

# Asyncio: offrez des tulipes à vos entrées-sorties asynchrone

Thierry Chappuis



Ecole d'ingénieurs et d'architectes de Fribourg  
Hochschule für Technik und Architektur Freiburg



# Qui?

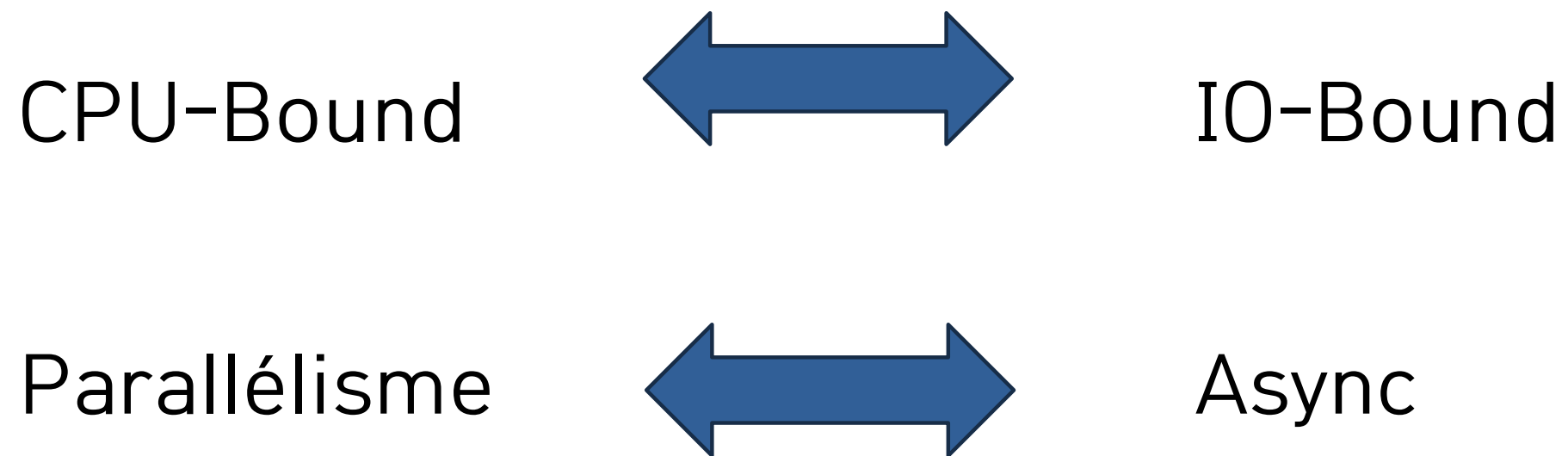
- Master en génie chimique et PhD en biotech à EPFL (Lausanne)
- Professeur de génie chimique à la HES-SO (Fribourg)
- Code en Python depuis 2000.
- Stack python scientifique
- Touche à tout, utilise python comme couteau suisse digital.

# Quoi? (1/2)

Dans la vraie vie, résoudre des problème à l'aide de e.g. Python implique de communiquer avec des sources extérieures:

- base de donnée
- socket
- autre worker
- service fournissant des données à travailler

# Quoi? (2/2)

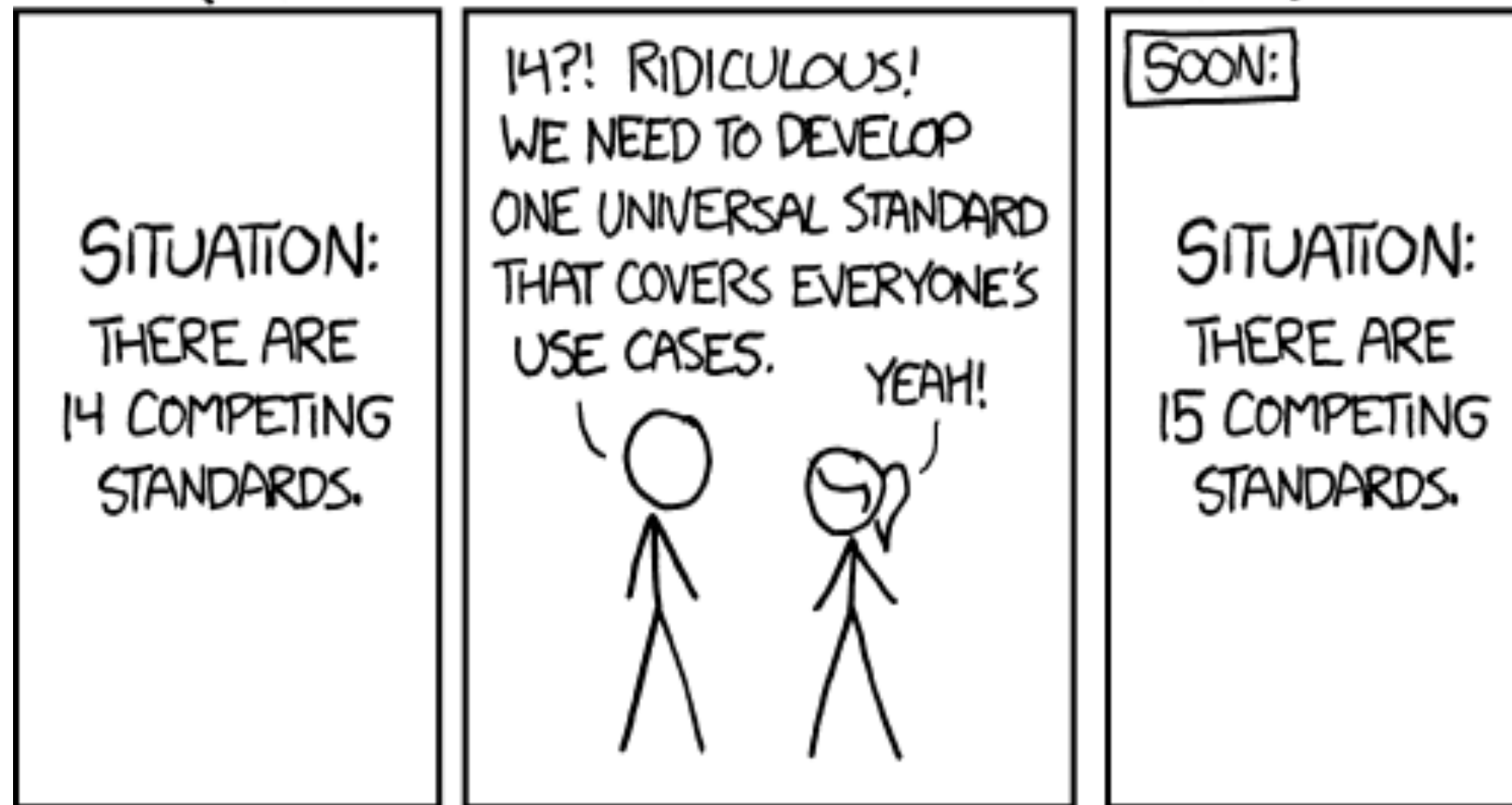


# Comment?

- Twisted
- gevent
- Tornado
- ...
- PEP-3148 → `concurrent.futures` (Python 3.2)
- PEP-380 → **yield from** (Python 3.3)
- PEP-3156 → Tulip → **asyncio** (Python 3.4)

# Vers un nouveau standard ? (1/2)

HOW STANDARDS PROLIFERATE:  
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)



<http://imgs.xkcd.com/comics/standards.png>

# Vers un nouveau standard? (2/2)

*"I'm not trying to reinvent the wheel. I'm trying to build a good one."*

*Guido van Rossum*

# Pourquoi? (1/3)

- ayncore et asynchat: included batteries don't fit
  - peu extensibles
  - personne ne les utilisent
- Nouvelle syntaxe yield from pour les générateurs
- Nouvelle implementation des IO asynchrones
  - Python  $\geq$  3.3
  - Trollius: backport pour Python 2.6



## Pourquoi ? (2/3)

- Conçu pour l'interopérabilité avec d'autres frameworks asynchrones
- Twisted est bien pour des protocoles ésotériques tandis que Tornado est excellent pour du http.

# Pourquoi? (3/3)

- Personne n'aime les fonctions de rappel
- Asyncio : Callbacks without callbacks

# Les composants de asyncio?

- Une **boucle événementielle** substituable avec des implémentations spécifiques à l'OS.
- Abstractions pour les **protocoles** et les canaux de communication (**transports**), à la Twisted.
- TCP, UDP, SSL, pipes, appels différés
- Co-routines et tâches basées sur yield from (PEP-380)
- Futures
- Primitives de synchronisation imitant le module threading

# Coroutines, Futures et Tâches?

- **Coroutine**
  - Fonction génératrice
  - Décorée avec @coroutine
- **Future**
  - Avance sur salaire (promesse d'un résultat ou d'une erreur)
- **Task**
  - Future qui exécute une coroutine

# L'objet Future en bref

- Une promesse: `f = Future()`
  - `f.set_result()`
  - `f.result()`
  
  - `f.set_exception(e)`
  - `e = f.exception()`
  
  - `f.add_done_callback()`
  - `f.remove_done_callback()`

# Future() et les coroutines

- `yield from` peut renvoyer un Future
  - `f = Future()`
    - Quelqu'un va affecter un résultat ou une exception à `f`
  - `r = yield from f`
    - Attend que la tâche soit réalisée et retourne `f.result()`

# Et les tâches?

- Une tâche est une coroutine enveloppée dans un Future.

# Quelle différence entre tâche et coroutine?

- La coroutine ne s'exécute pas sans un mécanisme d'ordonnancement.
- Une tâche avance "toute seule"
  - La boucle événementielle joue le rôle de scheduler
  - pour la magie: voir Victor Stinner



# Exemples jouets de la doc asyncio

## 1. Définition de la coroutine

```
import asyncio

@asyncio.coroutine
def factorial(name, number):
    f = 1
    for i in range(2, number+1):
        yield from asyncio.sleep(1)
    f *= i
```

# Exemples jouets de la doc asyncio

## 2. Création des tâches et lancement de la boucle événementielle

```
loop = asyncio.get_event_loop()
tasks = [
    asyncio.async(factorial("A", 2)),
    asyncio.async(factorial("B", 3)),
    asyncio.async(factorial("C", 4))]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

# Exemples jouets de la doc asyncio

## 1. Définition de la coroutine

```
import asyncio

@asyncio.coroutine
def tcp_echo_client(message, loop):
    r, w = yield from asyncio.open_connection(
                                                '127.0.0.1', 8888)

    w.write(message.encode())
    data = yield from r.read(100)
    w.close()
```

# Exemples jouets de la doc asyncio

## 2. Création de la tâche et lancement de la boucle événementielle

```
loop = asyncio.get_event_loop()
loop.run_until_complete(
    tcp_echo_client('Hello', loop))
loop.close()
```

# Exemples jouets de la doc asyncio

## 1. Définition de la coroutine

```
import asyncio

@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read()
    writer.write(data)
    yield from writer.drain()
    writer.close()
```

# Exemples jouets de la doc asyncio

## 2. Event loop

```
loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo,
                             '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

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

# Protocoles et transports à la Twisted

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, msg, loop):
        self.msg = msg
        self.loop = loop
        ...
```

# Protocoles et transports à la Twisted

```
class EchoClientProtocol(asyncio.Protocol):  
    ...  
    def connection_made(self, transport):  
        transport.write(self.msg.encode())  
        print('Data sent')  
    ...
```



# Protocoles et transports à la Twisted

```
class EchoClientProtocol(asyncio.Protocol):  
    ...  
    def data_received(self, data):  
        print('Data received')  
    ...
```

# Protocoles et transports à la Twisted

```
class EchoClientProtocol(asyncio.Protocol):  
    ...  
    def connection_lost(self, exc):  
        print('The server left')  
        print('Stop the event loop')  
        self.loop.stop()  
    ...
```

# Protocoles et transports à la Twisted

```
...
loop = asyncio.get_event_loop()
msg = 'Hello'
coro = loop.create_connection(
    lambda: EchoClientProtocol(msg, loop),
    '127.0.0.1', 8888)
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

# Un exemple avec asyncio-redis

```
import asyncio
import asyncio_redis as aior

@asyncio.coroutine
def my_subscriber(channels):
    conn = yield from aior.Connection.create(
        host='localhost', port= 6379)
    subscriber = yield from connection.start_subscribe()
    yield from subscriber.subscribe(channels)
    while True:
        reply = yield from subscriber.next_published()
        # do something with reply
```

# Un exemple avec asyncio-redis

...

```
loop = asyncio.get_event_loop()
asyncio.async(my_subscriber('channel-1'))
asyncio.async(my_subscriber('channel-2'))
loop.run_forever()
```

# Un exemple avec asyncio-redis

On peut écrire un PUB-SUB très similaire pour `aiozmq` pour ZeroMQ, qui utilise habituellement Tornado ou `gevent` pour l'asynchrone.

# Projets et ressources asyncio

- <https://code.google.com/p/tulip/wiki/ThirdParty>
- <http://asyncio.org>

# Conclusions

Asyncio/Tulip est un projet jeune mais...

- Présente un potentiel d'interopérabilité intéressant
- Modèle élégant
- La courbe d'apprentissage semble relativement aisée
- Code asynchrone linéaire (facile à lire)
  
- à suivre: la réponse à moyen terme des acteurs du domaines



**Questions?**