Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Python asyncio

Exceptions / Merging
aiohttp / aiofiles / …

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Objectives

- Exception handling in asyncio
- Splitting and Merging different task “flows”
- `aiofiles`
- `aiohttp`
- … other asyncio-capable libraries

References

[1] Python Concurrency With Asyncio, Matthew Fowler, 2022 Manning Publications, ISBN 978-1617298660

[2] aiofiles https://pypi.org/project/aiofile/

[3] aiohttp https://docs.aiohttp.org/en/stable/

Exceptions – something went wrong ☹

- Any coroutine (also tasks) can produce exceptions
- The exceptions must be catch and correctly treated

- Asynchronous exceptions are caught **across tasks** in the main coroutine
  - This makes catching exceptions centrally easy to implement
  - The readability of the code stays good

Exceptions – Simple coroutine

```
import asyncio


async def foo():
    raise ValueError("Foo value error")
    return("Foo finished") # not executed!


async def main():
    try:
        result = await foo()
        print(result)
    except ValueError as e:
        print(f"EXCEPTION: ValueError: {e}")


if __name__ == '__main__':
    asyncio.run(main())
```

When awaiting a **single coroutine** exception handling is managed in the **same way** as non-asyncio code using a

```
try
    …
except
    …
finally
```

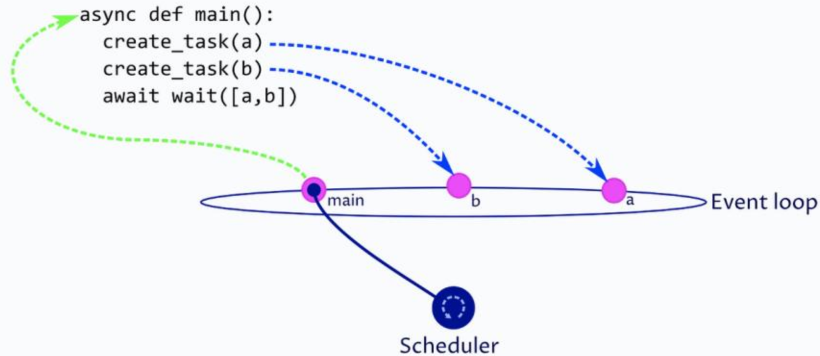
, block

Exceptions – Multiple coroutines

- If multiple coroutines are running, what should happen if an exception occurs?
  - Should it be immediately propagated to the main coroutine?
  - What about unfinished tasks?
  - Should they be terminated or allowed to continue?

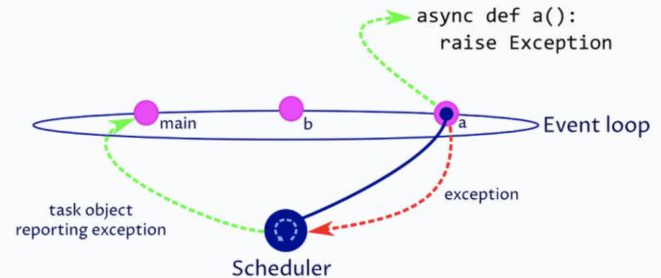
- If a coroutine (not the main coroutine) is currently running, the main coroutine is paused
  - So, where goes the exception?

Exceptions – Exception flow (1/2)



- The main coroutine is running in the EL
- The tasks are created and wait to get executed

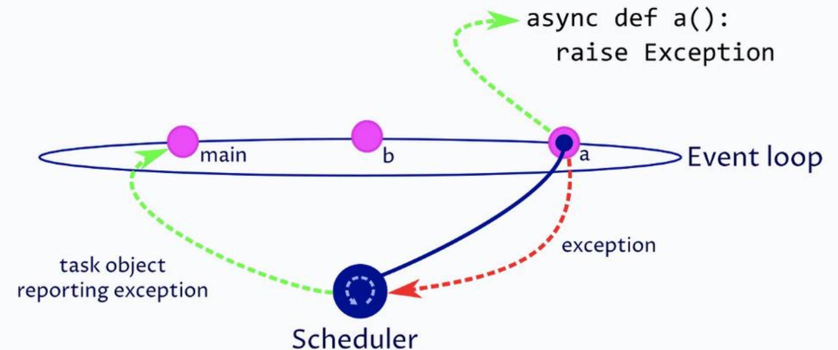
- Coroutine **a** is running in the EL
- An exception can't be propagated to main coroutine
- Propagation to scheduler (EL)



Source: https://python.plainenglish.io/how-to-manage-exceptions-when-waiting-on-multiple-asyncio-tasks-a5530ac10f02

Exceptions – Exception flow (2/2)

- The exception pauses the “faulty” coroutine
- The scheduler passes a *task object* with the exception information to the main coroutine
- Probably other coroutines are still running?
- Should the coroutines continue?
- Or should the main co-routine get the priority?



Source: https://python.plainenglish.io/how-to-manage-exceptions-when-waiting-on-multiple-asyncio-tasks-a5530ac10f02

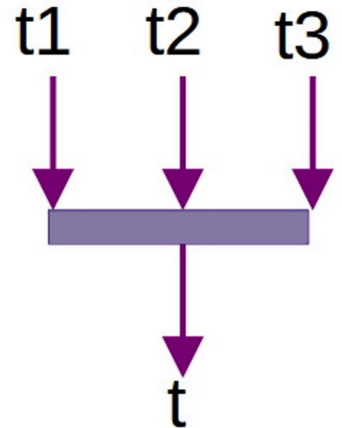
Exceptions – Waiting on different tasks, with a plan

- What behaviour is required when an exception is raised in a task (coroutine)?
- Do you want to wait for all other tasks to complete before the calling coroutine continues execution?
  - How will you determine which tasks returned results and which had exceptions?
  - How many tasks can simultaneously return exceptions?
- Or return immediately?
  - How do you ensure that all other tasks have completed execution and any resources they use are safely released?

Waiting the end of multiple tasks

- Multiple launched tasks can be “awaited” for the end (or an exception) with the asyncio.wait() method

```
await asyncio.wait({tasks},
                   return_when=…)
```



- `return_when=`
  - FIRST_COMPLETED
  - ALL_COMPLETED
  - FIRST_EXCEPTION

Exception – not working example!

```
async def main():
    foo_task = asyncio.create_task(foo(), name="Exception task")
    bar_task = asyncio.create_task(bar(), name="Waiting task")
    try:
        done, pending = await asyncio.wait(
            [foo_task, bar_task],
            return_when=asyncio.ALL_COMPLETED
        )
        for task in done:
            name = task.get_name()
            print(f"DONE: {name}")
        for task in pending:
            print(f"Pending: {name} --> cancel")
            task.cancel()
    except Exception as e:
        print(f"Exception caught: {e}")
```

```
return_when=
– FIRST_COMPLETED
– ALL_COMPLETED
– FIRST_EXCEPTION
```

Exception – Return information from tasks

- `get_name()`: give the name of the task (optional)
- `exception()`: returns exception object (None if no exception)
- `result()`: returns the result of the coroutine, **re-throws** the exception if there were one

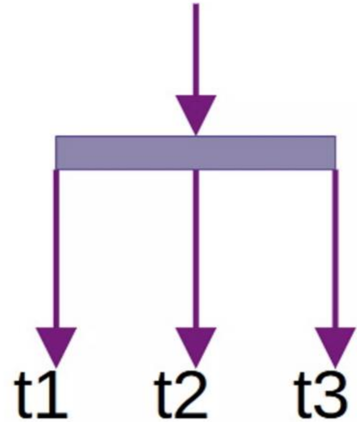
- **`asyncio.wait()`** does **not propagate** exceptions automatically. Instead, exceptions are stored inside Tasks, and the programmer decides how to inspect and handle them.

Exception –working example :-)

```
async def main():
    foo_task = asyncio.create_task(foo(), name="Exception_task")
    bar_task = asyncio.create_task(bar(), name="Waiting_task")
    try:
        done, pending = await asyncio.wait([foo_task, bar_task],
            return_when=asyncio.ALL_COMPLETED
        )
        for task in done:
            name = task.get_name()
            print(f"DONE: {name}")
            exception = task.exception()
            if isinstance(exception, Exception):
                print(f"{name} threw {exception}")
            try:
                result = task.result()
                print(f"{name} returned {result}")
            except ValueError as e: print(f"ValueError: {e}")
        for task in pending:
            name = task.get_name()
            print(f"Pending: {name} --> cancel")
            task.cancel()
    except Exception as e: print("Outer Exception")
```

Multiple usage of the same task

```
import asyncio

async def trigger(t, id):
    s = await t
    print(s, id)


async def main():
    t = asyncio.create_task(asyncio.sleep(1, 'Hello'))
    for i in range(3):
        asyncio.create_task(trigger(t, i))
    await t


if __name__ == '__main__':
    asyncio.run(main())
```



Gathering results from different coroutines

- Like the asyncio.wait(), there exists the asyncio.gather() call for gathering the results
- **gather()** defines:
  - how exceptions propagate
  - what happens to remaining tasks

```
list = await asyncio.gather(
               factorial("A", 5),
               factorial("B", 5),
               factorial("C", 5),)
# list contains a list of all return values from each task
```

Summary

API	Exception behavior
await coro	immediate propagation
asyncio.wait()	manual inspection
asyncio.gather()	automatic propagation (default)

aiofiles – Asynchronous file handling

aiofiles: file support for asyncio

pypi v0.8.0 | build passing | codecov 90% | python 3.6 | 3.7 | 3.8 | 3.9 | 3.10

aiofiles is an Apache2 licensed library, written in Python, for handling local disk files in asyncio applications.

Ordinary local file IO is blocking, and cannot easily and portably made asynchronous. This means doing file IO may interfere with asyncio applications, which shouldn't block the executing thread. aiofiles helps with this by introducing asynchronous versions of files that support delegating operations to a separate thread pool.

```
async with aiofiles.open('filename', mode='r') as f:
    contents = await f.read()
print(contents)
'My file contents'
```

Asynchronous iteration is also supported.

```
async with aiofiles.open('filename') as f:
    async for line in f:
        ...
```

Features

- a file API very similar to Python's standard, blocking API
- support for buffered and unbuffered binary files, and buffered text files
- support for `async`/`await` (PEP 492) constructs
- async interface to tempfile module

https://github.com/Tinche/aiofiles/

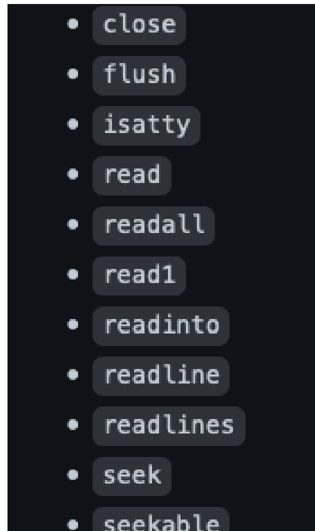
aiofiles - Example

- Install the `aiofiles` package (e.g. with pip)

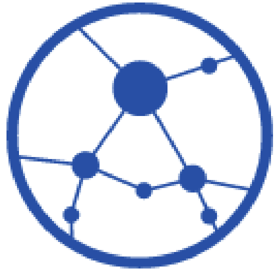
```
async with aiofiles.open('filename', mode='r') as f:
    contents = await f.read()
print(contents)

# or through iteration (line by line)

async with aiofiles.open('filename') as f:
    async for line in f:
        ...
```

aiofiles supported methods

- The API of `aiofiles` is very similar to the standard file operations
  - Except that they are executed as coroutines



aiohttp - Asynchronous



Async HTTP client/server for asyncio and Python

Welcome to AIOHTTP

Asynchronous HTTP Client/Server for asyncio and Python.

Current version is 3.8.1.

Key Features

- Supports both Client and HTTP Server.
- Supports both Server WebSockets and Client WebSockets out-of-the-box without the Callback Hell.
- Web-server has Middlewares, Signals and plugable routing.

https://docs.aiohttp.org

aiohttp doesn't like `requests` library

- Don't mix **`aiohttp`** with **`requests`** library calls
- `requests` library uses **blocking sockets**, you would destroy the benefits of asyncio
  - The entire event loop would be blocked ☹

aiohttp – Client example

```
import asyncio
import aiohttp


async def main():
    async with aiohttp.ClientSession() as session:
        async with session.get('https://api.github.com/events') as resp:
            print(resp.status)
            print(await resp.text())

if __name__ == '__main__':
    asyncio.run(main())


# syntactical sugar: If you can, work with the (async) context managers,
# all clean-up stuff in case of an exception, etc. will be done for you
```

aiohttp – The session

- The session is always the first thing that must be opened
  - The session is like launching the Browser
- **Use only one session**, even if you are downloading multiple content
  - You can, in certain situations, open a specific session per main URL
- With a session you'll keep many connections open
  - Connection pooling → performance
  - Cookies are saved internally in the session

Processing requests as they complete

- `asyncio.gather()` waits until **all** awaitables (tasks) **have finished** before allowing access to any result!
  - E.g., the result from a slow web server can block the post-processing of fast web server (where the results are already here)

- The API function `asyncio.as_completed()` returns an iterator of finished awaitables

Processing requests as they complete - Example

```
async def fetch_status(session: ClientSession, url: str, delay: int = 0) -> int:
    await asyncio.sleep(delay)
    async with session.get(url) as result:
        return result.status


async def main():
    async with aiohttp.ClientSession() as session:
        fetchers = [fetch_status(session, 'https:/ / www.example .com', 1),
                    fetch_status(session, 'https:/ / www.example .com', 1),
                    fetch_status(session, 'https:/ / www.example .com', 10)]
        for finished_task in asyncio.as_completed(fetchers):
            print(await finished_task)

asyncio.run(main())
```

Other asynchronous libraries

- Have a look for helping libraries in the asyncio world here:
- https://github.com/timofurrer/awesome-asyncio

A carefully curated list of awesome Python asyncio frameworks, libraries, software and resources.