Asyncio

Martin Natano, Ivana Kellyerova

May 4, 2019

What is asyncio?

What is asyncio?

What is asyncio?

Welcome to the asyncio workshop. I will start the workshop with a short introduction to asyncio. Afterwards there will be exercises you can work on at your own speed.

This is a workshop and supposed to be interactive, so please interrupt me at any time when you have a question.

asyncio is a library included in python's standard libary that allows to work with coroutines. It uses the builtin async/await syntax.

Coroutines

```
import asyncio

async def alice():
    print('Hi!')
    print('My␣name␣is␣Alice')
    await asyncio.sleep(1)
    print('Bye')

async def bob():
    print('Nice␣to␣meet␣you')
```

Coroutines

Coroutines look like normal functions, but they annotated with the async keyword. Coroutines can use the await keyword, which allows to transfer control to another coroutine.

Multiple coroutines can run concurrently. They do not run at the same time, but execution interleaves – meaning the program switches between the coroutines with only one coroutine active at any given time.

The switches only happen when an await statement is encountered. This also means you can rely on the fact, that your coroutine will not be interrupted when there is no await in a sequence of statements.

With coroutines you can have code that switches between doing a lot of different things, while still featuring seemingly linear function. This can make your codebase easier to understand and debug.

In the example Alice introduces herself and Bob replies in a polite manner. However, asyncio doesn't guarantee that Alice will run first. As a matter of fact Bob could talk first. To prevent that there are synchronization primitives available, that I will explain later.

note 1 of slide 3

Concurrency vs. Parallelism

- **parallelism**: multiple tasks being processed at the same time, e.g. on a multicore machine
  → CPU intensive computations
  → threads, processes
- **concurrency**: tasks can overlap, but don't necessarily run at the same time
  → waiting for IO, e.g. networking
  → coroutines :)

4 / 25

Concurrency vs. Parallelism

parallelism: Think: trains. They all drive at the same time. This also means that they need synchronization – you don't want trains to collide.

concurrency: Concurrency is what I do when working on two tickets at once. Implement a bit of the first ticket, then switch to the other one to work on it, switch back to the first one, and so on.

What does waiting for IO mean? That is for example waiting for the response of an HTTP request, waiting for the database to return data or waiting for another process to produce output.

note 1 of slide 4

Libraries using asyncio

- ☐ clients and servers for HTTP, websockets
- ☐ DB connectors: InfluxDB, MySQL, Postgres, ...
- ☐ message queue connectors: Kafka, ZeroMQ, ...
- ☐ implementations of networking protocols: DNS, SSH, ...
- ☐ ...and so much more!

5 / 25

Running coroutines

6 / 25

asyncio.run()

```
asyncio.run(coro())
```

7 / 25

asyncio.run()

The easiest way to run a coroutine from a normal function and wait for it to finish.

The function call looks simple, but there is much more going on in the background than might be apparent here. Coroutines are run by an event loop that is responsible for switching between them. The run function creates such an event loop, schedules execution of the coroutine and waits for it to finish.

note 1 of slide 7

await

```
result = await coro()
```

8 / 25

await

Run a coroutine from inside another coroutine and wait for it to finish.

note 1 of slide 8

asyncio.wait_for()

```
try:
    result = await asyncio.wait_for(
        coro(),
        timeout=1,
    )
except asyncio.TimeoutError:
    print('Timeout!')
```

asyncio.wait_for()

Run a coroutine with a timeout. When the coroutine takes longer than the timeout, it will be cancelled and a TimeoutError is raised.

note 1 of slide 9

asyncio.gather()

```
results = await asyncio.gather(
    coro_1(),
    coro_2(),
)
```


asyncio.gather()

Run multiple coroutines and gather their return values. The result of gather is a future, that when awaited returns a list of results.

note 1 of slide 10

asyncio.wait()

```
await asyncio.wait([coro_1(), coro_2()])

done, pending = await asyncio.wait(
    [coro_1(), coro_2()],
    timeout=1,
)

done, pending = await asyncio.wait(
    [coro_1(), coro_2()],
    return_when=asyncio.FIRST_COMPLETED,
)
```


asyncio.wait()

Wait for one or more coroutines to finish. You can either wait for one or all of the coroutines to finish. Timeouts are also supported.

note 1 of slide 11

asyncio.create_task()

```
task = asyncio.create_task(coro())
...
result = await task
task.cancel()
```

12 / 25

asyncio.create_task()

Asyncio allows the creation tasks. With a task you can start a coroutine without immediately waiting for it. The result of that coroutine can be obtained later.

note 1 of slide 12

Synchronization 13 / 25

Synchronization

Sometimes you have to communicate between coroutines, or manage how and when they access resources. That's where the synchronization primitives help you.

This is gonna look familiar to people who already worked with threads.

note 1 of slide 13

Lock

```
sound_lock = asyncio.Lock()

async def play_sound():
    async with sound_lock:
        ...          # play sound
```

14 / 25

Lock

Locks allow to manage exclusive access to a resource. Only one coroutine can hold the lock at any given time. If another coroutine already holds the lock, acquiring the lock will block until it is available.

Locks act as asynchronoous context manager - that's the async with you can see here. An asynchronous context manager is similar to a normal context manager, except that they can use the await keyword inside and have to be used with async with.

note 1 of slide 14

Event

```
bob_is_done = asyncio.Event()

async def alice():
    await bob_is_done.wait()
    print('finally')

async def bob():
    await asyncio.sleep(60)          # chill
    bob_is_done.set()
```

15 / 25

Event

Just like a boolean variable, but you can wait for it to become true.

note 1 of slide 15

Semaphore

```
max_three = asyncio.Semaphore(3)

async def download_large_file():
    async with max_three:
        ...          # download large file
```

16 / 25

Semaphore

Semaphores allow to control how many coroutines can enter a section or multiple sections at once.

In this example we use it to limit the number of simultaneous downloads.

note 1 of slide 16

Condition

```
cond = asyncio.Condition()
```

17 / 25

Condition

There are also conditions object. I'm not going to explain them here. I never had to use them in actual code, but in some circumstances they can be useful. If you are interested in them please read it up in the documentation.

note 1 of slide 17

Queue

```
queue = asyncio.Queue()

async def compute_squares():
    for i in range(1000):
        await queue.put(i ** 2)

async def print_squares():
    while True:
        print(await queue.get())
```

18 / 25

Queue

Queues allow to move data between coroutines in a first in, first out manner. The *put* method pushes a value to the queue. Per default the size of a queue is unlimited, but if you specify a maximum size, the *put* method would block until there is space available. *get* fetches a value from the queue and block if it is empty, until there is a value available.

note 1 of slide 18

Hands-on

19 / 25

Which Python version

Python 3.7
(or newer)

20 / 25

virtualenv

```
$ pyenv install 3.7.3
$ pyenv local 3.7.3
$ python --version
Python 3.7.3
$ python -m venv env
$ . env/bin/activate
(env) $ pip install -U pip setuptools
(env) $ pip install jupyter
(env) $ jupyter notebook
```

Exercises: `https://www.natano.net/data/workshops/pydays2019/`

21 / 25

Quiz

Should you use asyncio for ...

a HTTP microservice? yes
calculating prime numbers? no
AI, machine learning? probably not
a server for an online multiplayer game? yes, absolutely!

23 / 25

Questions

Any questions?

24 / 25

Thanks

Thx! `https://www.natano.net/data/workshops/pydays2019/`

25 / 25