EuroPython 2018

asyncio today & tomorrow

Hi, I'm Yury

- Python Core Developer since 2013
- PEPs 362, 492, 525, 530, 550, 567
- asyncio maintainer
- uvloop, asyncpg
- EdgeDB
- Twitter, GitHub: **@1st1**

Part I

brief asyncio history

Brief asyncio History

Python 3.3

- Python 3.3 ~2013

  Guido works on Tulip

- Tulip is a reference implementation of PEP 3156

- Inspired by Twisted & co

- Becomes part of Python 3.4

Python 3.4

- Provisional
- Low-level APIs: Protocols, Transports, Futures, and callbacks
- Coroutines via `yield from`
- High-level APIs: Streams, Subprocesses, etc

Brief asyncio History

Python 3.5

- 🎉 async / await 🎉
- asyncio is still provisional
- A bunch of new APIs
- uvloop
- New framework: **Curio**

  hm, what can we learn from it?

Brief asyncio History

Python 3.6

- No longer provisional :(
- async generators
- We've fixed `get_event_loop()` 🎉
- New low-level APIs...
- New framework: **Trio**

  hm, what can we learn from it?

Python 3.7

- Context Variables—**`contextvars`**
- asyncio's own code uses async/await
- `asyncio.run()` [thanks, Curio!]
- Grab bag: sendfile, start TLS, create_task(), get_running_loop(), BufferedProtocol, etc
- Better third-party event loops support

Part II

async / await

async / await

asyncio layers

asyncio.run()

asyncio.gather()

asyncio.create_task()

asyncio.sleep()

streams API

normal

---

hardcore

loop.*()

protocols & transports

asyncio.Future

async / await

asyncio.run()

```
import asyncio

async def main():
    ...

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

async / await

asyncio.run()

```
import asyncio

async def main():
    ...

asyncio.run(main())
```

async / await

asyncio.run()

```
def run(main, *, debug=False):
    loop = events.new_event_loop()
    try:
        events.set_event_loop(loop)
        loop.set_debug(debug)
        return loop.run_until_complete(main)
    finally:
        try:
            _cancel_all_tasks(loop)
            loop.run_until_complete(
                loop.shutdown_asyncgens())
        finally:
            events.set_event_loop(None)
            loop.close()
```

```
def _cancel_all_tasks(loop):
    to_cancel = tasks.all_tasks(loop)
    if not to_cancel:
        return

    for task in to_cancel:
        task.cancel()

    loop.run_until_complete(
        tasks.gather(*to_cancel, loop=loop,
                     return_exceptions=True))

    for task in to_cancel:
        if task.cancelled():
            continue
        if task.exception() is not None:
            loop.call_exception_handler({
                ...
            })
```

async / await

use it!

- with `asyncio.run()` you don't need the loop
- have just one entry point
- use async / await for everything
- don't pass a reference to the loop anywhere

async / await

serve_forever()

```
import asyncio

async def handle_client(rd, wr):
    # handle client

loop = asyncio.get_event_loop()
server = loop.run_until_complete(
    asyncio.start_server(
        handle_client,
        '127.0.0.1', 8888,
        loop=loop))
```

```
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(
    server.wait_closed())
loop.close()
```

async / await

serve_forever()

```python
import asyncio

async def handle_client(reader, writer):
    # handle client


async def main():
    srv = await asyncio.start_server(
        handle_client, '127.0.0.1', 8888)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())
```

async / await

get_running_loop()

```
import asyncio

async def main():
    # do things

loop = asyncio.get_event_loop()
loop.create_task(main())
loop.add_signal_handler(...)
loop.run_forever()
```

async / await

get_running_loop()

```
import asyncio

async def main():
    loop = asyncio.get_running_loop()
    loop.add_signal_handler(...)

    # do things

asyncio.run(main())
```

async / await

don'ts

- don't use `@coroutine`,
  we will remove it soon-ish

- don't use low-level APIs (futures,
  call_soon(), call_later(), transports,
  protocols, event loop)
  unless you *have* to.

Part III

good async / await code

good code

What code is good?

- That you can write quick? **(subjective)**
- Maintainable? **(subjective)^2**
- Beautiful? **(subjective)$^{3^3}$**
- Robust?
- Fast?

Let's talk about fast & robust

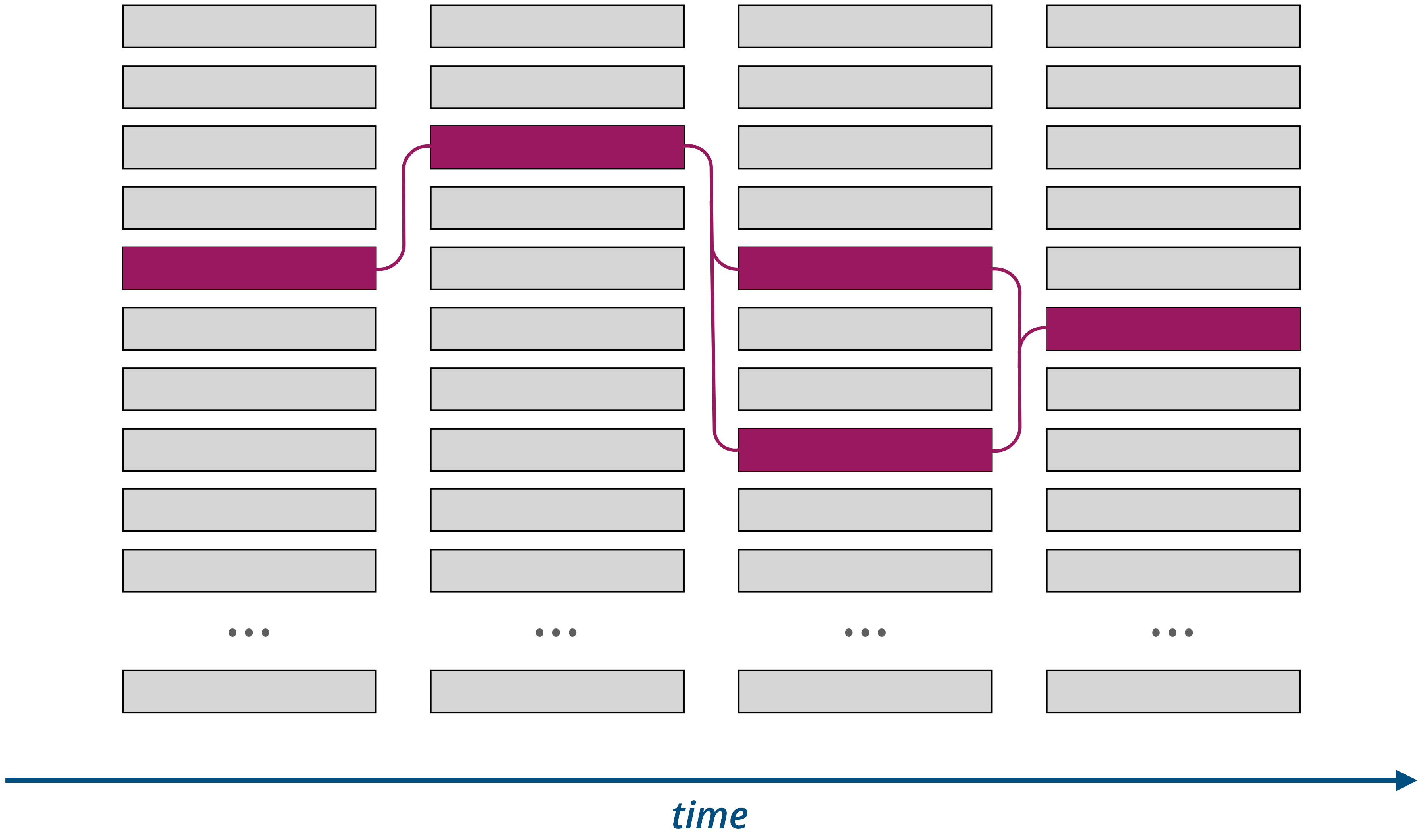
monitoring

in

production

good code

Is async code good?



good code

contextvars

- PEP 550, 4 different revisions
- PEP 567
- ~900 emails on python-ideas and python-dev

good code

contextvars

- *warning:* it's magic 🎩
- shipped with Python 3.7
- standard library module
- full *asyncio* support
- *decimal* context uses it

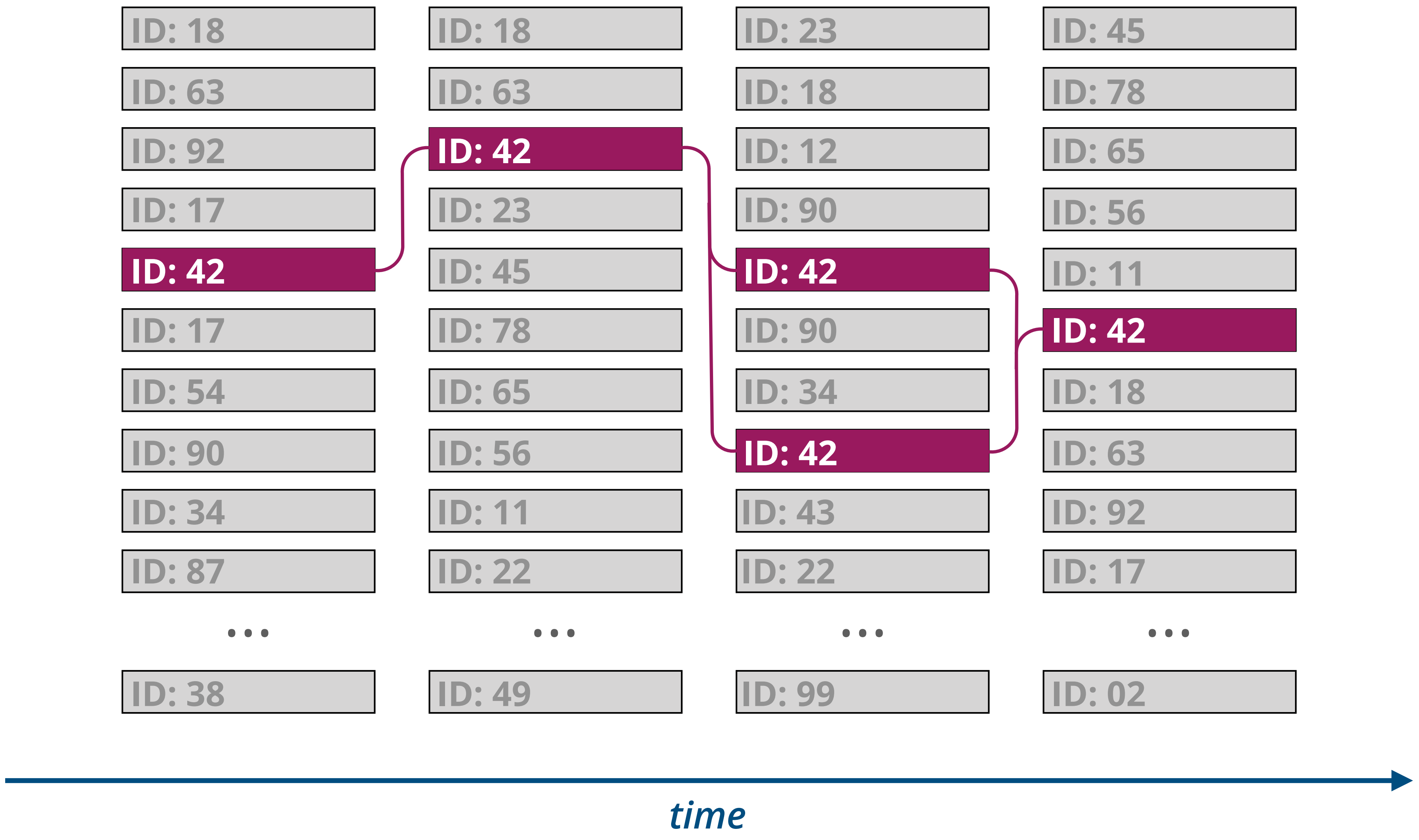
good code

contextvars

```
import contextvars

task_id = contextvars.ContextVar(
    'Task tracking ID')

task_id.set(unique_number)

task_id.get()
```

good code

contextvars



good code

Use contextvars for

- **monitoring:**
  e.g. how long some operations take
- **localization:**
  e.g. current language for HTTP request
- **security:**
  e.g. current user or permissions
- **debug:** 🚀 🚀 🚀
- **execution context:**
  e.g. decimal context & numpy error context

Part IV

what's next

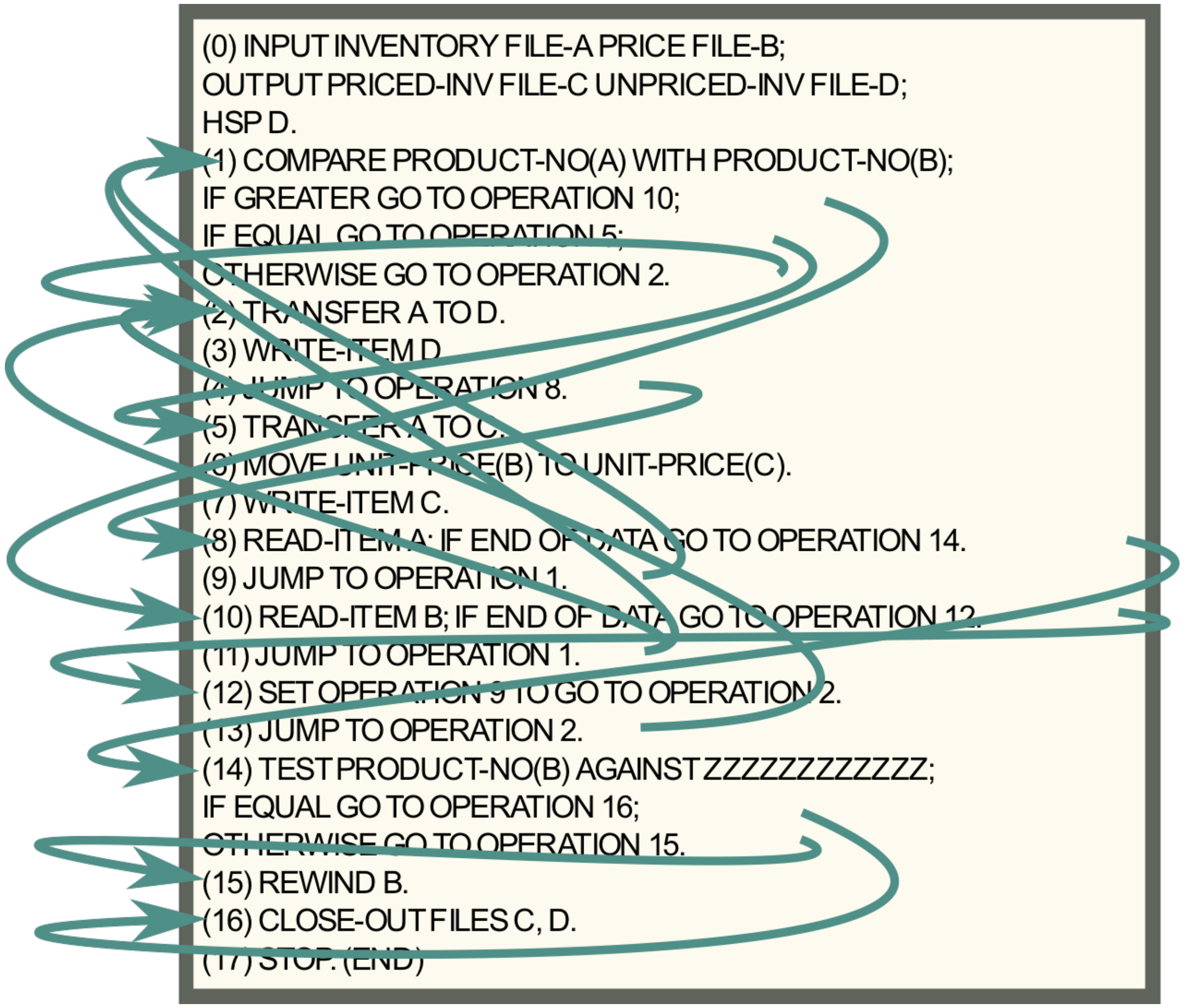
what's next

Let's talk about Trio

- new library by Nathaniel J. Smith
- designed from scratch
- incompatible with asyncio
- hard focus on usability
- got many things right!
- youtu.be/oLkfnc_UMcE

what's next

back in 1958...

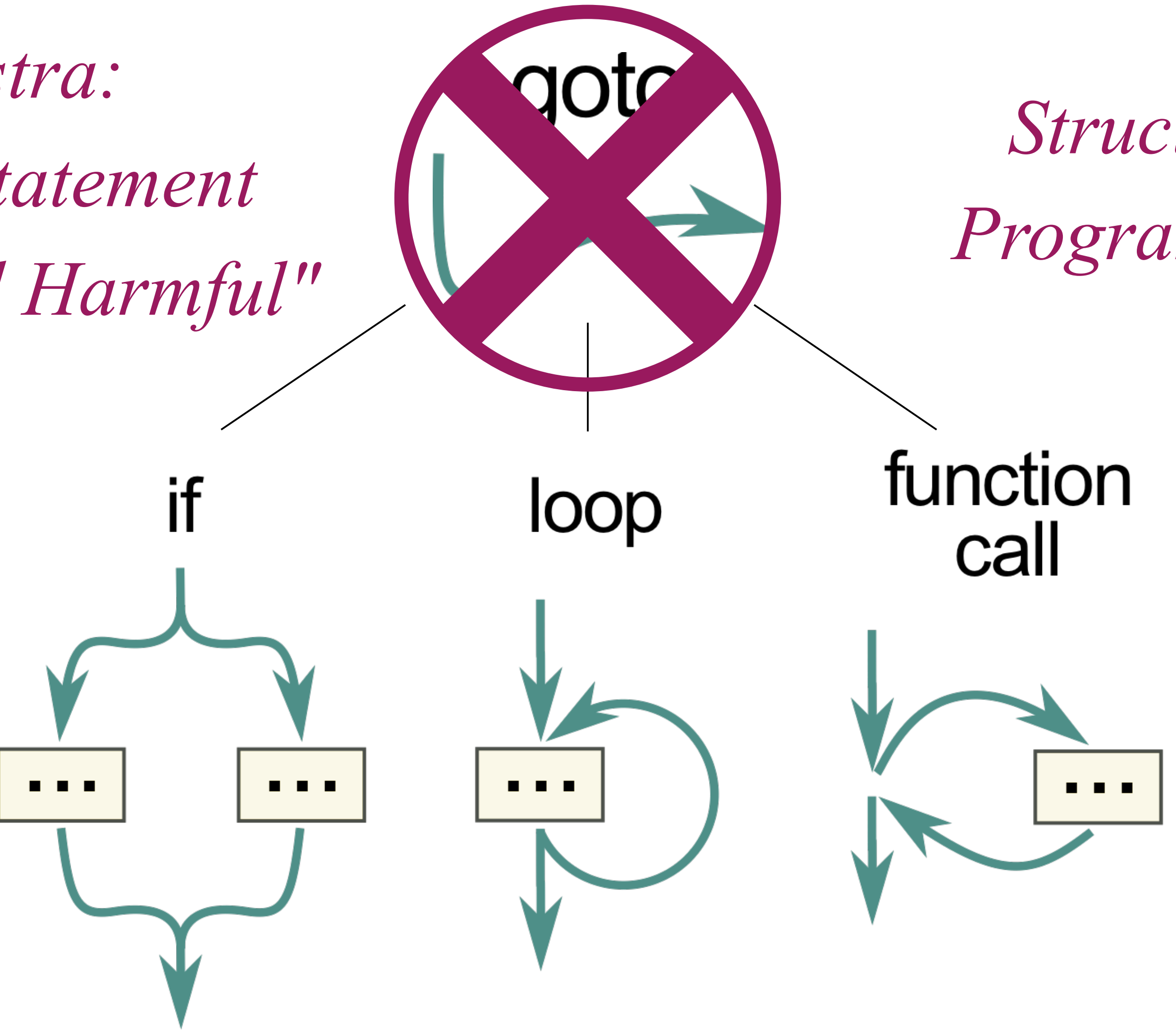


```
(0) INPUT INVENTORY FILE-A PRICE FILE-B;
OUTPUT PRICED-INV FILE-C UNPRICED-INV FILE-D;
HSP D.
(1) COMPARE PRODUCT-NO(A) WITH PRODUCT-NO(B);
IF GREATER GO TO OPERATION 10;
IF EQUAL GO TO OPERATION 5;
OTHERWISE GO TO OPERATION 2.
(2) TRANSFER A TO D.
(3) WRITE-ITEM D.
(4) JUMP TO OPERATION 8.
(5) TRANSFER A TO C.
(6) MOVE UNIT-PRICE(B) TO UNIT-PRICE(C).
(7) WRITE-ITEM C.
(8) READ-ITEM A; IF END OF DATA GO TO OPERATION 14.
(9) JUMP TO OPERATION 1.
(10) READ-ITEM B; IF END OF DATA GO TO OPERATION 12.
(11) JUMP TO OPERATION 1.
(12) SET OPERATION 9 TO GO TO OPERATION 2.
(13) JUMP TO OPERATION 2.
(14) TEST PRODUCT-NO(B) AGAINST ZZZZZZZZZZZZ;
IF EQUAL GO TO OPERATION 16;
OTHERWISE GO TO OPERATION 15.
(15) REWIND B.
(16) CLOSE-OUT FILES C, D.
(17) STOP. (END)
```

what's next

10 years later, 1968

Dijkstra:
"Go To Statement Considered Harmful"



Structured Programming

what's next

Back to Trio: nurseries

```
async def child():
    ...

async def parent():
    async with trio.open_nursery() as nursery:
        # Make two concurrent calls to child()
        nursery.start_soon(child)
        nursery.start_soon(child)
```

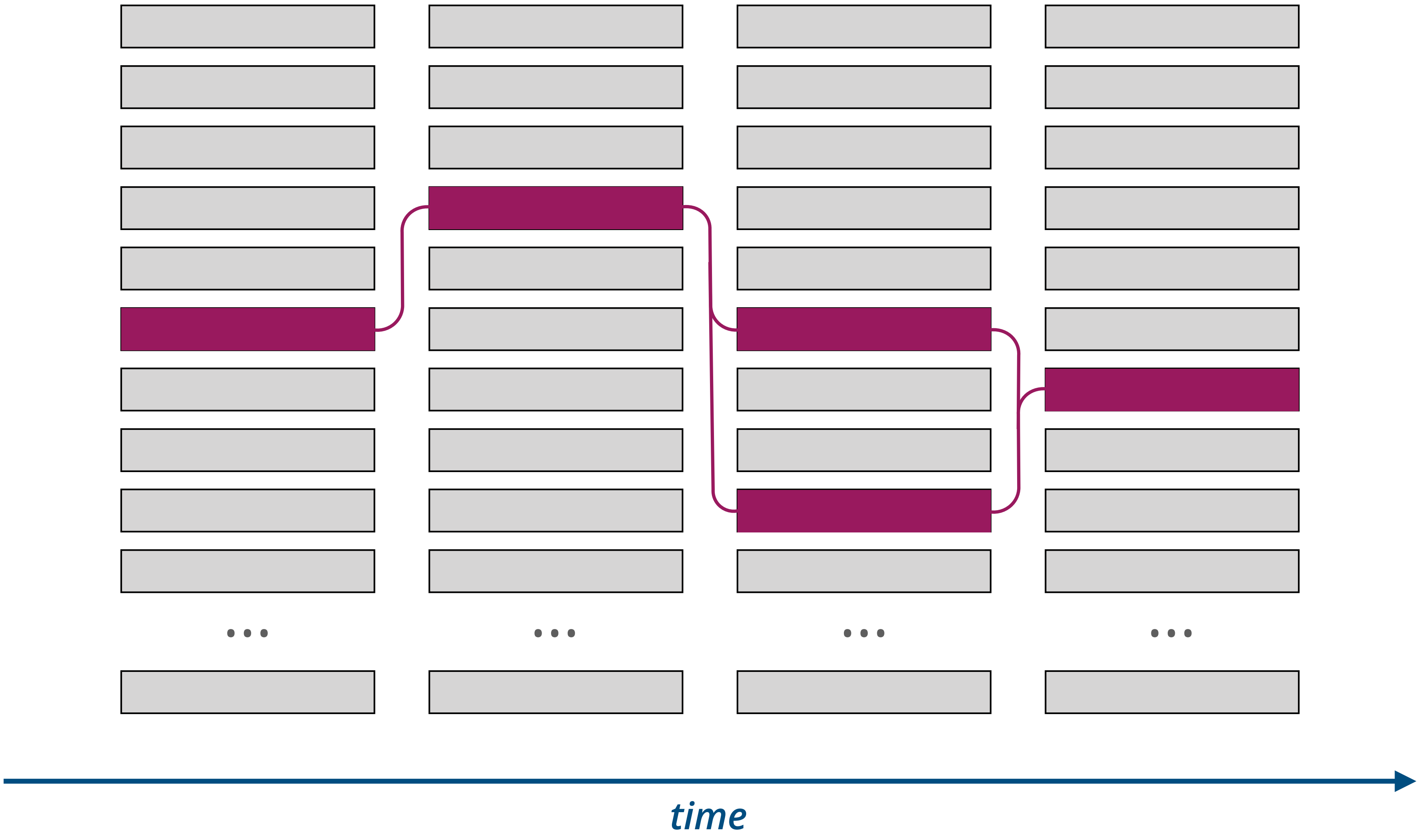
what's next

Trio's nurseries are cool!

- almost no out of order execution
- control flow is traceable
- exceptions are never lost
- `with` and `try` blocks work
- they solve the "goto problem" in concurrency

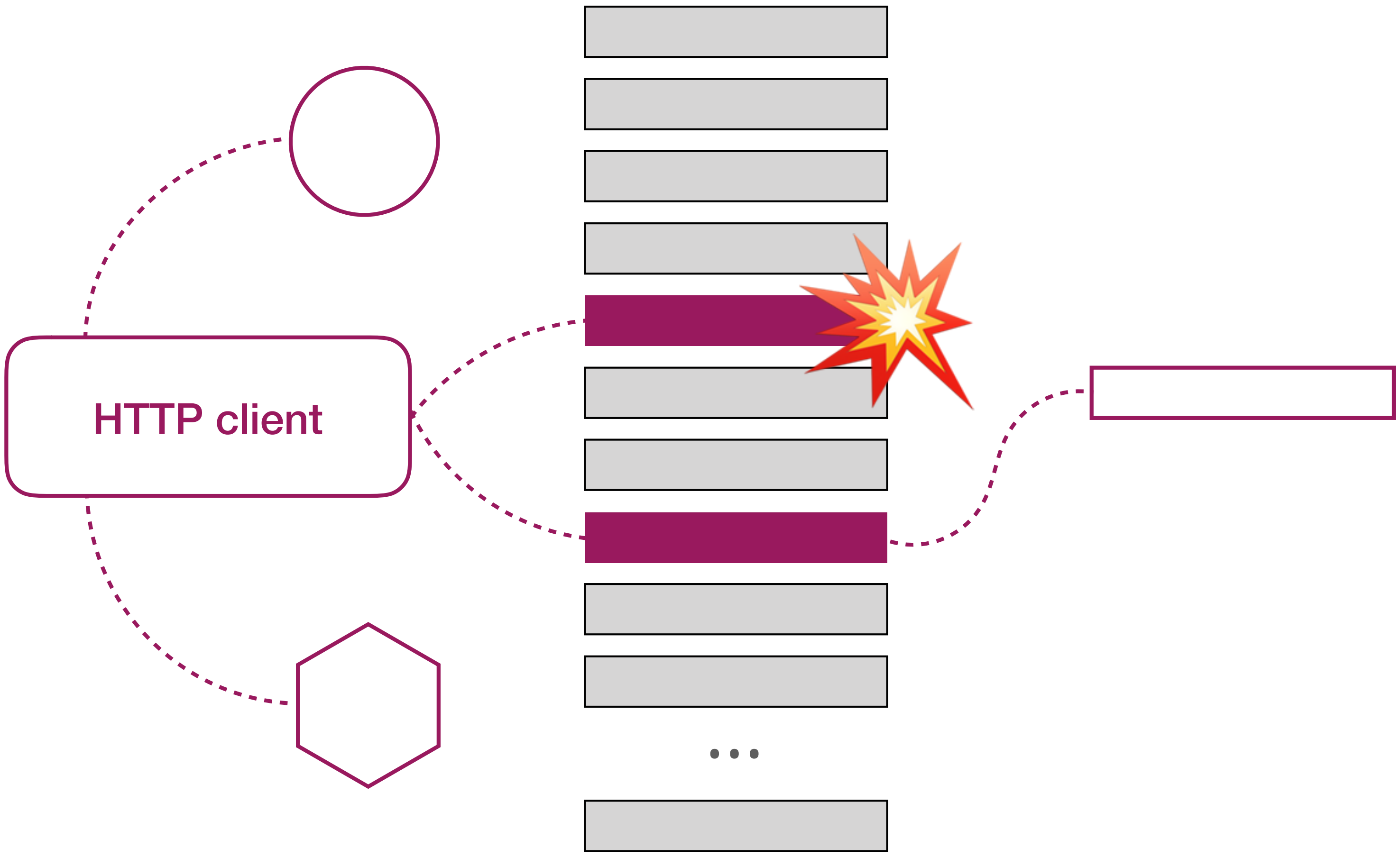
what's next

Back to asyncio



what's next

a typical asyncio library



what's next

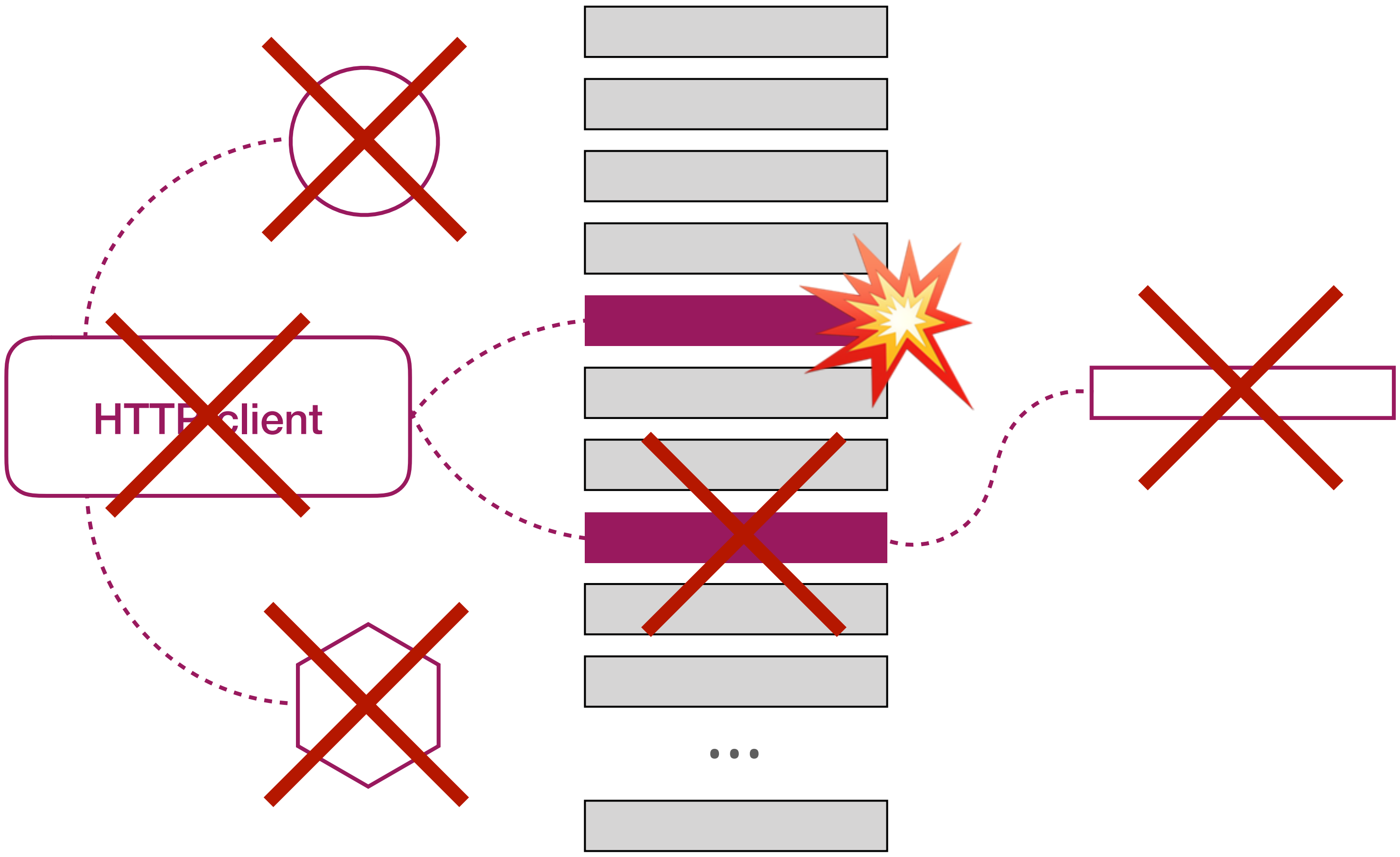
a typical asyncio library

```
def a_callback():
    try:
        # logic
    except Exception:
        # do something?

loop.call_soon(a_callback)
```

- most libraries do nothing 😿
- some invent ad-hoc half-working solutions
- this is a bug magnet in asyncio

what's next

what we need



what's next

💡 an idea for Python 3.8 💡

`loop.create_supervisor()`

- new **low-level** API for libraries and frameworks
- something I'm thinking about to land in Python 3.8

what's next

create_supervisor()

- returns an asynchronous context manager
- supervisor mirrors all asyncio event loop APIs
- can be passed from one coroutine to another

what's next

create_supervisor()

```
async def get(url):
    loop = asyncio.get_running_loop()

    async with loop.create_supervisor() as sup:
        sup.create_connection(
            http_proto_factory, ...)
        # or
        sup.call_soon(...)
        # or
        fut = sup.create_future(...)
        # or
        task = sup.create_task()
```

create_supervisor()

- any unhandled exception in a callback or transport (IO) cleans up all resources allocated through the supervisor
- easier to implement cancellation and cleanup logic on top
- third-party loops are in control

what's next

💡 another idea for Python 3.8 💡

`asyncio.TaskGroup()`

[thanks, Curio!]

```
async def main():
    async with asyncio.TaskGroup() as t:
        t.create_task(coro1)
        t.create_task(coro2)
```

what's next

TaskGroup()

- will use the new `loop.create_supervisor()` under the hood
- more convenient API than `asyncio.gather()`
- easy to schedule tasks in "buckets"

what's next

So what to expect from 3.8?

1. we hear you!
2. documentation improvements
3. likely: `loop.create_supervisor()`
4. likely: `asyncio.TaskGroup()`
5. maybe: event loop low-level tracing API

what's next

So what to expect from 3.8?

6. maybe: timeout and cancel scopes like in Trio
7. maybe: new streams API
8. maybe: add a context manager for `shield()`
9. likely: SSL over SSL
10. Make CancelledError a BaseException

Thank you!

Questions?