty? Gamble? drugs?
I left an ec2 instance on.



CODE CARBON

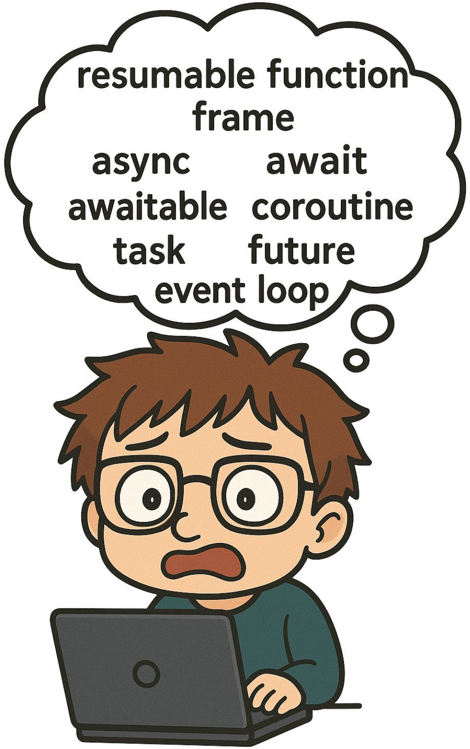
How can we reduce CPU waiting time?

- Buy faster / more expensive hardware and Internet connections.
- Use a CPU that is physically closer to the data sources it needs to access.
- Give the CPU more jobs to do so it can switch jobs while waiting on one to be ready.

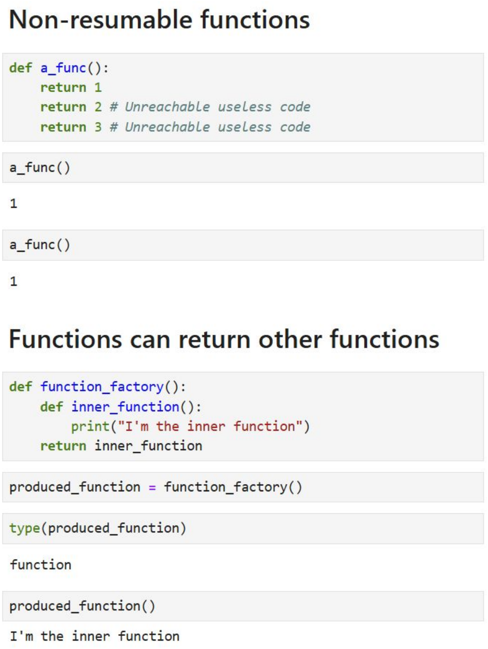


Async and awaiting: some asyncio lingo

- We can give the CPU multiple jobs with the Python asyncio library.
- There's a few key concepts and some terminology you need to understand asyncio.
- First resumable functions:
  - Resumable functions (coroutines) have existed since some of the earliest programming languages.
  - These functions preserve their interval state (frame) so they can resume from where left off since their last call.
  - Python has two types of coroutines, generators and async coroutines.



Functions, Frames, & Resumable Functions (Coroutines)



Non-resumable functions

```python
def a_func():
    return 1
    return 2 # Unreachable useless code
    return 3 # Unreachable useless code
```

```python
a_func()
```

```
1
```

```python
a_func()
```

```
1
```

Functions can return other functions

```python
def function_factory():
    def inner_function():
        print("I'm the inner function")
    return inner_function
```

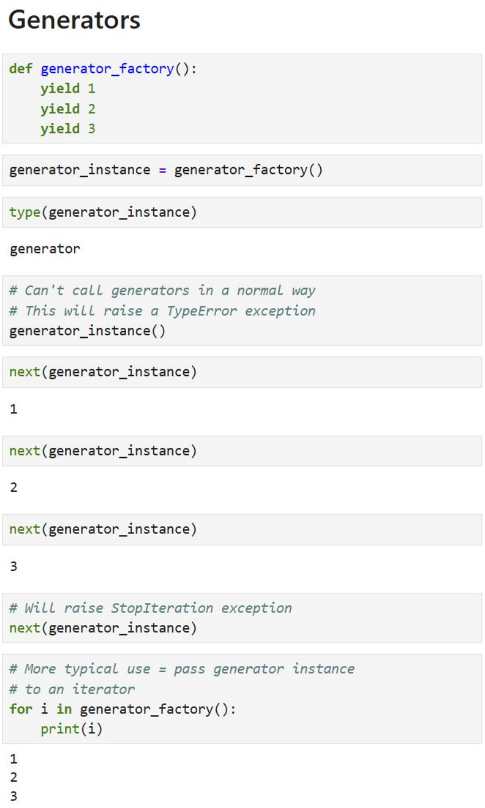
```python
produced_function = function_factory()
```

```python
type(produced_function)
```

```
function
```

```python
produced_function()
```

```
I'm the inner function
```



Generators

```python
def generator_factory():
    yield 1
    yield 2
    yield 3
```

```python
generator_instance = generator_factory()
```

```python
type(generator_instance)
```

```
generator
```

```python
# Can't call generators in a normal way
# This will raise a TypeError exception
generator_instance()
```

```python
next(generator_instance)
```

```
1
```

```python
next(generator_instance)
```

```
2
```

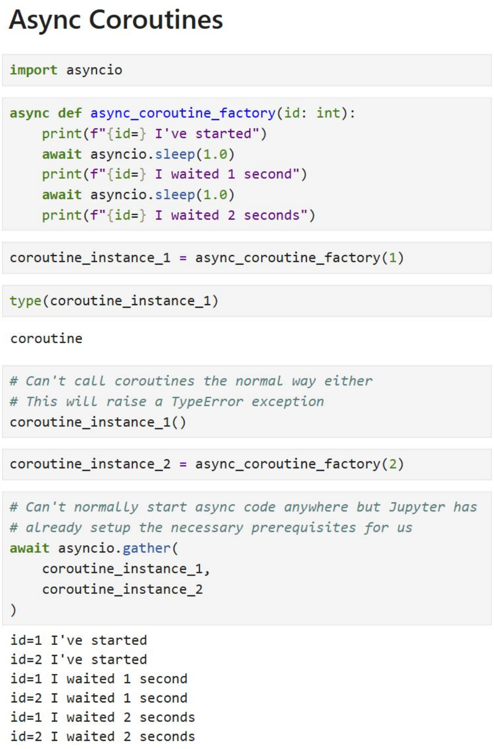
```python
next(generator_instance)
```

```
3
```

```python
# Will raise StopIteration exception
next(generator_instance)
```

```python
# More typical use = pass generator instance
# to an iterator
for i in generator_factory():
    print(i)
```

```
1
2
3
```



Async Coroutines

```python
import asyncio
```

```python
async def async_coroutine_factory(id: int):
    print(f"{id=} I've started")
    await asyncio.sleep(1.0)
    print(f"{id=} I waited 1 second")
    await asyncio.sleep(1.0)
    print(f"{id=} I waited 2 seconds")
```

```python
coroutine_instance_1 = async_coroutine_factory(1)
```

```python
type(coroutine_instance_1)
```

```
coroutine
```

```python
# Can't call coroutines the normal way either
# This will raise a TypeError exception
coroutine_instance_1()
```

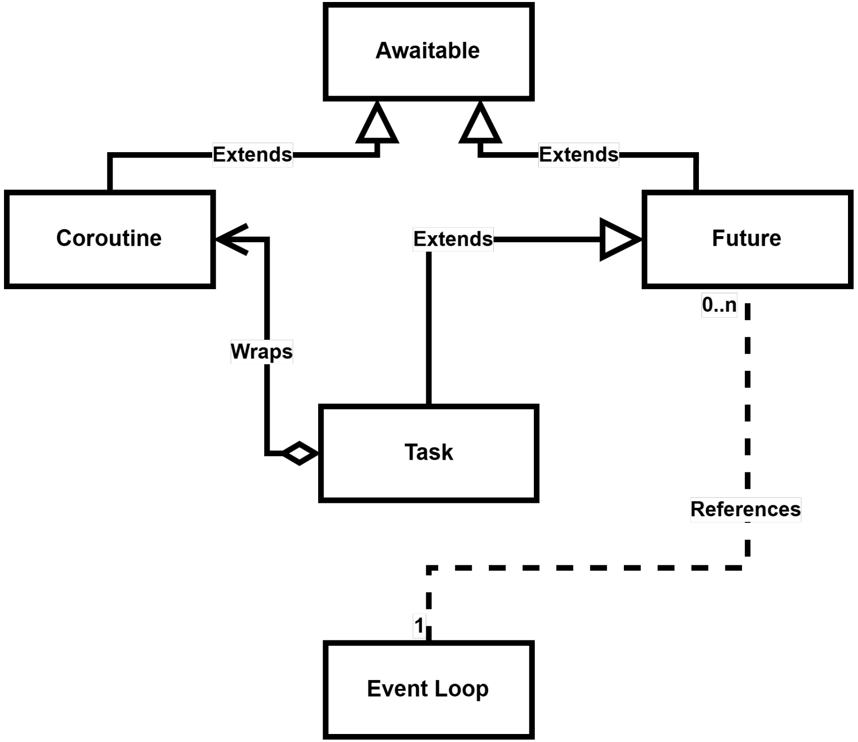
```python
coroutine_instance_2 = async_coroutine_factory(2)
```

```python
# Can't normally start async code anywhere but Jupyter has
# already setup the necessary prerequisites for us
await asyncio.gather(
    coroutine_instance_1,
    coroutine_instance_2
)
```

```
id=1 I've started
id=2 I've started
id=1 I waited 1 second
id=2 I waited 1 second
id=1 I waited 2 seconds
id=2 I waited 2 seconds
```

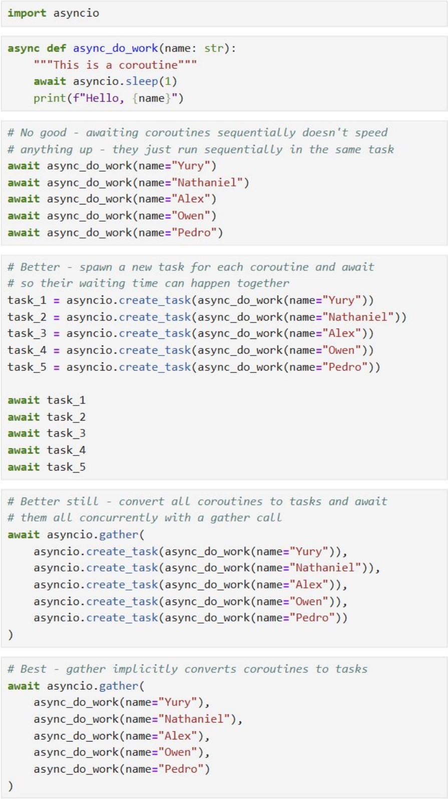
Asyncio abstractions

- Anything that can follow the **await** keyword is an Awaitable (await is kind of like a specialised yield).
- Coroutines (in code) are what we call functions defined with an **async def**, and they return Coroutine instances.
- Coroutines get wrapped in Tasks (a type of Future) and the Event Loop loops through Futures executing the next one that is ready to progress.



Awaiting many tasks at once

- When using asyncio you want to await tasks.
- You can await coroutines directly, that just adds them to the current task.
- Awaiting multiple tasks simultaneously allows the event loop to start them all and progress each one as it is ready.
- You can explicitly spawn new tasks for coroutines - and some asyncio functions implicitly wrap coroutines with tasks.



```python
import asyncio
```

```python
async def async_do_work(name: str):
    """This is a coroutine"""
    await asyncio.sleep(1)
    print(f"Hello, {name}")
```

```python
# No good - awaiting coroutines sequentially doesn't speed
# anything up - they just run sequentially in the same task
await async_do_work(name="Yury")
await async_do_work(name="Nathaniel")
await async_do_work(name="Alex")
await async_do_work(name="Owen")
await async_do_work(name="Pedro")
```

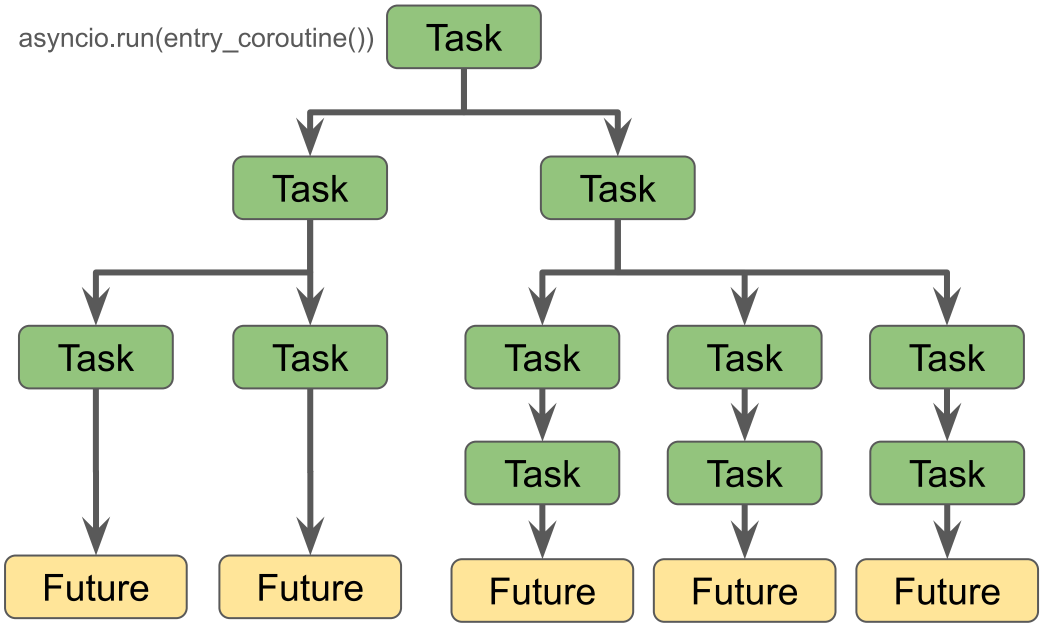
```python
# Better - spawn a new task for each coroutine and await
# so their waiting time can happen together
task_1 = asyncio.create_task(async_do_work(name="Yury"))
task_2 = asyncio.create_task(async_do_work(name="Nathaniel"))
task_3 = asyncio.create_task(async_do_work(name="Alex"))
task_4 = asyncio.create_task(async_do_work(name="Owen"))
task_5 = asyncio.create_task(async_do_work(name="Pedro"))

await task_1
await task_2
await task_3
await task_4
await task_5
```

```python
# Better still - convert all coroutines to tasks and await
# them all concurrently with a gather call
await asyncio.gather(
    asyncio.create_task(async_do_work(name="Yury")),
    asyncio.create_task(async_do_work(name="Nathaniel")),
    asyncio.create_task(async_do_work(name="Alex")),
    asyncio.create_task(async_do_work(name="Owen")),
    asyncio.create_task(async_do_work(name="Pedro"))
)
```

```python
# Best - gather implicitly converts coroutines to tasks
await asyncio.gather(
    async_do_work(name="Yury"),
    async_do_work(name="Nathaniel"),
    async_do_work(name="Alex"),
    async_do_work(name="Owen"),
    async_do_work(name="Pedro")
)
```

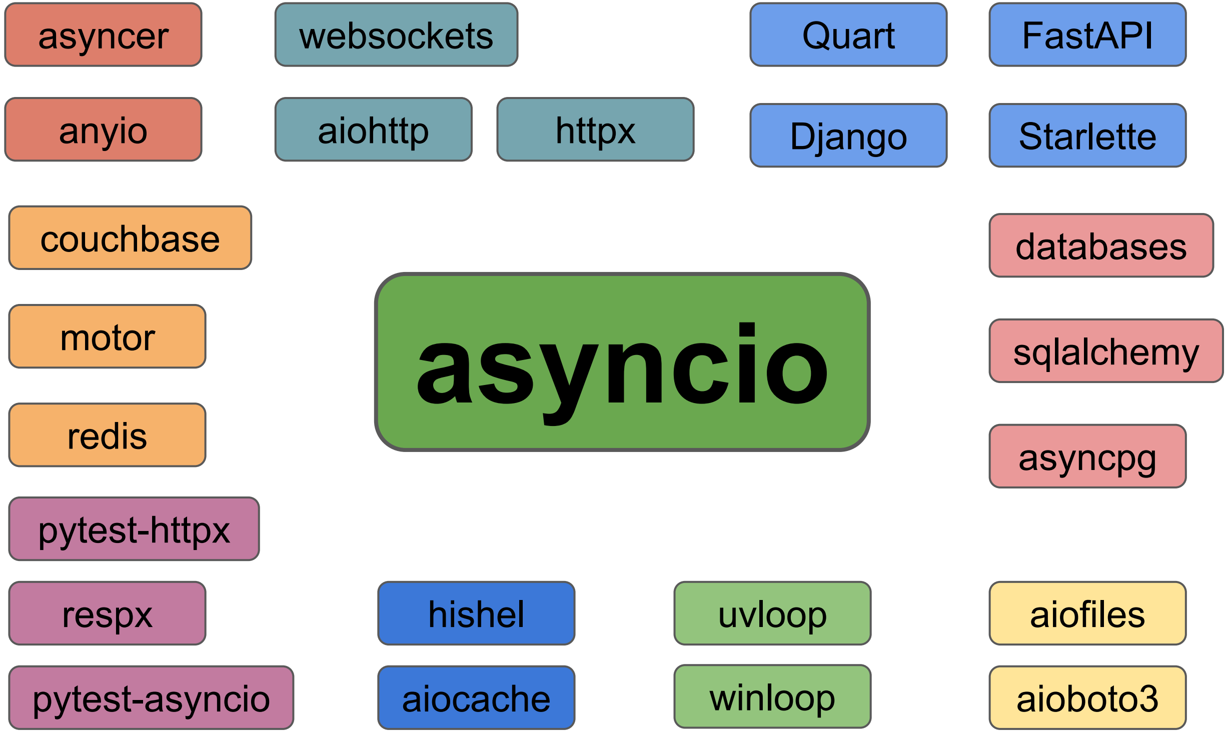
A well structured asyncio program

- You can visualise a good asyncio program as a tree of Tasks with Future leaves.
- Those Futures can leverage low level asyncio and OS APIs (or parallel processes and threads) to yield control to the event loop.
- When multiple Futures are awaited concurrently CPU wait time is reduced.



Asyncio and Friends

- Many packages are built around asyncio.
- With these we can await I/O bound jobs, e.g. database queries, file reads, API requests…
- They use low level asyncio promises that hook into the OS or threads/processes to parallelize wait times.



asyncer
anyio
websockets
aiohttp
httpx
Quart
FastAPI
Django
Starlette
couchbase
motor
redis
asyncio
databases
sqlalchemy
asyncpg
pytest-httpx
respx
pytest-asyncio
hishel
aiocache
uvloop
winloop
aiofiles
aioboto3

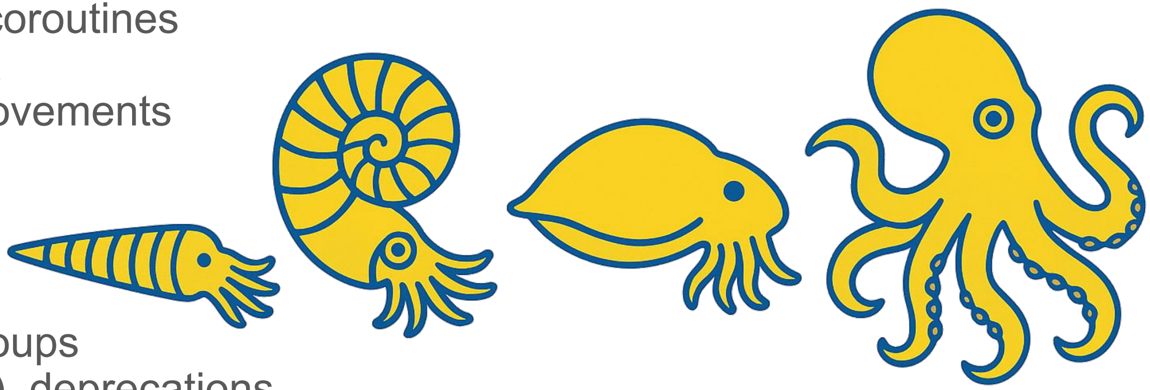
It’s too Asynchronous!

- It’s relatively easy to create Denial of Service (DOS) attacks by accident.
- Computers don’t like getting 10000 concurrent requests, opening 10000 connections, or 10000 files.
- Asynchronous code is a minor superpower - use it responsibly!
- You can rate limit concurrency with asyncio Semaphore, leverage limits in httpx, and use packages like aiolimiter.

Asyncio is evolving fast in Python

Python Version / Async Features

3.4	Initial release, yield from coroutines
3.5	Native async/await syntax
3.6	Executor/threadsafe improvements
3.7	asyncio.run(), stable API
3.8	Debug tools
3.9	asyncio.to_thread()
3.10	Context-based loop
3.11	TaskGroup, Exception Groups
3.12	Performance, timeout_at(), deprecations
3.13	Cleanup, loop removal
3.14	Enhanced debugging and inspection



People to follow and learning resources

- Read Sebastián Ramírez async tutorial in the FastAPI and Asyncer docs.
- Watch some of Łukasz Langa's presentations / Pycon talks.
- I found Michael Kennedy's course on async Python very helpful (not free, but sometimes discounted on Humble Bundle).
- Armin Ronacher has several articles and critiques of asyncio.

Sebastián Ramírez

Łukasz Langa

Michael Kennedy

Armin Ronacher

Things I left out

- Advanced asyncio like __await__, asyncio primitives, event loop creation/life times.
- Challenges of asynchronous code, e.g. non-determinism, debugging, resource rate limiting, limited dunder method support.
- Other Python async libraries, e.g. Twisted, Tornado, Gevent, Trio, etc.
- CPU bound code and approaches.
- Parallelism with threads / processes / sub-interpreters / compute clusters.

Questions and Contact

I can be reached on:

- https://www.linkedin.com/in/owen-lamont/
- https://fosstodon.org/@owenrlamont
- https://bsky.app/profile/owen7ba.bsky.social

The example code can be found here: https://github.com/owenlamont/asyncio_meetup_talk

Let me know if you have any questions or comments!