LOKA | PYCON COLOMBIA 2025

How Not to Write Async Python

(and What to Do Instead)

Juan Pablo Vanegas
Backend Engineer @ Loka

whoami

jpvanegasc

LOKA

aws

PARTNER OF THE YEAR

WINNER

2024

This talk aims to help you write faster code with less bugs in less time by:

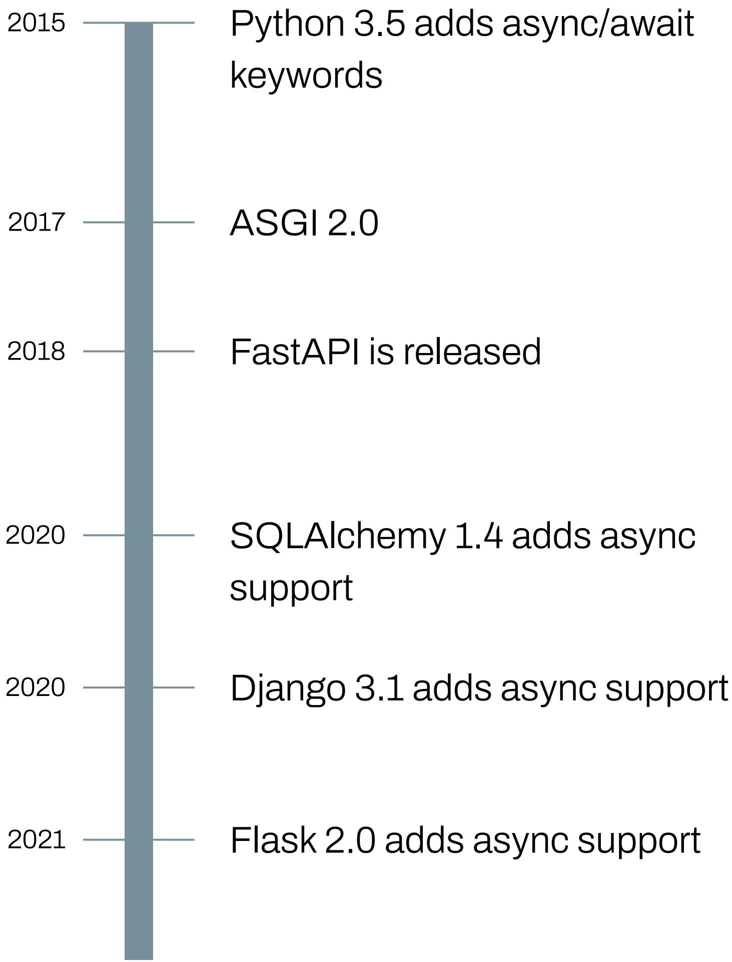
Triggering the pattern-recognizing human brain to identify async code smells and have a couple of tools at hand to solve them

It’s not:

- Deep dive on the event loop implementation
- Comparison between event loops (asyncio, uvloop) or backends (asyncio, trio)
- Philosophical discussions on programming techniques
- In-depth python core concepts (GIL and so on)

The Async Hype: Where Are We Now?

sin $\frac{1}{2}$ $\frac{\sqrt{2}}{2}$ $\frac{\sqrt{3}}{2}$
cos $\frac{\sqrt{3}}{2}$ $\frac{\sqrt{2}}{2}$ $\frac{1}{2}$
tan $\frac{\sqrt{3}}{3}$ 1 $\sqrt{3}$
2x
60°
x
30°
x$\sqrt{3}$
45°
s$\sqrt{2}$



2015 Python 3.5 adds async/await keywords

2017 ASGI 2.0

2018 FastAPI is released

2020 SQLAlchemy 1.4 adds async support

2020 Django 3.1 adds async support

2021 Flask 2.0 adds async support

"Faster execution because your program doesn't stay idle while waiting for a result"

Sync vs. Async vs. Parallel: What’s the Difference?

Let's make frozen pizza and a salad for dinner:

Option 1:

1. Open pizza box
2. Put pizza in oven
3. Wait 10 min
4. Put pizza in plate
5. Cut vegetables
6. Mix vegetables
7. Put vegetables in plate
8. Eat

Option 2:

1. Open pizza box
2. Put pizza in oven
3. Put timer for 10 min
4. Cut vegetables
5. Mix vegetables
6. Put vegetables in plate
7. Put pizza in plate
8. Eat

Option 3:

0. Call a friend

You:

1. Open pizza box
2. Put pizza in oven
3. Wait 10 min
4. Put pizza in plate

Them:

1. Cut vegetables
2. Mix vegetables
3. Put vegetables in plate

Eat

Now in programming terms:

Two kinds of waiting: I/O-bound and CPU bound:

- I/O-bound: waiting on *something else* to give the result
- CPU-bound: waiting on your own calculations to finish

Async is more suited for I/O bound, parallel for CPU-bound:

- Parallel adds overhead.
- Asynchronous has limited processing power available.

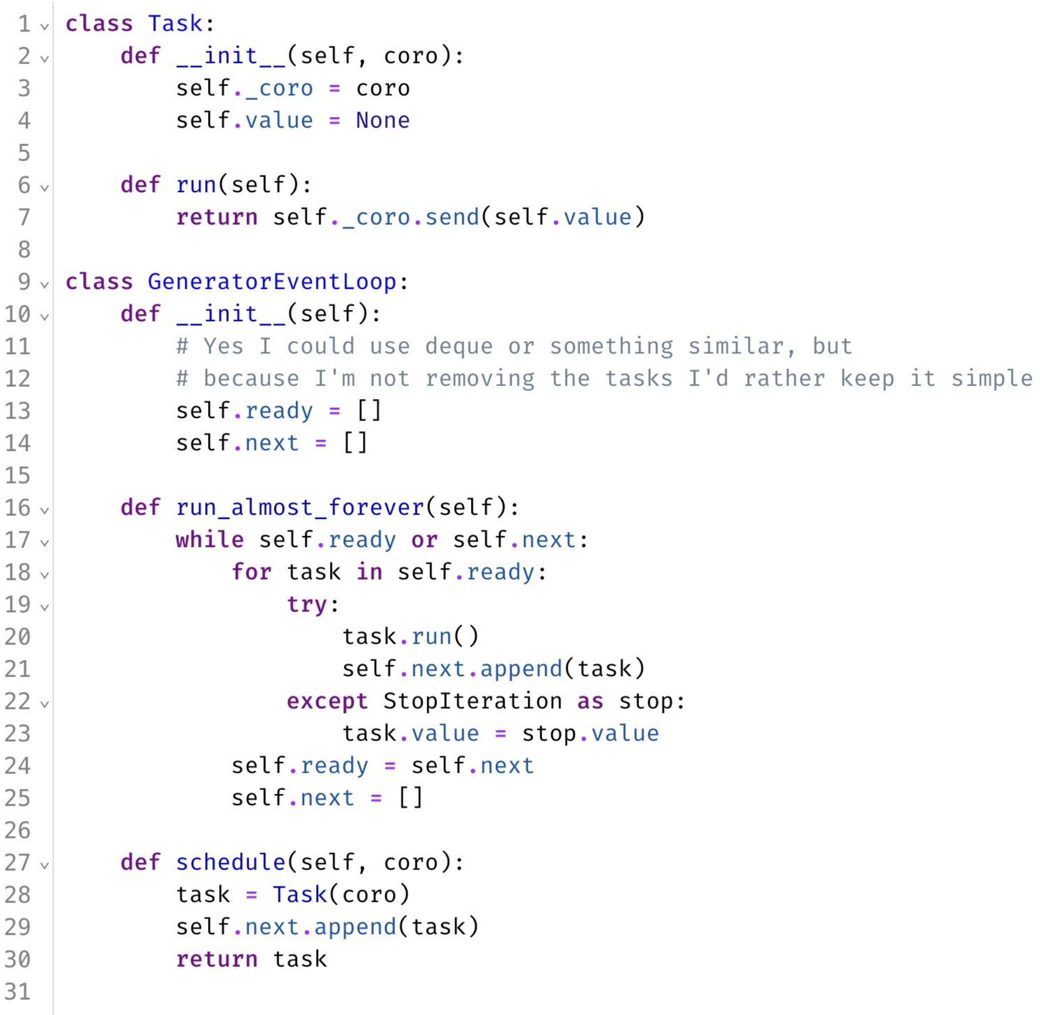
How the Event Loop *Actually* Works

The event loop manages the execution of asynchronous operations by continuously checking a queue for tasks that are ready to be executed, allowing the program to run non-blocking code while still processing events.

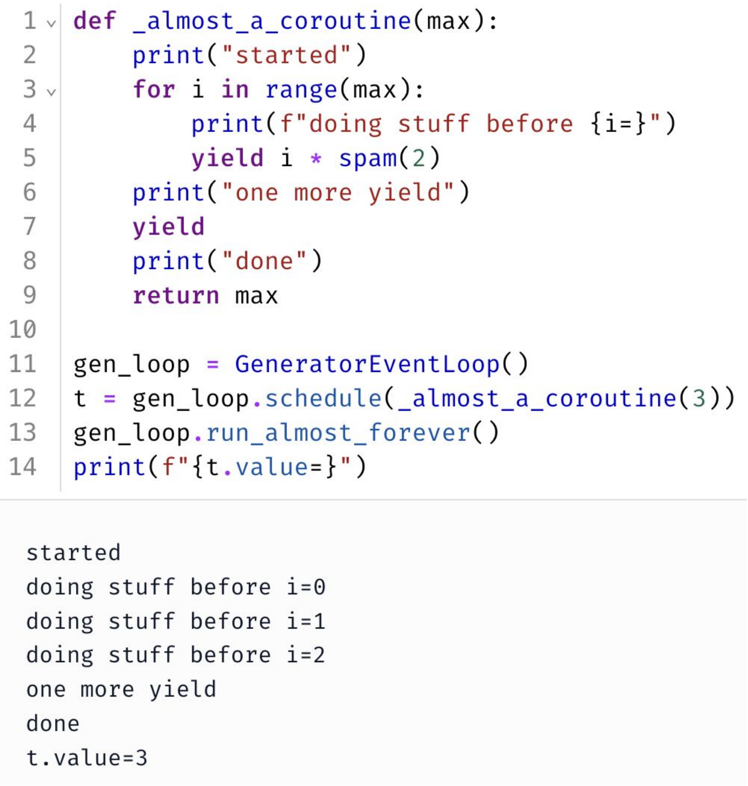
→ async operations: operations that depend on *something else* to finish

→ tasks ready to be executed: tasks that haven’t started waiting or that have already finished waiting

→ Non-blocking code: code that doesn’t need to wait

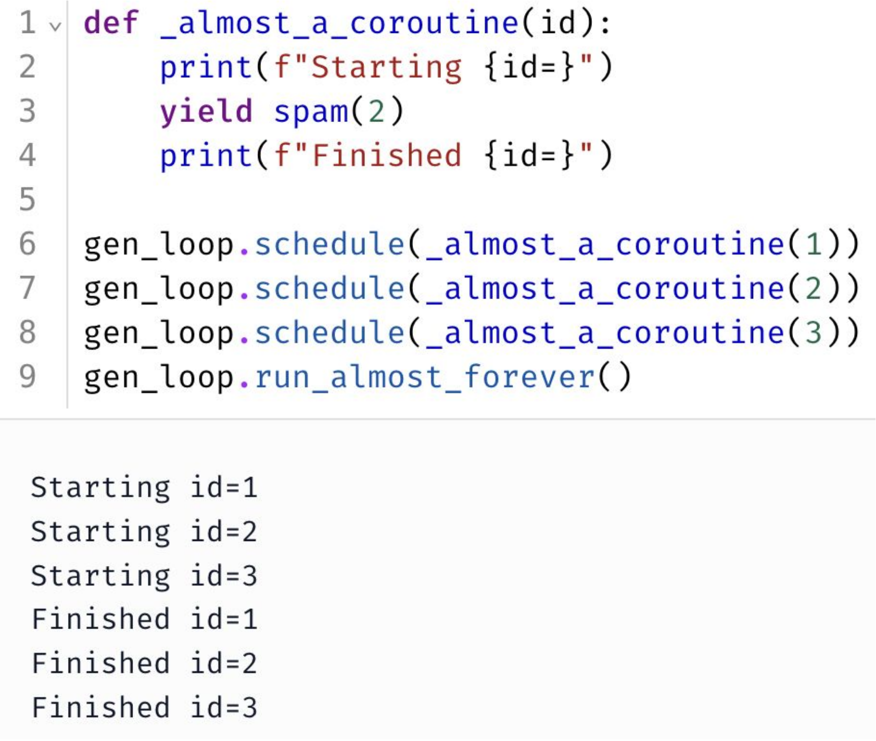


```python
class Task:
    def __init__(self, coro):
        self._coro = coro
        self.value = None

    def run(self):
        return self._coro.send(self.value)

class GeneratorEventLoop:
    def __init__(self):
        # Yes I could use deque or something similar, but
        # because I'm not removing the tasks I'd rather keep it simple
        self.ready = []
        self.next = []

    def run_almost_forever(self):
        while self.ready or self.next:
            for task in self.ready:
                try:
                    task.run()
                    self.next.append(task)
                except StopIteration as stop:
                    task.value = stop.value
            self.ready = self.next
            self.next = []

    def schedule(self, coro):
        task = Task(coro)
        self.next.append(task)
        return task

```



```python
def _almost_a_coroutine(max):
    print("started")
    for i in range(max):
        print(f"doing stuff before {i=}")
        yield i * spam(2)
    print("one more yield")
    yield
    print("done")
    return max

gen_loop = GeneratorEventLoop()
t = gen_loop.schedule(_almost_a_coroutine(3))
gen_loop.run_almost_forever()
print(f"{t.value=}")
```

```
started
doing stuff before i=0
doing stuff before i=1
doing stuff before i=2
one more yield
done
t.value=3
```



```python
def _almost_a_coroutine(id):
    print(f"Starting {id=}")
    yield spam(2)
    print(f"Finished {id=}")

gen_loop.schedule(_almost_a_coroutine(1))
gen_loop.schedule(_almost_a_coroutine(2))
gen_loop.schedule(_almost_a_coroutine(3))
gen_loop.run_almost_forever()
```

```
Starting id=1
Starting id=2
Starting id=3
Finished id=1
Finished id=2
Finished id=3
```

The Problem with Awaiting Everywhere

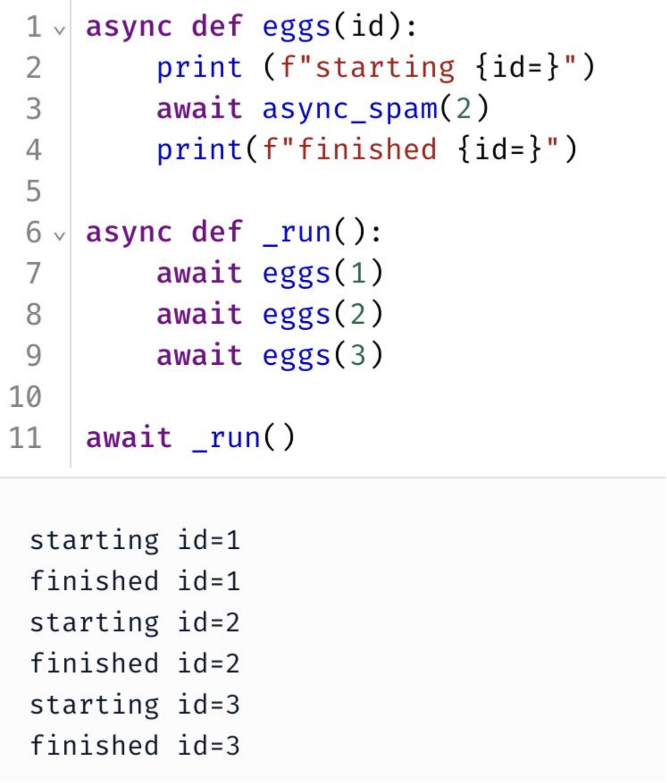
`await` tells the interpreter it should **wait** for something to complete.

- Do you *need* the result from the coroutine you're calling?
- Does the coroutine perform something that has to be done as a standalone request.

If you didn't answer yes to any of these questions, you might not want to await.

→ Maybe you need *several* results (e.g., a select and a count query)

→ Maybe the task can be completed eventually after you schedule



```python
async def eggs(id):
    print (f"starting {id=}")
    await async_spam(2)
    print(f"finished {id=}")

async def _run():
    await eggs(1)
    await eggs(2)
    await eggs(3)

await _run()
```

```
starting id=1
finished id=1
starting id=2
finished id=2
starting id=3
finished id=3
```

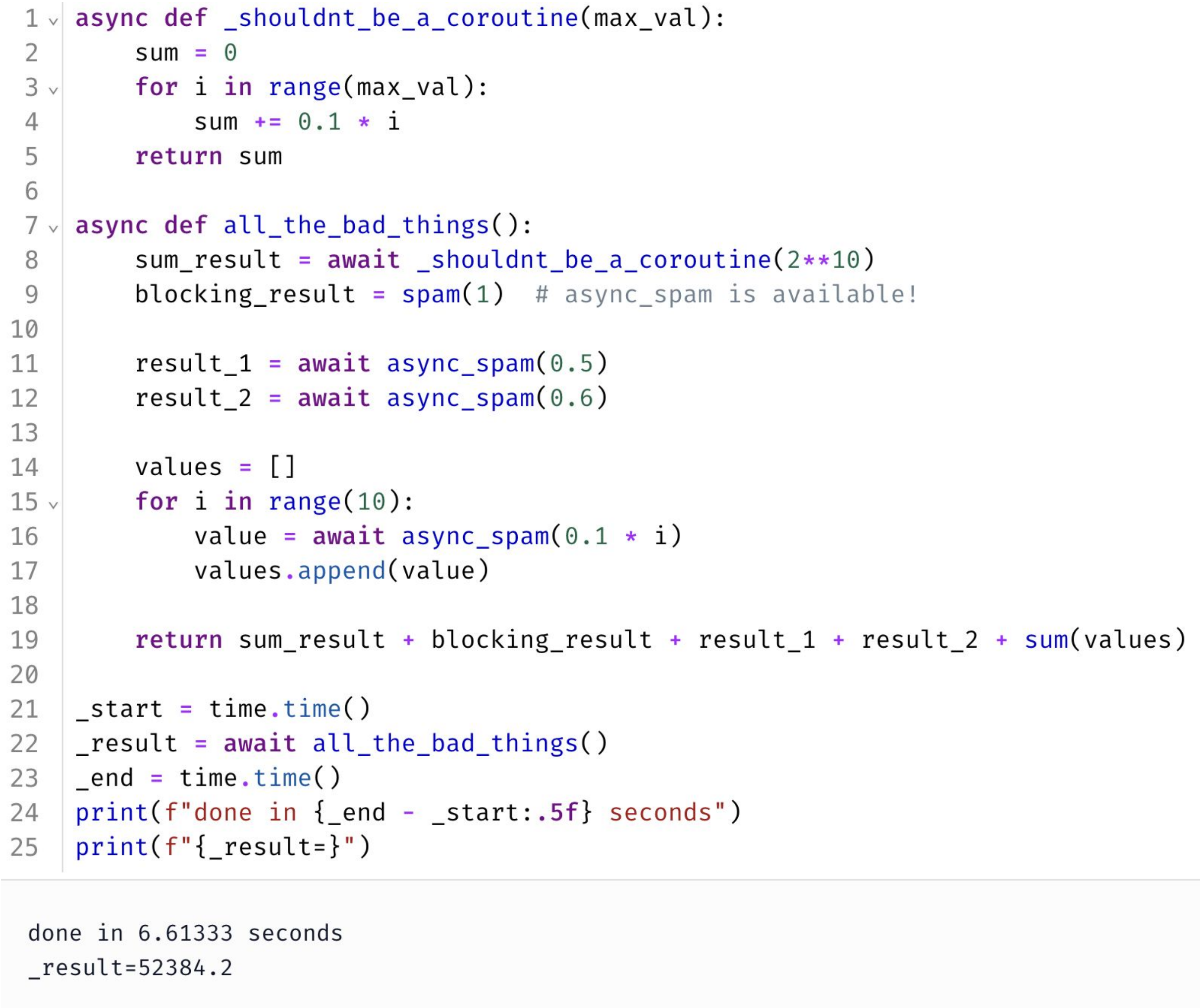
Fixing Async Code: The Right Tools

(to get you started)

YOU KNOW WHAT REALLY GRINDS MY GEARS?
WHAT REALLY
GRINDS
MY
GEARS
UNNECESSARY AWAITS IN FOR LOOPS
imgflip.com

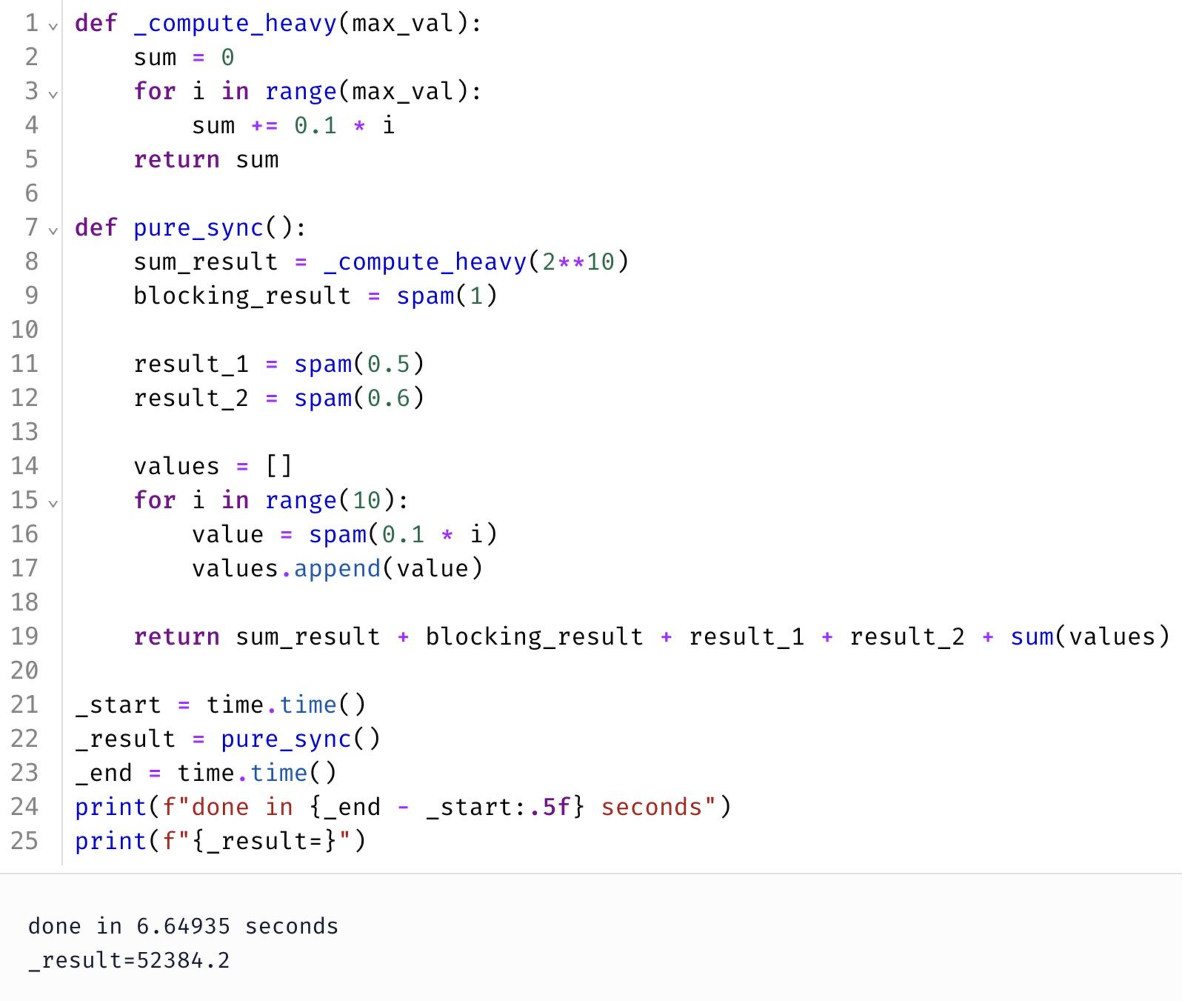
Common code smells:

- Defining a coroutine without awaiting anything inside → change `async def` to `def`
- Calling a blocking function inside a coroutine (using a library that doesn't support async) → Find a library that does, or wrap the sync function (`asyncio.run_in_executor`, `anyio.to_thread.run`, etc.)
- Awaiting inside loops → use `gather`
- Multiple awaits one after another → use `gather` or `TaskGroup`



```python
async def _shouldnt_be_a_coroutine(max_val):
    sum = 0
    for i in range(max_val):
        sum += 0.1 * i
    return sum

async def all_the_bad_things():
    sum_result = await _shouldnt_be_a_coroutine(2**10)
    blocking_result = spam(1)  # async_spam is available!

    result_1 = await async_spam(0.5)
    result_2 = await async_spam(0.6)

    values = []
    for i in range(10):
        value = await async_spam(0.1 * i)
        values.append(value)

    return sum_result + blocking_result + result_1 + result_2 + sum(values)

_start = time.time()
_result = await all_the_bad_things()
_end = time.time()
print(f"done in {_end - _start:.5f} seconds")
print(f"{_result=}")
```

```
done in 6.61333 seconds
_result=52384.2
```



```python
def _compute_heavy(max_val):
    sum = 0
    for i in range(max_val):
        sum += 0.1 * i
    return sum

def pure_sync():
    sum_result = _compute_heavy(2**10)
    blocking_result = spam(1)

    result_1 = spam(0.5)
    result_2 = spam(0.6)

    values = []
    for i in range(10):
        value = spam(0.1 * i)
        values.append(value)

    return sum_result + blocking_result + result_1 + result_2 + sum(values)

_start = time.time()
_result = pure_sync()
_end = time.time()
print(f"done in {_end - _start:.5f} seconds")
print(f"{_result=}")
```

```
done in 6.64935 seconds
_result=52384.2
```

Your new best friends:

`asyncio.gather(*coroutines)`

→ Use it to immediately run several coroutines (e.g. several requests to an API service).

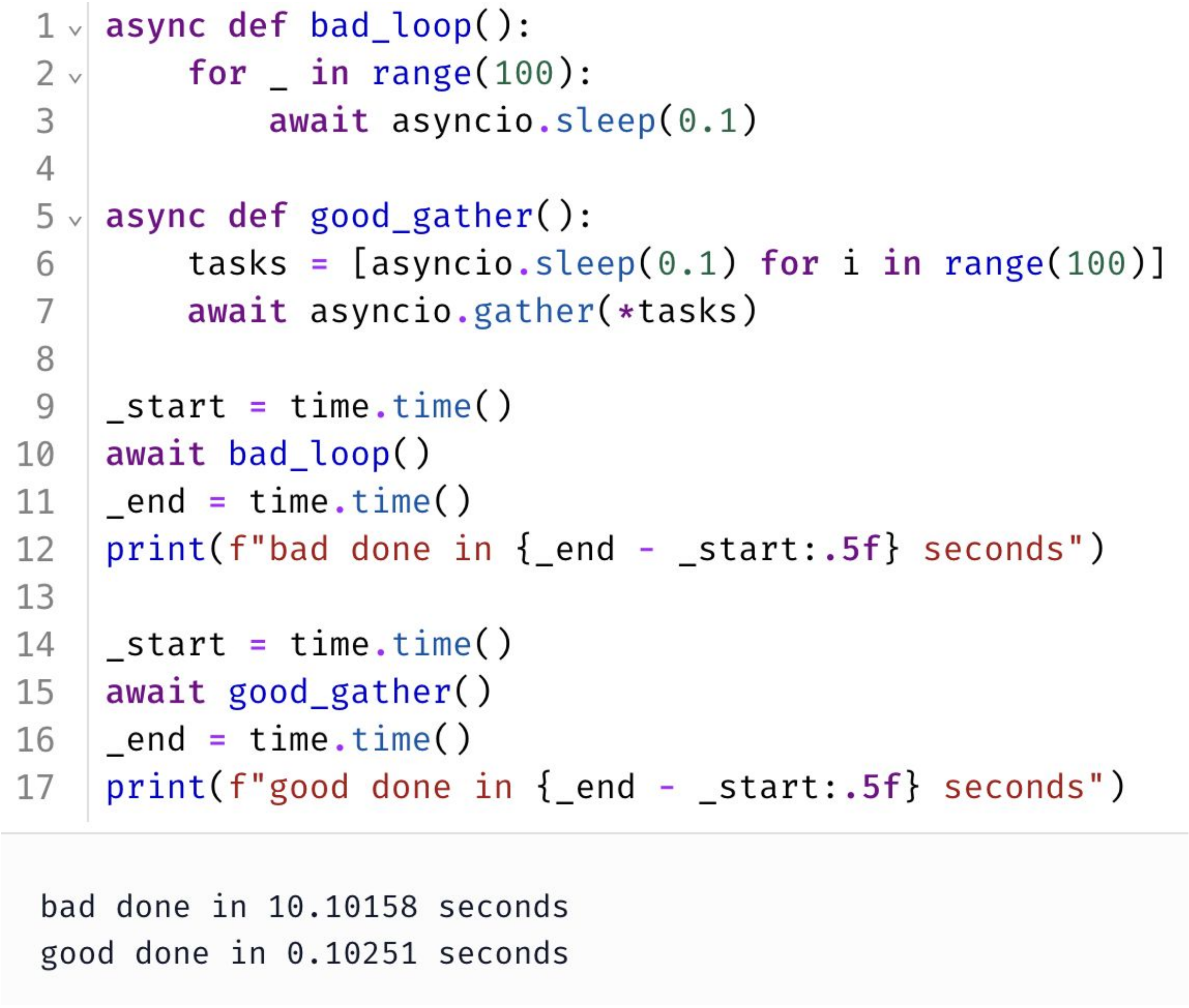
`asyncio.create_task(coroutine)`

→ Use it to *schedule* a coroutine for execution. Might be done now, might be done later.

`asyncio.TaskGroup()` (Introduced in 3.11)

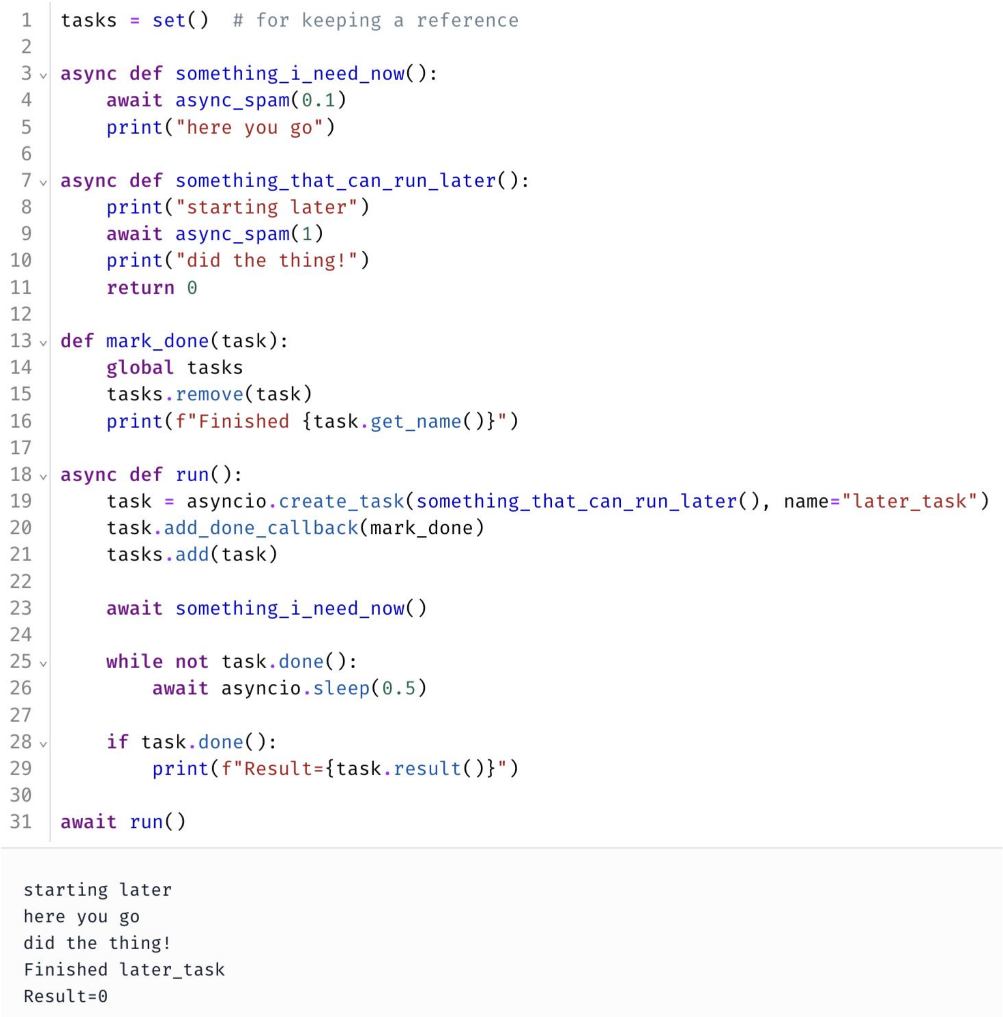
→ Similar to `gather`. Use it to easily hold a reference to the tasks! (coroutines sent to `gather` or to `create_task` might be garbage collected and mysteriously disappear if not properly referenced)

→ Catch exceptions with the `except*` syntax



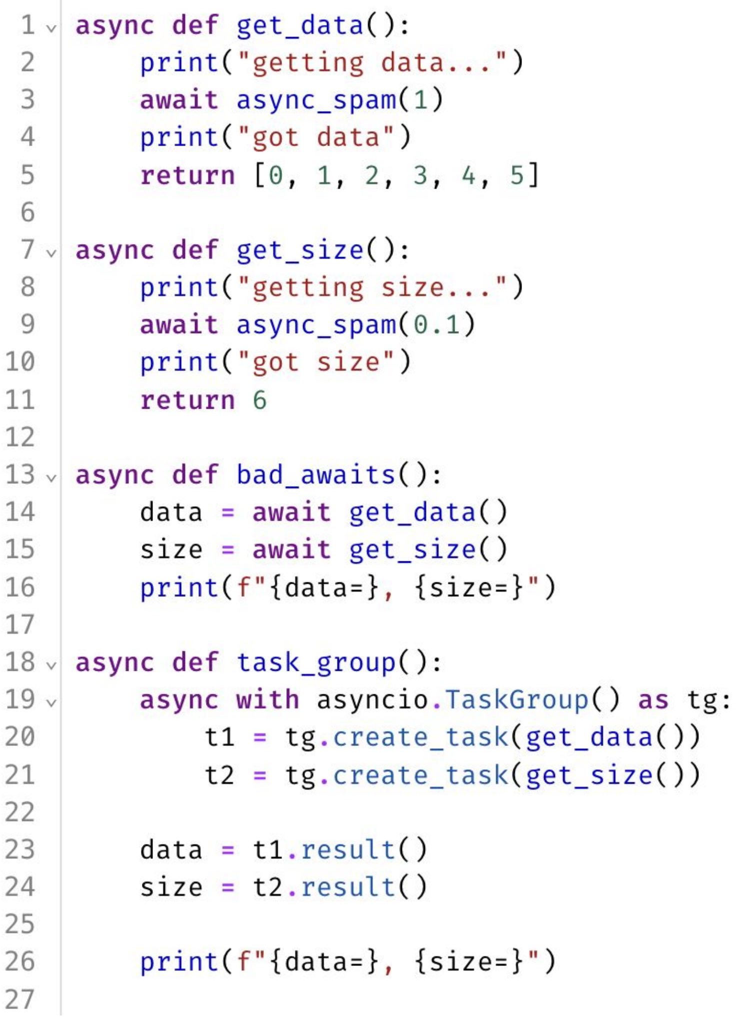
```python
async def bad_loop():
    for _ in range(100):
        await asyncio.sleep(0.1)

async def good_gather():
    tasks = [asyncio.sleep(0.1) for i in range(100)]
    await asyncio.gather(*tasks)

_start = time.time()
await bad_loop()
_end = time.time()
print(f"bad done in {_end - _start:.5f} seconds")

_start = time.time()
await good_gather()
_end = time.time()
print(f"good done in {_end - _start:.5f} seconds")
```

```
bad done in 10.10158 seconds
good done in 0.10251 seconds
```

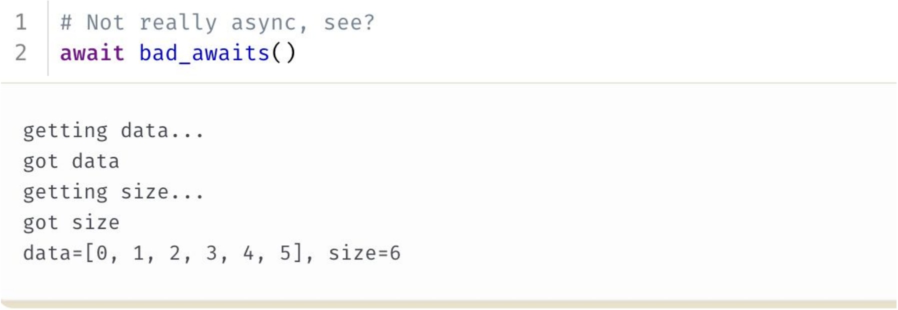


```python
tasks = set()  # for keeping a reference

async def something_i_need_now():
    await async_spam(0.1)
    print("here you go")

async def something_that_can_run_later():
    print("starting later")
    await async_spam(1)
    print("did the thing!")
    return 0

def mark_done(task):
    global tasks
    tasks.remove(task)
    print(f"Finished {task.get_name()}")

async def run():
    task = asyncio.create_task(something_that_can_run_later(), name="later_task")
    task.add_done_callback(mark_done)
    tasks.add(task)

    await something_i_need_now()

    while not task.done():
        await asyncio.sleep(0.5)

    if task.done():
        print(f"Result={task.result()}")

await run()
```

```
starting later
here you go
did the thing!
Finished later_task
Result=0
```

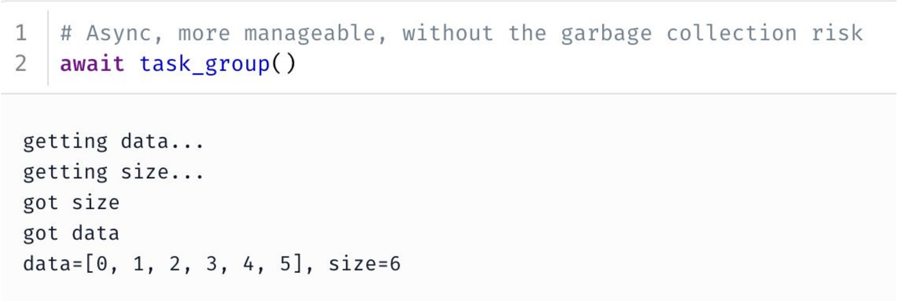


```python
async def get_data():
    print("getting data...")
    await async_spam(1)
    print("got data")
    return [0, 1, 2, 3, 4, 5]

async def get_size():
    print("getting size...")
    await async_spam(0.1)
    print("got size")
    return 6

async def bad_awaits():
    data = await get_data()
    size = await get_size()
    print(f"{data=}, {size=}")

async def task_group():
    async with asyncio.TaskGroup() as tg:
        t1 = tg.create_task(get_data())
        t2 = tg.create_task(get_size())

    data = t1.result()
    size = t2.result()

    print(f"{data=}, {size=}")

```



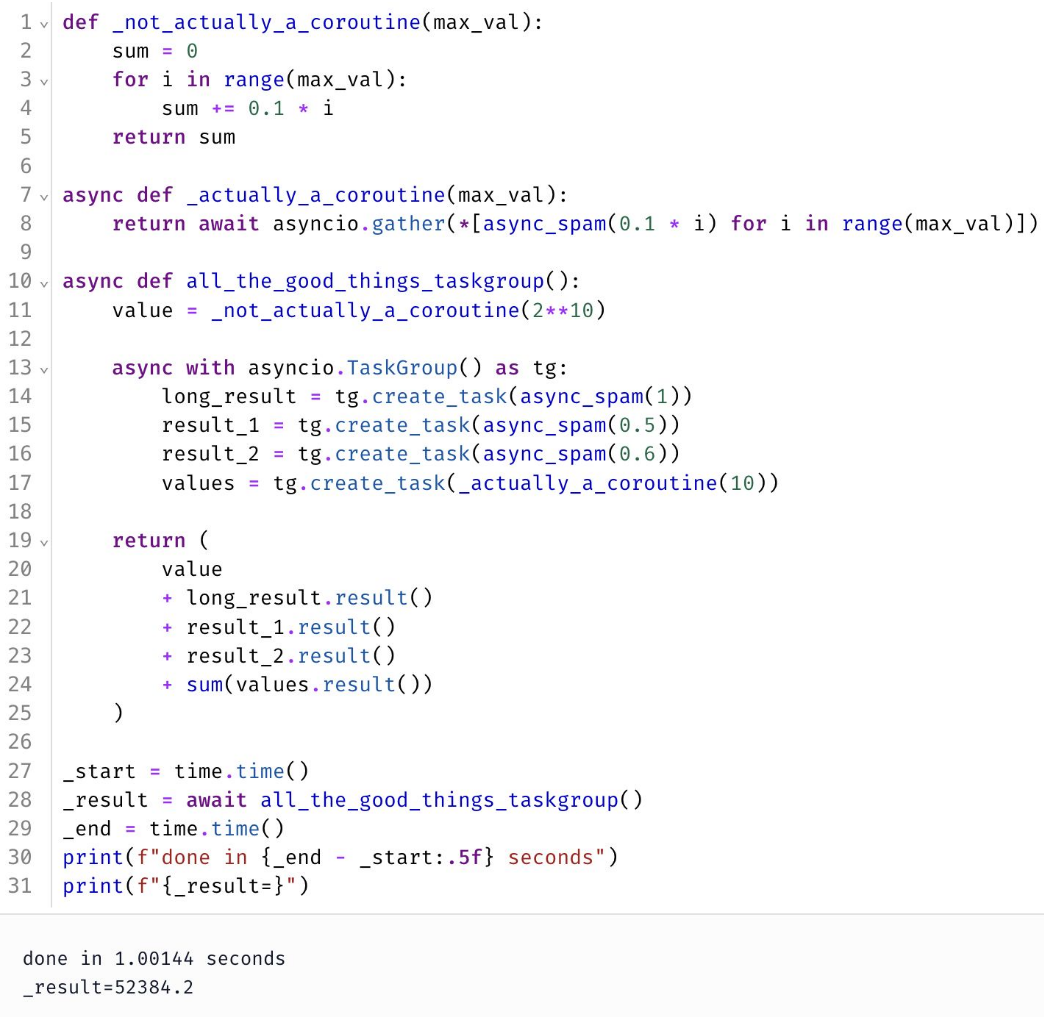
```python
# Not really async, see?
await bad_awaits()
```

```
getting data...
got data
getting size...
got size
data=[0, 1, 2, 3, 4, 5], size=6
```



```python
# Async, more manageable, without the garbage collection risk
await task_group()
```

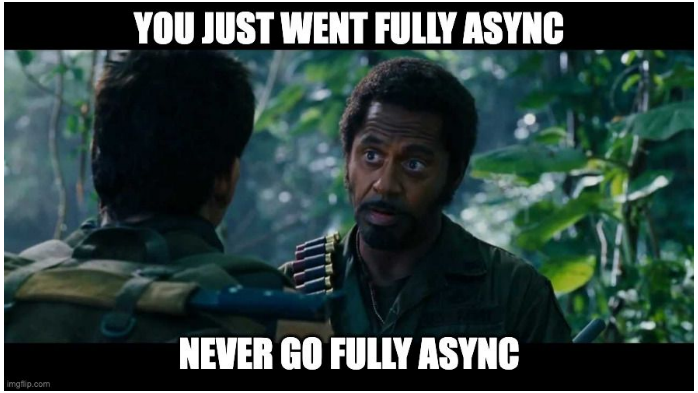
```
getting data...
getting size...
got size
got data
data=[0, 1, 2, 3, 4, 5], size=6
```



```python
def _not_actually_a_coroutine(max_val):
    sum = 0
    for i in range(max_val):
        sum += 0.1 * i
    return sum

async def _actually_a_coroutine(max_val):
    return await asyncio.gather(*[async_spam(0.1 * i) for i in range(max_val)])

async def all_the_good_things_taskgroup():
    value = _not_actually_a_coroutine(2**10)

    async with asyncio.TaskGroup() as tg:
        long_result = tg.create_task(async_spam(1))
        result_1 = tg.create_task(async_spam(0.5))
        result_2 = tg.create_task(async_spam(0.6))
        values = tg.create_task(_actually_a_coroutine(10))

    return (
        value
        + long_result.result()
        + result_1.result()
        + result_2.result()
        + sum(values.result())
    )

_start = time.time()
_result = await all_the_good_things_taskgroup()
_end = time.time()
print(f"done in {_end - _start:.5f} seconds")
print(f"{_result=}")
```

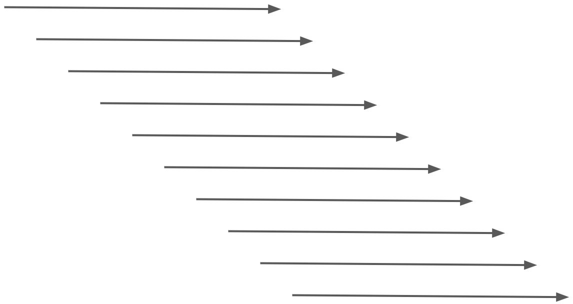
```
done in 1.00144 seconds
_result=52384.2
```

Concurrency Limits: When to *Not* Go Fully Async

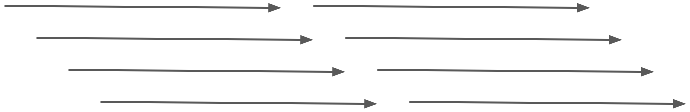


Did you (like me) get excited about `gather` and sent thousands of requests to an external API? You might just have crashed someone else's service (or, if it was your DB, your own) :(

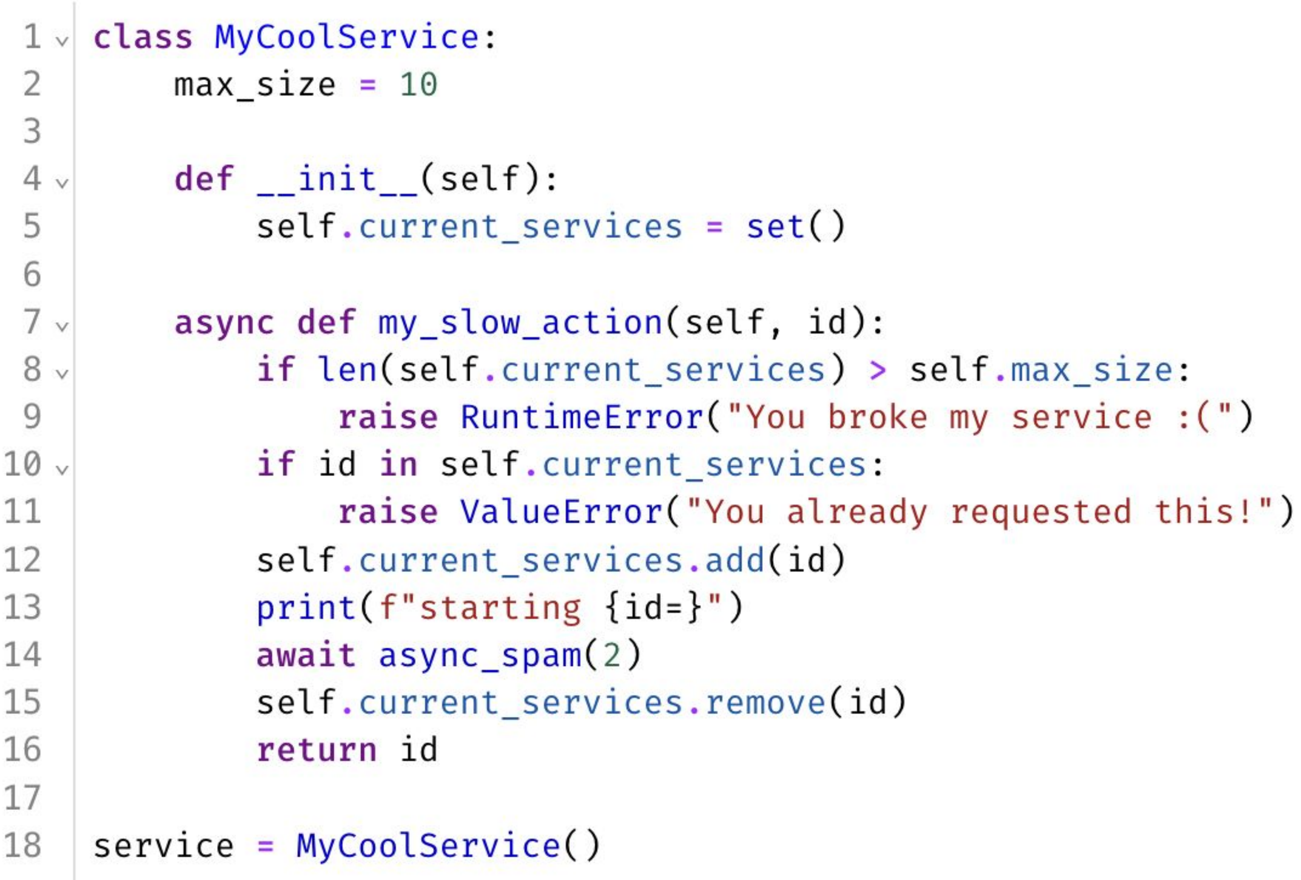
Sometimes, you *really* just want a couple of async requests made at the same time: Introducing `asyncio.Semaphore`



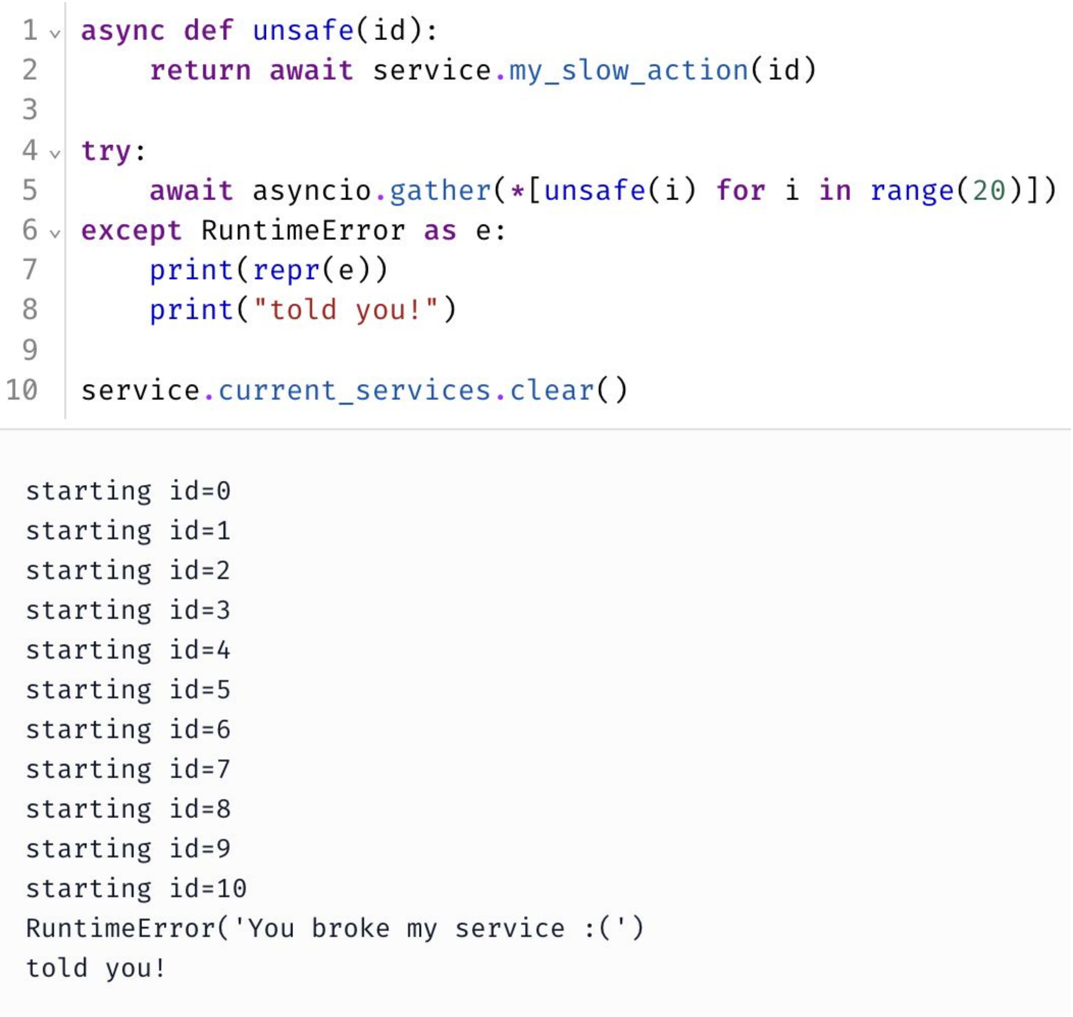
Send everything right away



Wait for some of the tasks to continue before sending more

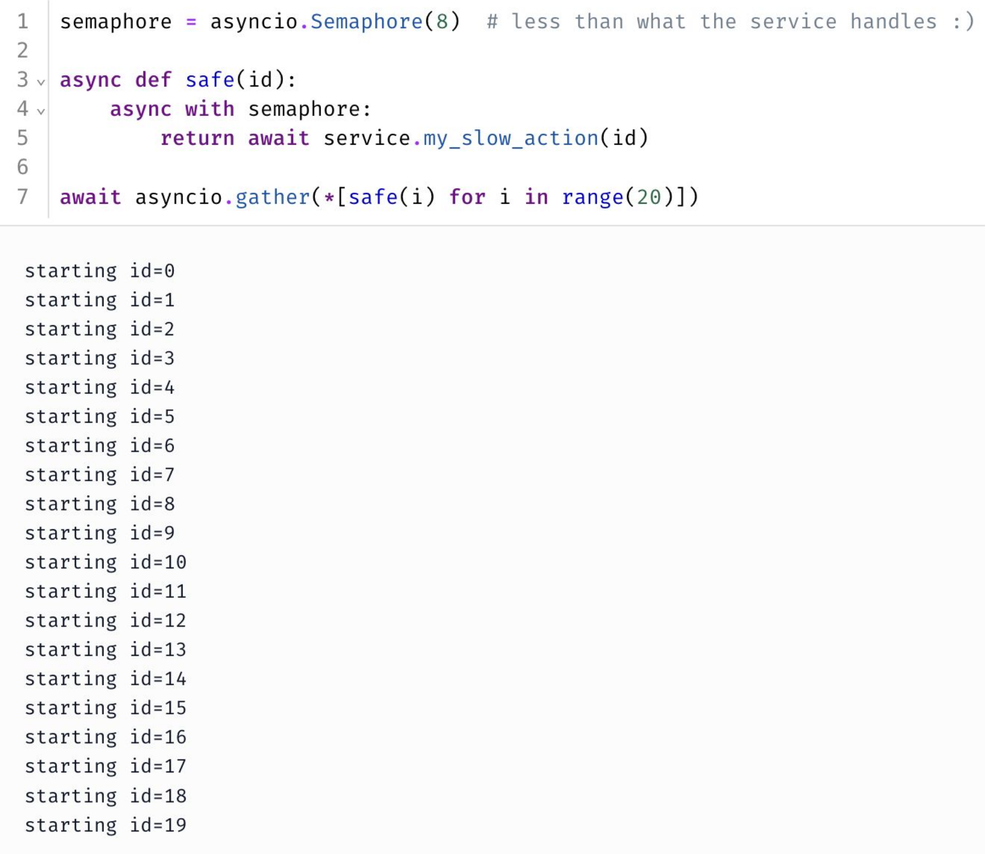


```python
class MyCoolService:
    max_size = 10

    def __init__(self):
        self.current_services = set()

    async def my_slow_action(self, id):
        if len(self.current_services) > self.max_size:
            raise RuntimeError("You broke my service :(")
        if id in self.current_services:
            raise ValueError("You already requested this!")
        self.current_services.add(id)
        print(f"starting {id=}")
        await async_spam(2)
        self.current_services.remove(id)
        return id

service = MyCoolService()
```



```python
async def unsafe(id):
    return await service.my_slow_action(id)

try:
    await asyncio.gather(*[unsafe(i) for i in range(20)])
except RuntimeError as e:
    print(repr(e))
    print("told you!")

service.current_services.clear()
```

```
starting id=0
starting id=1
starting id=2
starting id=3
starting id=4
starting id=5
starting id=6
starting id=7
starting id=8
starting id=9
starting id=10
RuntimeError('You broke my service :(')
told you!
```



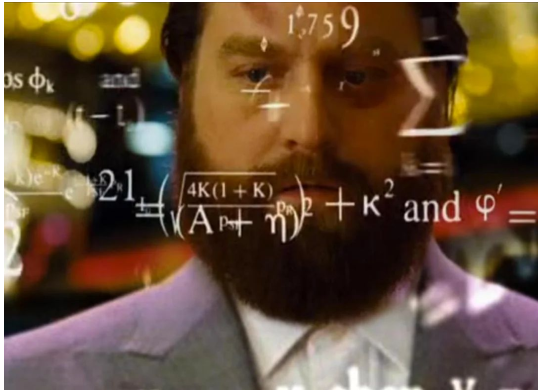
```python
semaphore = asyncio.Semaphore(8)  # less than what the service handles :)

async def safe(id):
    async with semaphore:
        return await service.my_slow_action(id)

await asyncio.gather(*[safe(i) for i in range(20)])
```

```
starting id=0
starting id=1
starting id=2
starting id=3
starting id=4
starting id=5
starting id=6
starting id=7
starting id=8
starting id=9
starting id=10
starting id=11
starting id=12
starting id=13
starting id=14
starting id=15
starting id=16
starting id=17
starting id=18
starting id=19
```

Other synchronization primitives:

- `Lock`: only one task can access a resource at a time
- `Event`: notify multiple tasks that a specific event has occurred
- `Condition`: Combines `Lock` and `Event`
- `Barrier`: Makes all tasks wait at a specific point (like a checkpoint)

Key Takeaways and Conclusion



“Faster execution ~~because~~ if your program doesn’t stay idle while waiting for a result”

- Asynchronous programming is more than just adding async/await! You make the asynchrony happen, not the keywords.
- Easy wins: `async def` only for coroutines, asynchronous libraries when in coroutines.
- Having await everywhere is a code smell: Be mindful of *why* you’re awaiting.
- Use the tools: `gather` and `TaskGroup` FTW
- Be curious! Stay on the lookout for improvements and better ways of doing things!

Thank you!

PYCON
COLOMBIA
2025

Slides, code and references:

jpvanegasc/async-python-talk