You Might Not Want Async

Quick Questions

- Concurrency with Python
- Threads
- Multi-processing
- Single-thread asynchrony
- asyncio

Me

- Call me TP
- Follow @uranusjr
- https://uranusjr.com

django girls

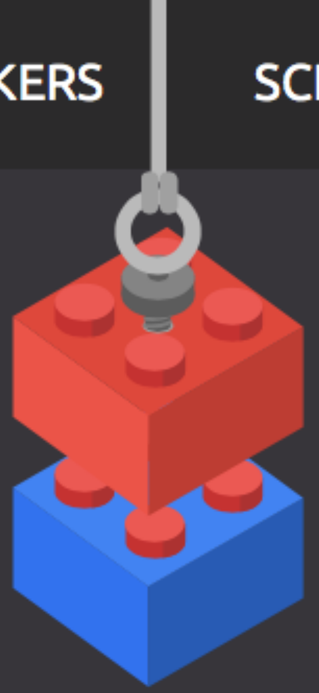
TAIPEI

http://macdown.uranusjr.com

www.Bimetek.com

OSDC

ACADEMIA SINICA TAIPEI, TAIWAN



中央研究院人文社會科學館
INTERNATIONAL CONFERENCE HALL, RESEARCH CENTER FOR HUMAN AND SOCIAL SCIENCE, ACADEMIA SINICA

04/11 - 04/12

OPEN SOURCE DEVELOPERS' CONFERENCE 2014

» REGISTER

NEWS

Apr 30 OSDC 2014 照片出爐

TWITTER

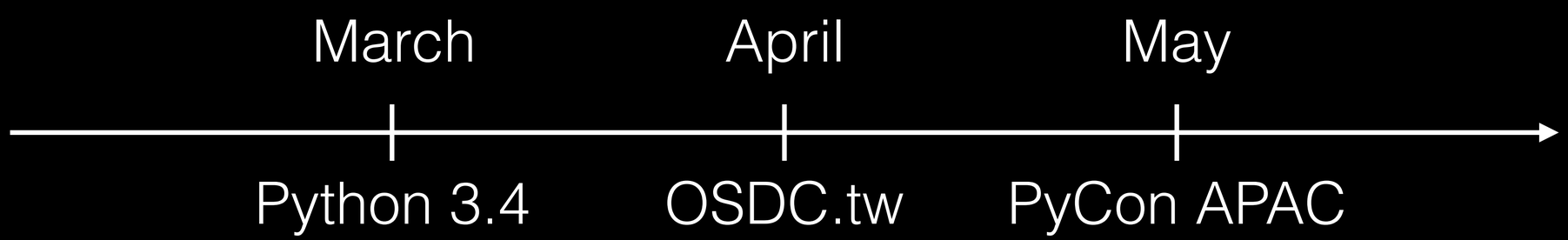
推文關於"#osdc.tw"

MAY 17-18 2014 in Taipei

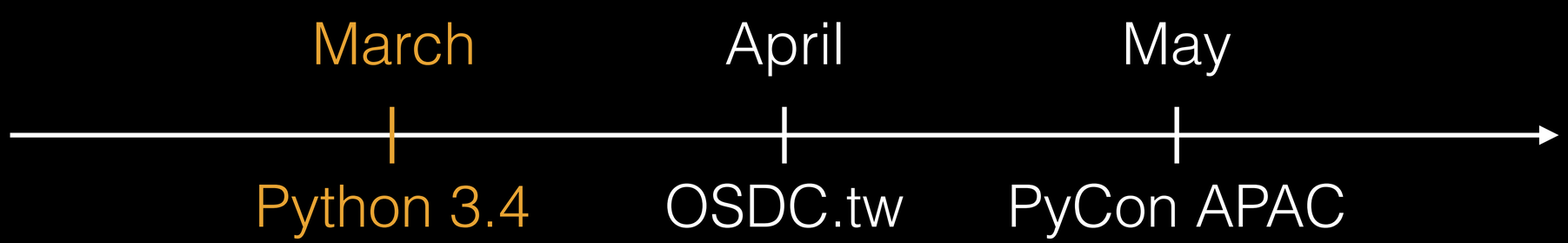
from everything import future


meh

2014

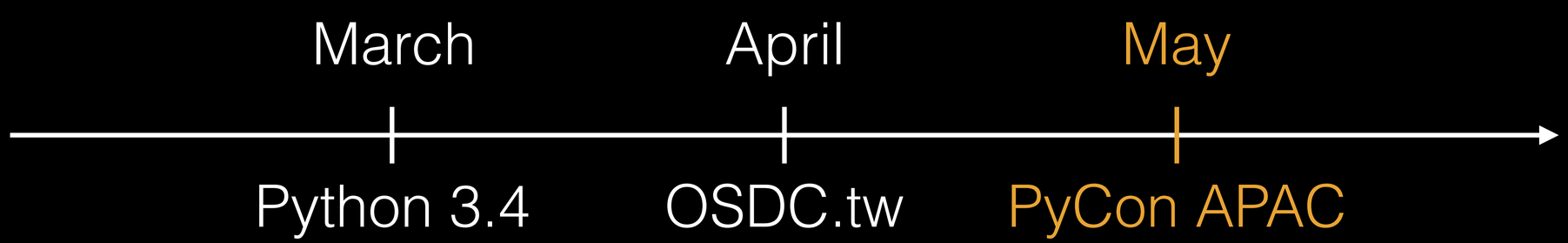


2014

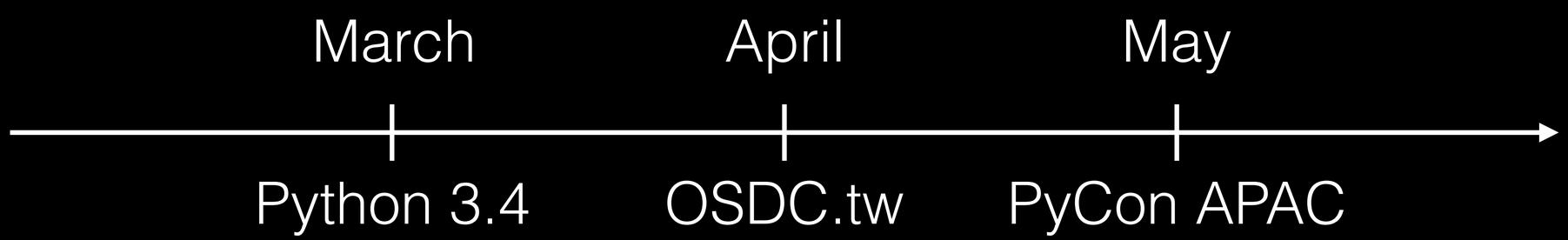


2014

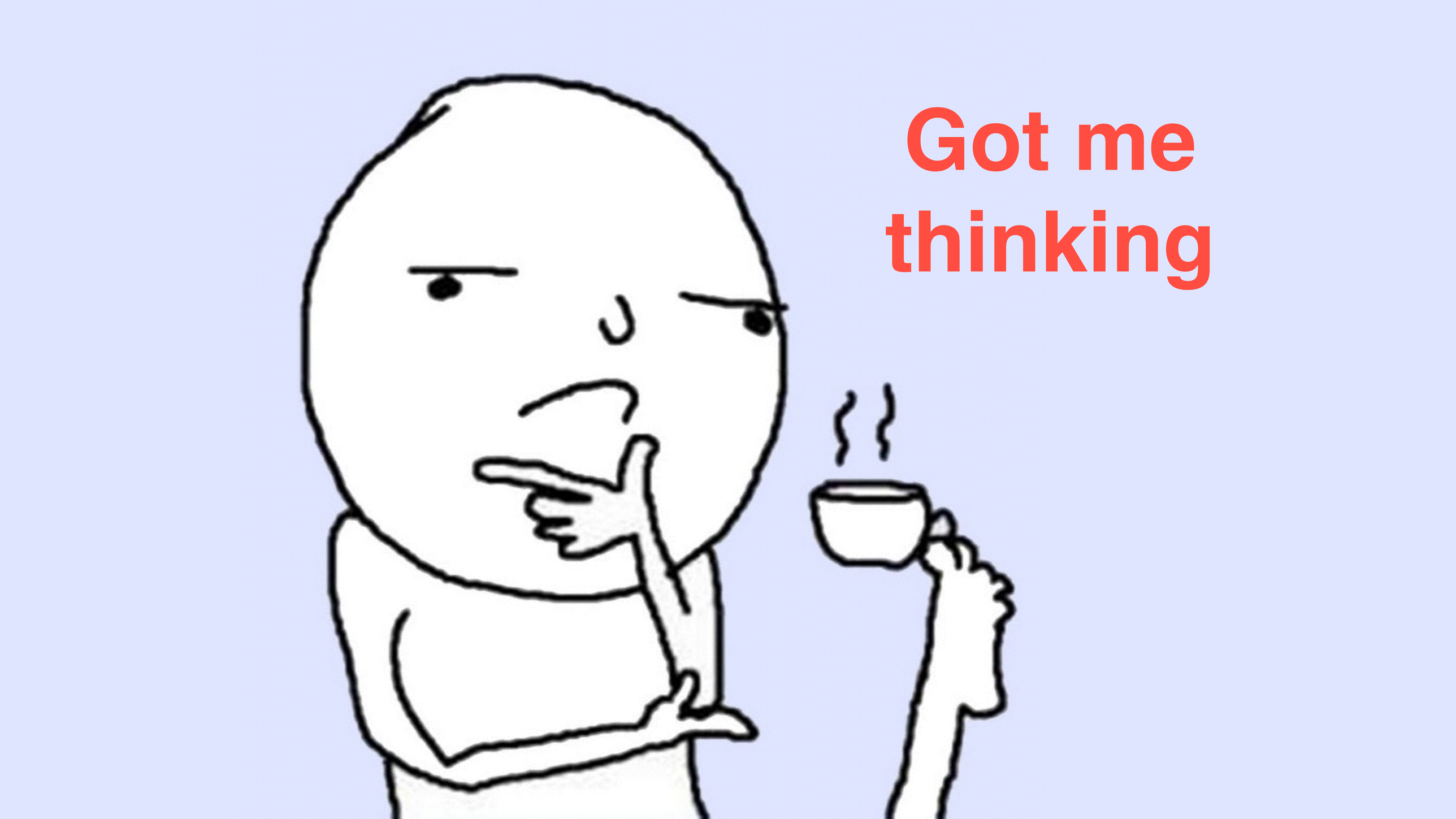
2014



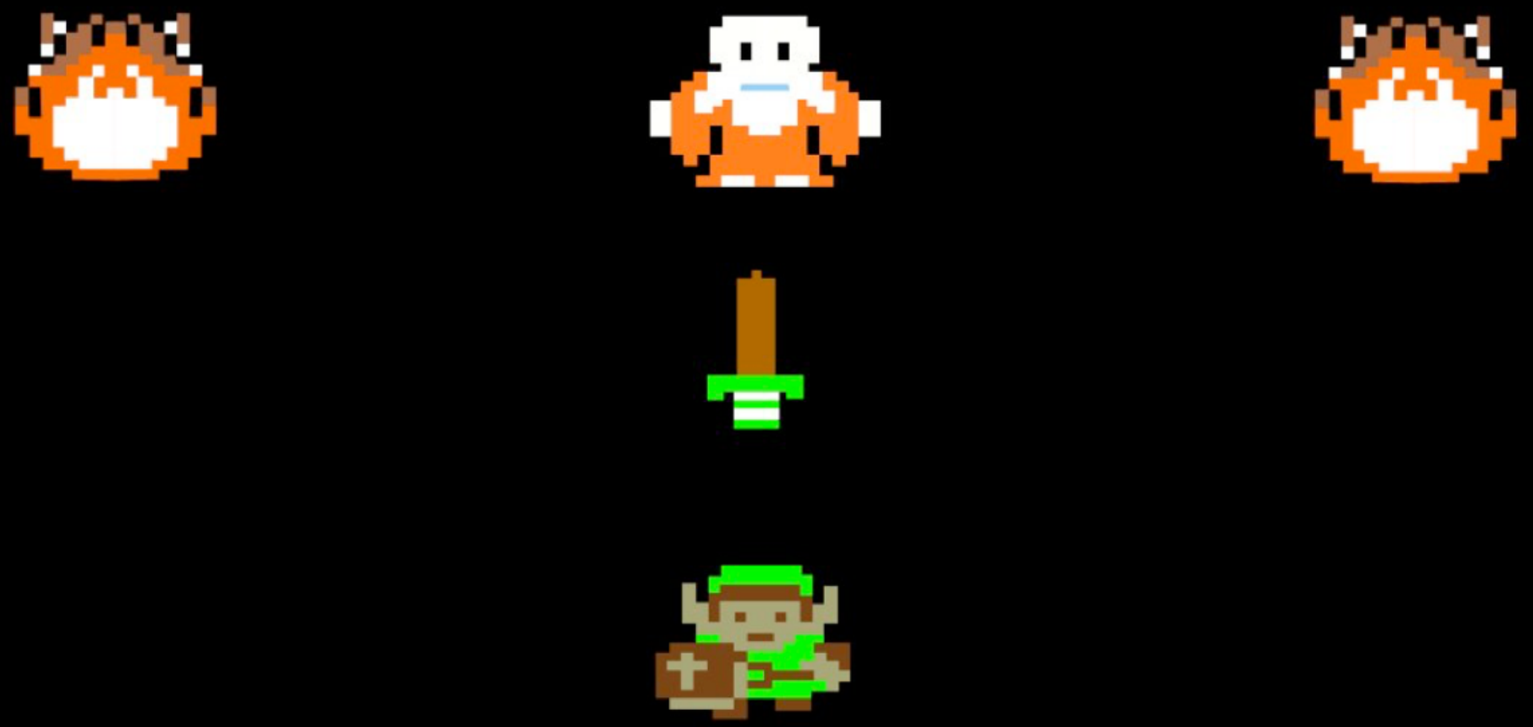
2014



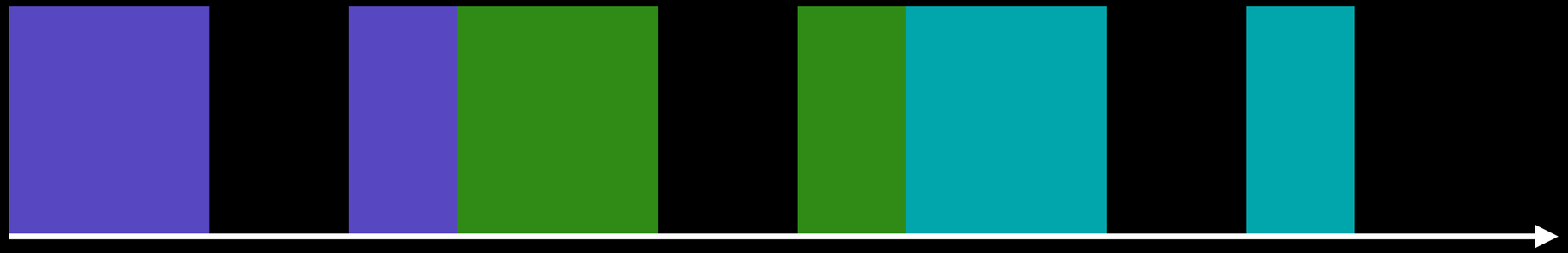

2016


Got me
thinking

IT'S DANGEROUS TO GO ALONE! TAKE THIS.

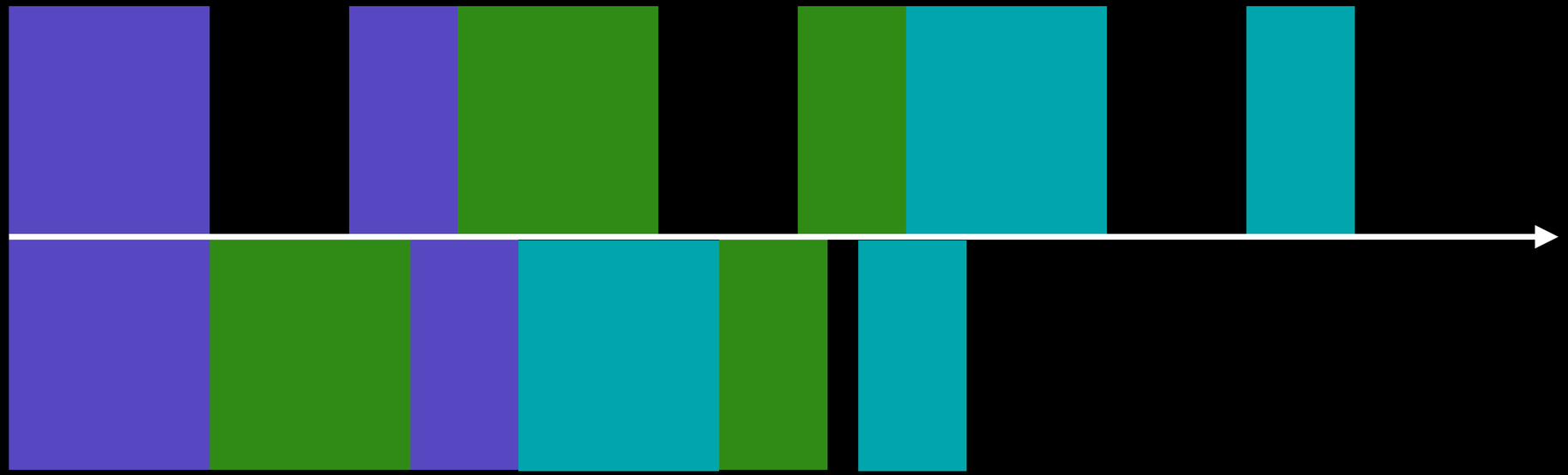


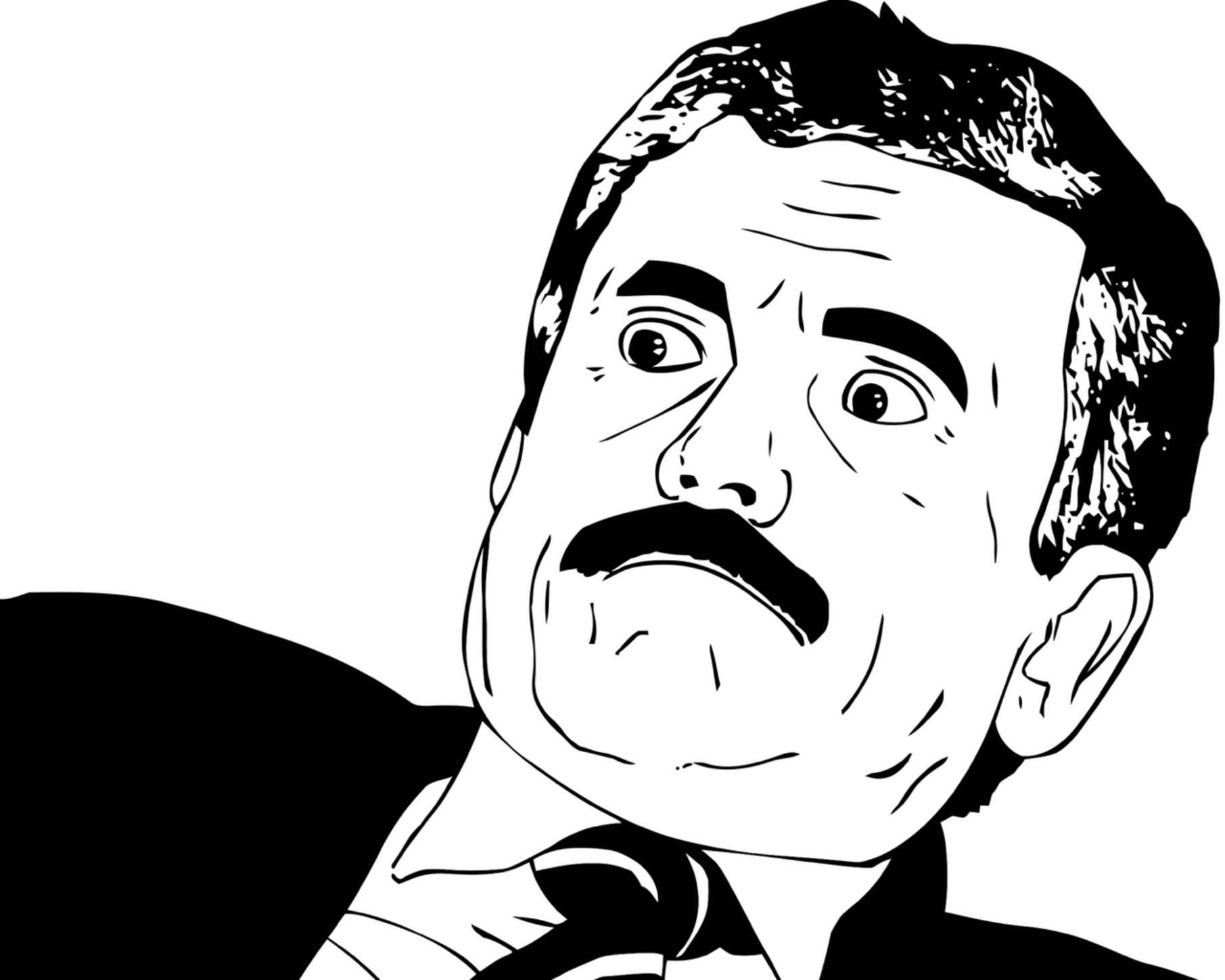
Synchronous



(Single-Threaded) Async

Sync vs. Async





Absolute magic

Before

After

Sync vs. Async

Not For You

- Infects the whole program
- Async is not parallelism
- Third party support

```
import sqlite3

def read_data(dbname):
    con = sqlite3.connect(dbname)
    cur = con.cursor()
    cur.execute('SELECT * FROM data OFFSET 0 LIMIT 1')
    data = cur.fetchone()
    cur.close()
    con.close()
    return data
```

```
import aioodbc

async def read_data(dbname):
    con = await aioodbc.connect(
        dsn='Driver=SQLite;Database={}'.format(dbname),
    )
    cur = await con.cursor()
    await cur.execute('SELECT * FROM data OFFSET 0 LIMIT 1')
    data = await cur.fetchone()
    await cur.close()
    await con.close()
    return data
```

```
from .db import read_data

def main():
    data = read_data('data.sqlite3')
    print(data)

main()
```

```
import asyncio
from .db import read_data

async def main():
    data = await read_data('db.sqlite3')
    print(data)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

U+1F937 **SHRUG** (Unicode 9.0)


RELAX PEOPLE
I'M JUST GETTING STARTED

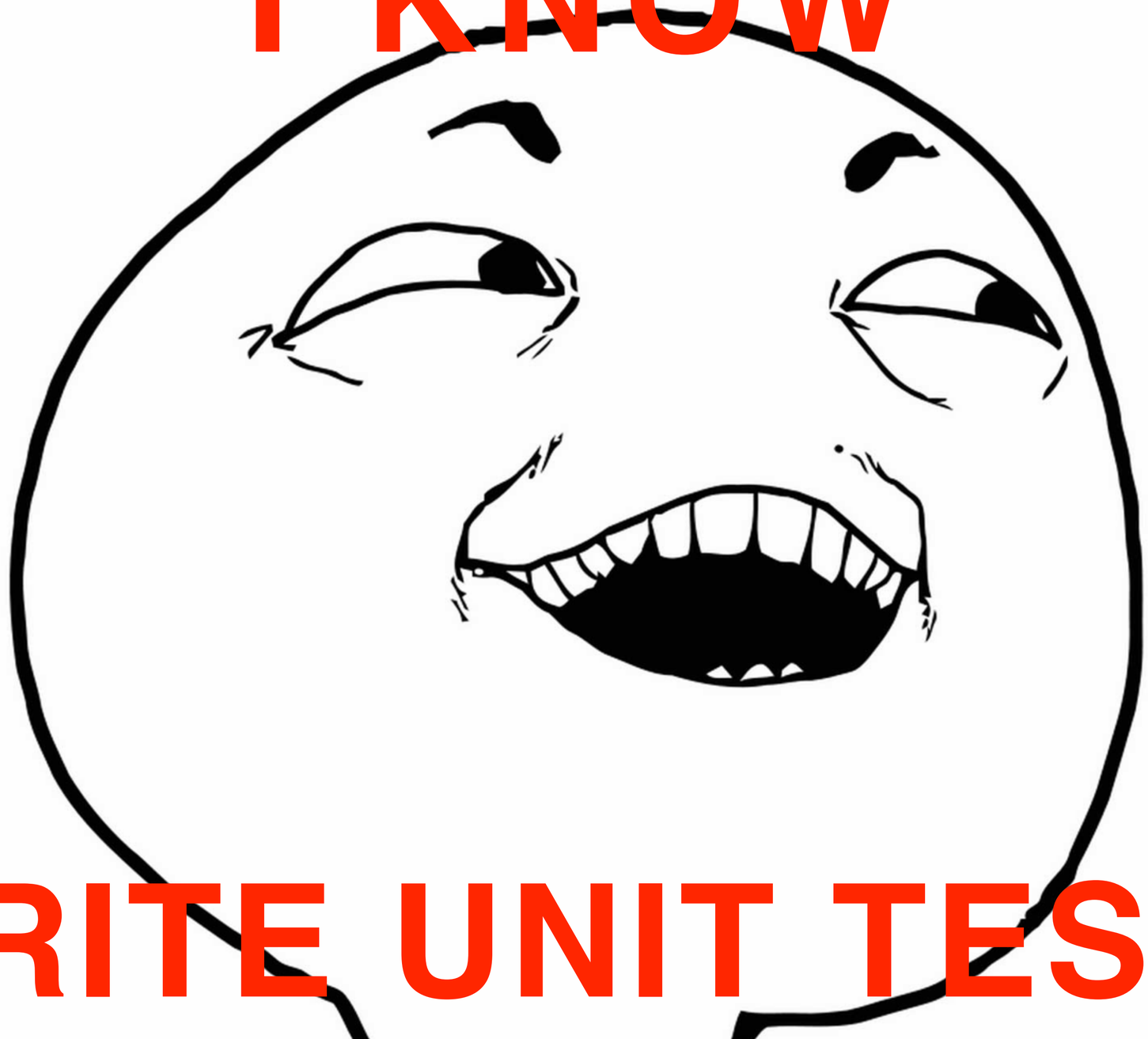
```
import asyncio
from .db import read_data

async def main():
    data = read_data('db.sqlite3')
    print(data)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

```
(demo) $ python demo.py
<coroutine object read_data at 0x10e2d1f10>
```

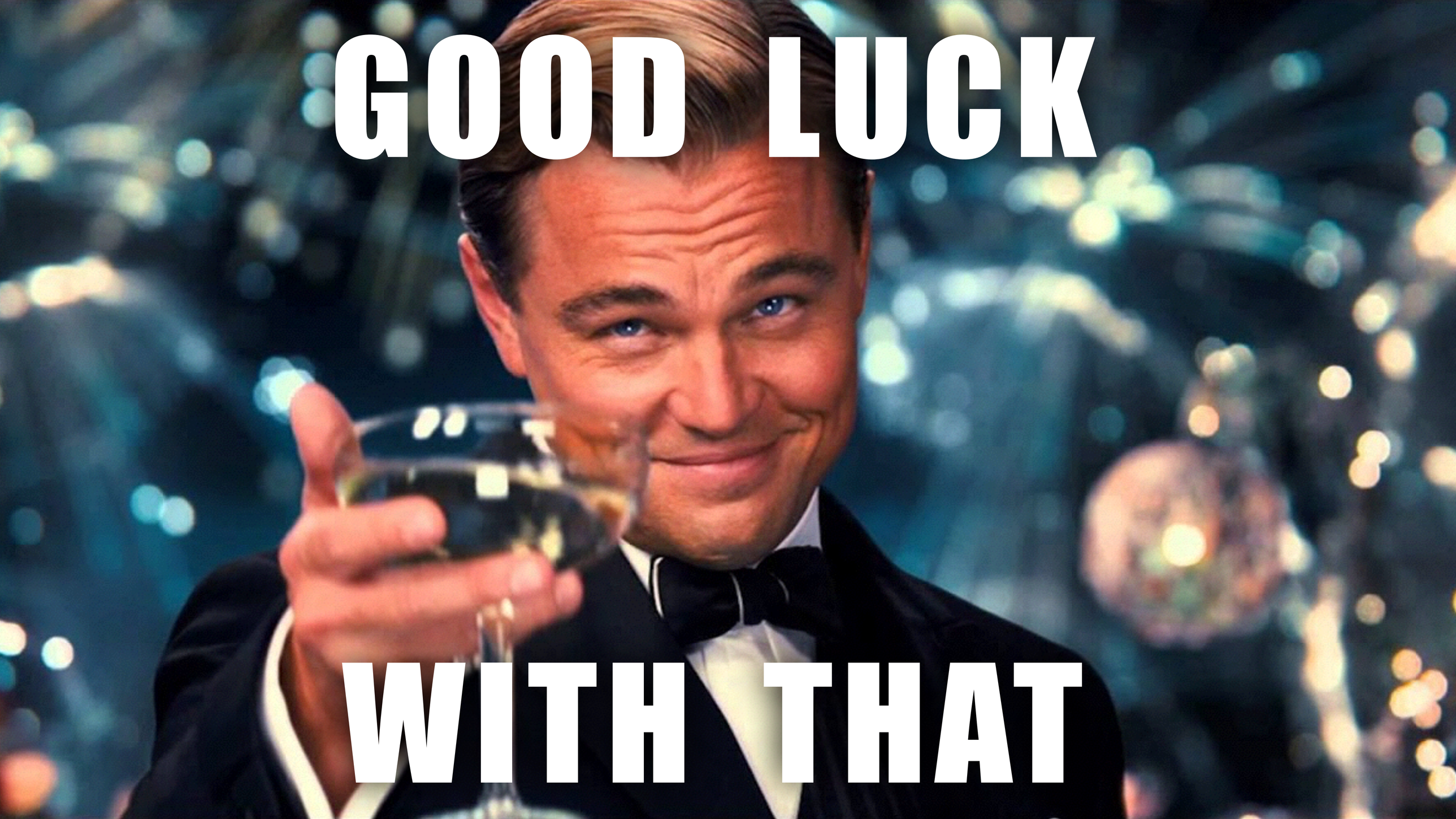
```
import asyncio
from .db import read_data

async def main():
    data = read_data('db.sqlite3')
    print(data)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

```
import asyncio
from .db import read_data

async def main():
    data = await read_data('db.sqlite3')
    print(data)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

I KNOW



WRITE UNIT TESTS


GOOD LUCK
WITH THAT

How to test Python 3.4 asyncio code?

31

10

What's the best way to write unit tests for code using the Python 3.4 `asyncio` library? Assume I want to test a TCP client (`SocketConnection`):

```
import asyncio
import unittest

class TestSocketConnection(unittest.TestCase):
    def setUp(self):
        self.mock_server = MockServer("localhost", 1337)
        self.socket_connection = SocketConnection("localhost", 1337)

    @asyncio.coroutine
    def test_sends_handshake_after_connect(self):
        yield from self.socket_connection.connect()
        self.assertTrue(self.mock_server.received_handshake())
```

When running this test case with the default test runner, the test will always succeed as the method executes only up until the first `yield from` instruction, after which it returns before executing any assertions. This causes tests to always succeed.

Is there a prebuilt test runner that is able to handle asynchronous code like this?

Eventually Correct

A. Jesse Jiryu Davis
@jessejiryudavis
MongoDB

```
import asyncio
import time

async def do_something():
    print('Before', time.monotonic())
    await asyncio.sleep(2)
    print('After ', time.monotonic())
```

```
>>> await do_something()
Before 199208.190632853
After  199210.192092425
```

```
import unittest


class MyTestCase(unittest.TestCase):
    async def test_do_something(self):
        await do_something()


if __name__ == '__main__':
    unittest.main()
```

```
(demo) $ python tests.py
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

```
(demo) $ python tests.py
.
----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

What Happened?

- `unittest` does not know about `asyncio`
- Coroutine methods are executed “normally”
- Called, but not executed (`awaited`)


WHAT IF I TOLD YOU
THIS IS GONNA BE TOUGH

```
import asyncio
import functools

def asynchronous(func):

    @functools.wraps(func)
    def asynchronous_inner(*args, **kwargs):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        try:
            loop.run_until_complete(func(*args, **kwargs))
        finally:
            loop.close()

    return asynchronous_inner
```

```
import asyncio
import functools

def asynchronous(func):

    @functools.wraps(func)
    def asynchronous_inner(*args, **kwargs):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        try:
            loop.run_until_complete(func(*args, **kwargs))
        finally:
            loop.close()

    return asynchronous_inner
```

```
import unittest


class MyTestCase(unittest.TestCase):
    @asynchronous
    async def test_do_something(self):
        await do_something()


if __name__ == '__main__':
    unittest.main()
```

```
(demo) $ python tests.py
Before 51728.420457105
After  51730.424515145
.
----------------------------------------------------------------------
Ran 1 test in 2.006s

OK
```

Tips

- Beware of warning output
  - Especially if you redirect
- Add coverage report to *testing code*
- Consider `pip install asynctest`

Or use pytest instead

```
import time

async def test_do_something():
    before = time.monotonic()
    await do_something()
    delta_t = time.monotonic() - before
    assert -0.01 < delta_t < 0.01
```

```
(demo) $ py.test tests.py
========================= test session starts ==========================
platform darwin -- Python 3.5.1, pytest-2.9.1, py-1.4.31,
pluggy-0.3.1
collected 0 items / 1 errors

============================== ERRORS ==============================
_____________________ ERROR collecting tests.py_____________________
>       for i, x in enumerate(self.obj()):
E       TypeError: 'coroutine' object is not iterable

python3.5/site-packages/_pytest/python.py:765: TypeError
====================== 1 error in 0.15 seconds =======================
```

pytest-asyncio 0.3.0

Pytest support for asyncio.

Downloads ↓

pytest-asyncio is an Apache2 licensed library, written in Python, for testing asyncio code with pytest.

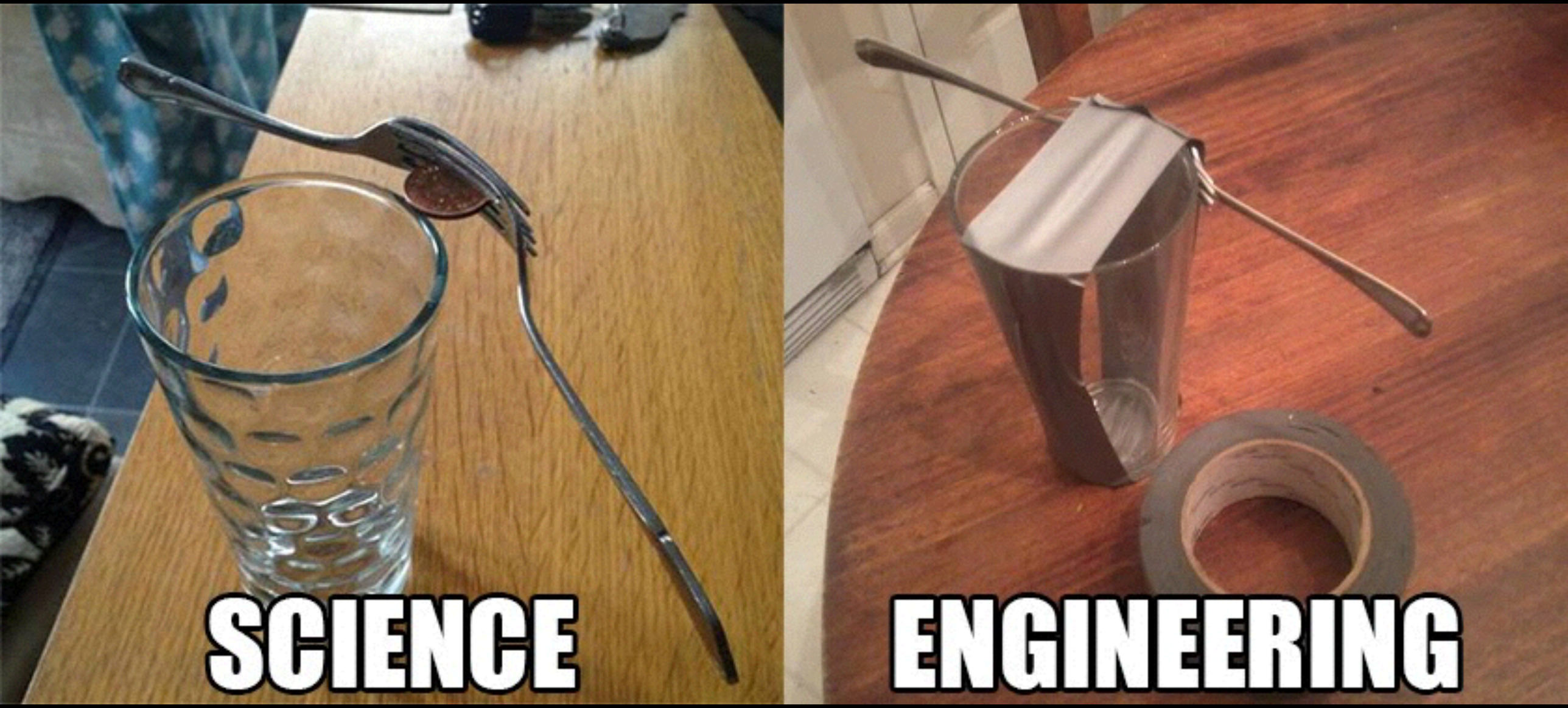
asyncio code is usually written in the form of coroutines, which makes it slightly more difficult to test using normal testing tools. pytest-asyncio provides useful fixtures and markers to make testing easier.

```
@pytest.mark.asyncio
async def test_some_asyncio_code():
    res = await library.do_something()
    assert b'expected result' == res
```

or, if you're using the pre-Python 3.5 syntax:

```
import time
import pytest

@pytest.mark.asyncio
async def test_do_something(capsys):
    before = time.monotonic()
    await do_something()
    delta_t = time.monotonic() - before
    assert -0.01 < delta_t < 0.01
```


But Wait...
THERE'S MORE!!!


SCIENCE
ENGINEERING

asyncio

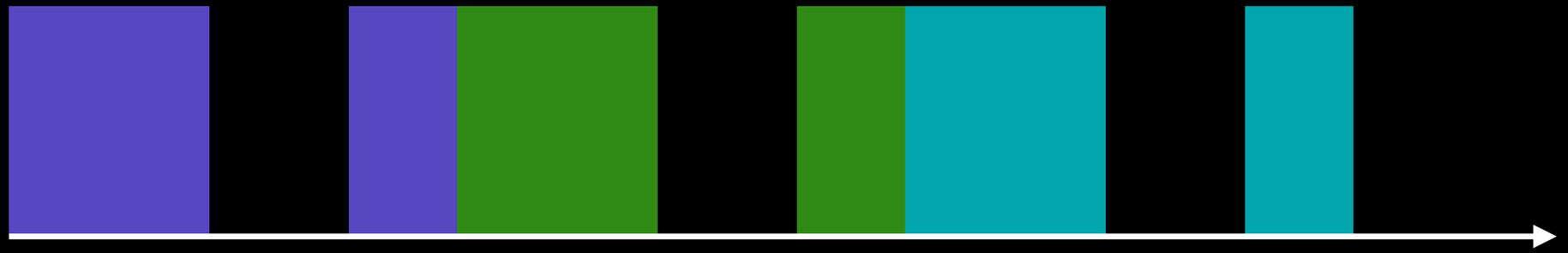
- Boilerplate
- Error-prone
- Immature API (I think)

```
import requests


def collect_contents(urls):
    contents = []
    for url in urls:
        resp = requests.get(url)
        if resp.status_code != 200:
            continue
        content = resp.text
        contents.append(content)
    return contents
```

```python
import aiohttp

async def collect_contents(urls):
    contents = []
    with aiohttp.ClientSession() as session:
        for url in urls:
            async with session.get(url) as rest:
                if resp.status != 200:
                    continue
                content = await resp.text()
                contents.append(content)
    return contents
```

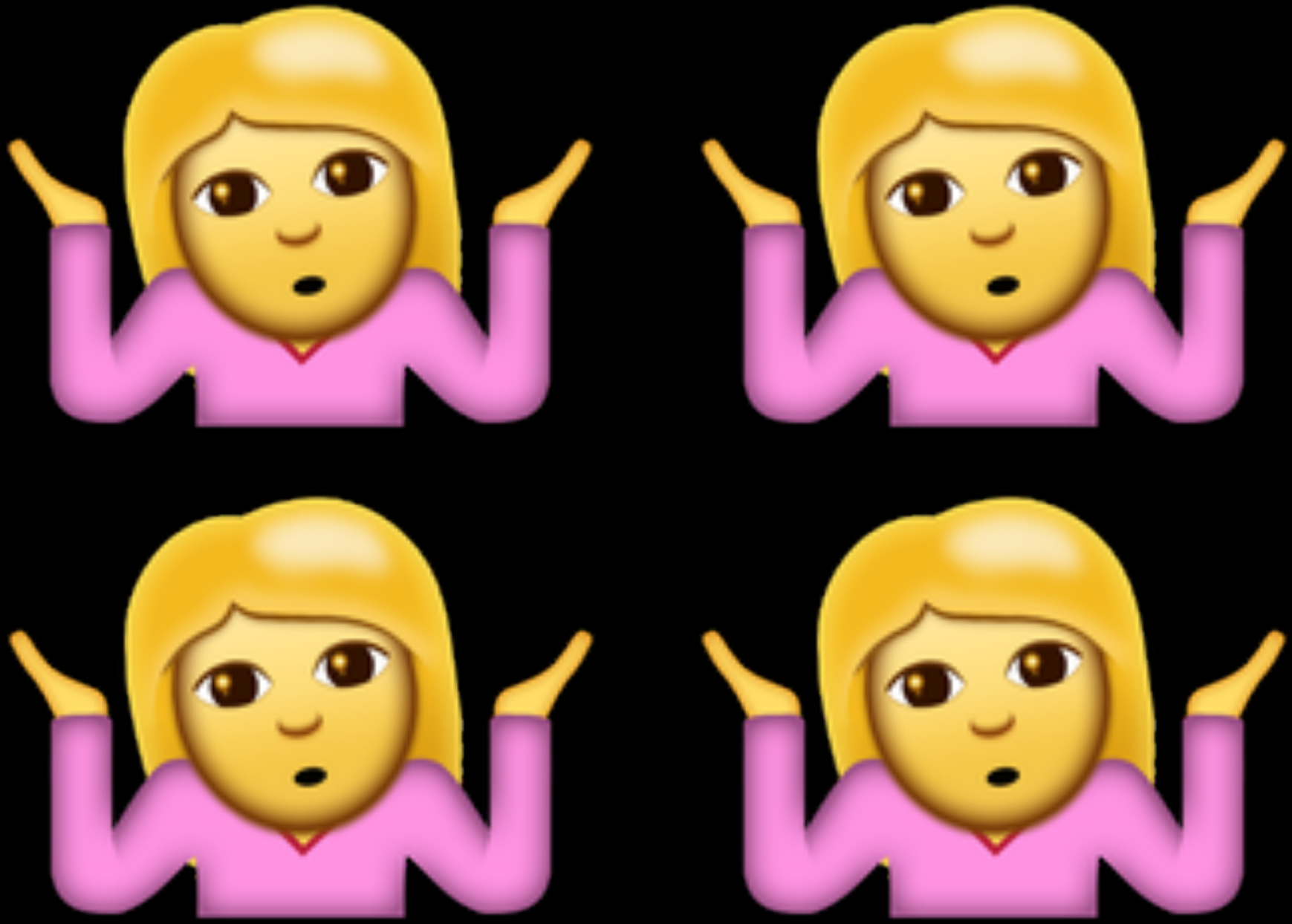

ONE DOES NOT SIMPLY
ADD “ASYNC” AND “AWAIT”

```
import aiohttp


async def collect_contents(urls):
    contents = []
    with aiohttp.ClientSession() as session:
        for url in urls:
            async with session.get(url) as rest:
                if resp.status != 200:
                    continue
                content = await resp.text()
                contents.append(content)
    return contents
```



```
import aiohttp


async def collect_contents(urls):
    contents = []
    with aiohttp.ClientSession() as session:
        for url in urls:
            async with session.get(url) as rest:
                if resp.status != 200:
                    continue
                content = await resp.text()
                contents.append(content)
    return contents
```

```
import asyncio
import aiohttp

async def collect_contents(urls):
    coroutines = []
    with aiohttp.ClientSession() as session:
        for url in urls:
            async with session.get(url) as rest:
                if resp.status != 200:
                    continue
                coroutines.append(resp.text())
        contents = await asyncio.gather(*coroutines)
    return contents
```



coroutine asyncio.**wait**(*futures*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the Futures and coroutine objects given by the sequence *futures* to complete. Coroutines will be wrapped in Tasks. Returns two sets of `Future`: (done, pending).

The sequence *futures* must not be empty.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants of the `concurrent.futures` module:

asyncio.**gather**(**coros_or_futures*, *loop=None*, *return_exceptions=False*)

Return a future aggregating results from the given coroutine objects or futures.

All futures must share the same event loop. If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival). If *return_exceptions* is True, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

Cancellation: if the outer Future is cancelled, all children (that have not completed yet) are also cancelled. If any child is cancelled, this is treated as if it raised `CancelledError` – the outer Future is *not* cancelled in this case. (This is to prevent the cancellation of one child to cause other children to be cancelled.)

coroutine asyncio.**wait**(*futures*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the Futures and coroutine objects given by the sequence *futures* to complete. Coroutines will be wrapped in Tasks. Returns two sets of Future: (done, pending).

The sequence *futures* must not be empty.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or None, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants of the concurrent.futures module:

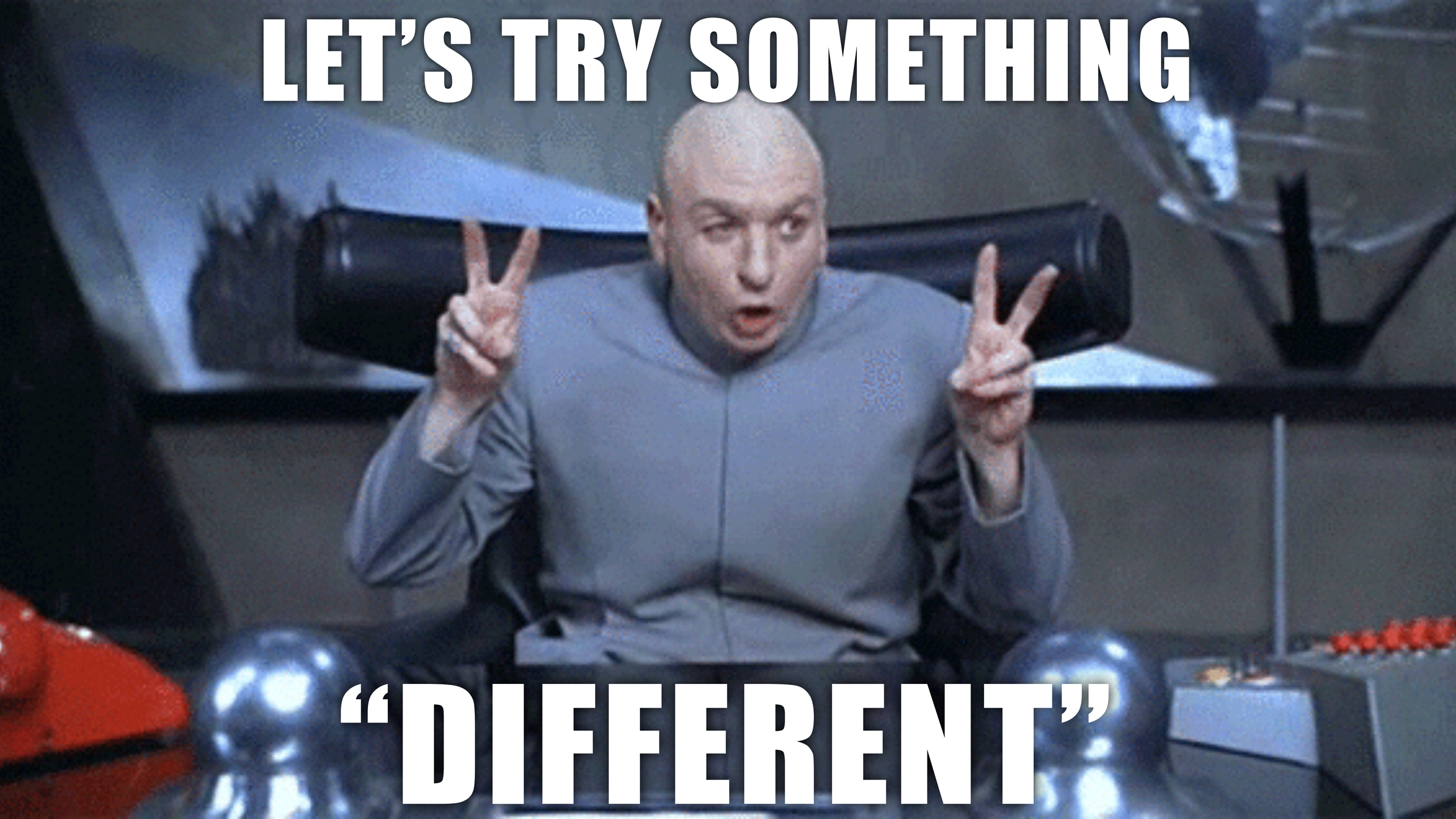
asyncio.**gather**(**coros_or_futures*, *loop=None*, *return_exceptions=False*)

Return a future aggregating results from the given coroutine objects or futures.

All futures must share the same event loop. If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival). If *return_exceptions* is True, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

Cancellation: if the outer Future is cancelled, all children (that have not completed yet) are also cancelled. If any child is cancelled, this is treated as if it raised CancelledError – the outer Future is *not* cancelled in this case. (This is to prevent the cancellation of one child to cause other children to be cancelled.)


WHY!!!???


LET'S TRY SOMETHING
"DIFFERENT"

Alternatives

- `concurrent` & `multiprocessing`
- *Greenlets*
  - Similar idea, but less infectious
  - C extension
- `threading`
  - Standard I/O

greenlet: Lightweight concurrent programming

Motivation

The "greenlet" package is a spin-off of Stackless, a version of CPython that supports micro-threads called "tasklets". Tasklets run pseudo-concurrently (typically in a single or a few OS-level threads) and are synchronized with data exchanges on "channels".

A "greenlet", on the other hand, is a still more primitive notion of micro-thread with no implicit scheduling; coroutines, in other words. This is useful when you want to control exactly when your code runs. You can build custom scheduled micro-threads on top of greenlet; however, it seems that greenlets are useful on their own as a way to make advanced control flow structures. For example, we can recreate generators; the difference with Python's own generators is that our generators can call nested functions and the nested functions can yield values too. (Additionally, you don't need a "yield" keyword. See the exam

http://greenlet.readthedocs.io

GEVENT

MEH


ROUTINES
SPACE RANGER
LIGHTYEAR
ROUTINES EVERYWHERE

```
package main
import ("fmt"; "time")


func doSomething() {
    time.Sleep(2 * time.Second)
    fmt.Println(time.Now(), "Slept")
}


func main() {
    doSomething()
    fmt.Println(time.Now(), "OK")
}
```

```
00:00:00 Slept
00:00:00 OK
```

```go
package main
import ("fmt"; "time")


func doSomething() {
    time.Sleep(2 * time.Second)
    fmt.Println(time.Now(), "Slept")
}


func main() {
    go doSomething()
    fmt.Println(time.Now(), "OK")
}
```

```
00:00:00 OK
```

```
package main
import ("fmt"; "time")

var sem = make(chan bool)
func doSomething() {
    time.Sleep(2 * time.Second)
    fmt.Println(time.Now(), "Slept")
    sem ← true
}

func main() {
    go doSomething()
    fmt.Println(time.Now(), "OK")
    ←sem
}
```

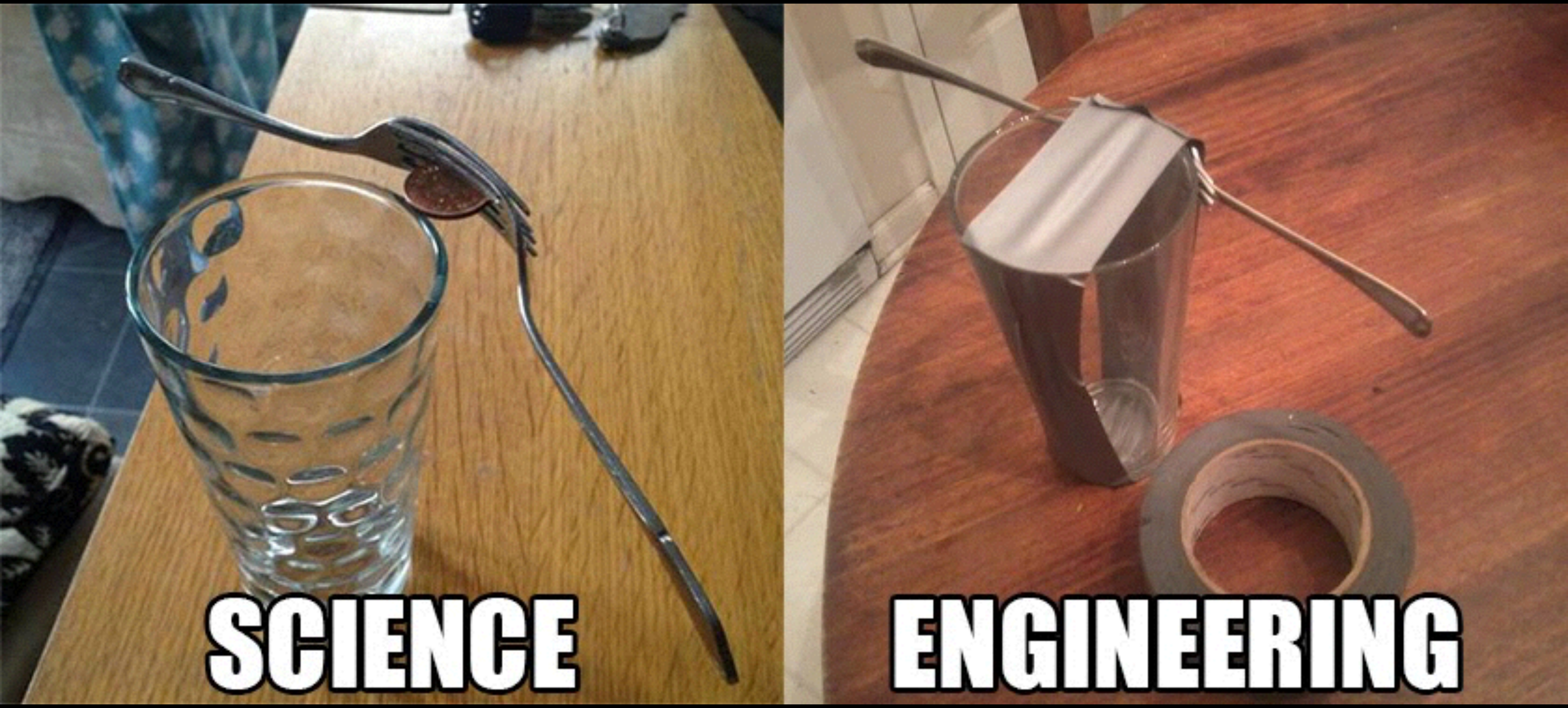
```
00:00:00 OK
00:00:02 Slept
```

The Go Model

- ~~Boilerplate~~
- ~~Error-prone~~
- ~~Immature API (I think)~~

```
"""Do something. Synchronous version."""

import datetime
import time


def do_something():
    time.sleep(2)
    print(datetime.datetime.now(), 'Slept')


do_something()
print(datetime.datetime.now(), 'Done')
```

```
"""Do something. Asynchronous version."""

import asyncio
import datetime


async def do_something():
    await asyncio.sleep(2)
    print(datetime.datetime.now(), 'Slept')


loop = asyncio.get_event_loop()
task = loop.create_task(do_something())
print(datetime.datetime.now(), 'Done')
loop.run_until_complete(asyncio.wait([task]))
loop.close()
```


SCIENCE
ENGINEERING

```
"""Do something. Synchronous version."""


import datetime
import time


def do_something():
    time.sleep(2)
    print(datetime.datetime.now(), 'Slept')


do_something()
print(datetime.datetime.now(), 'Done')
```

```
"""What if I can just write this?"""

import asyncio
import datetime
import time


async def do_something():
    await time.sleep(2)
    print(datetime.datetime.now(), 'Slept')


await do_something()
print(datetime.datetime.now(), 'Done')
asyncio.run_event_loop()
```

I know, it’s not really possible.

At least we can dream.

Or wait until Python 6.0?

But Seriously

- Asynchrony is not the silver bullet
- It makes you jump through loops
- There are alternatives
- Fingers crossed

Recap

- Rant
- Moar rant
- Susceptible advice

But Seriously

- Asynchrony is not the silver bullet
- It makes you jump through loops
- There are alternatives

IT'S DANGEROUS TO GO ALONE! TAKE THIS.