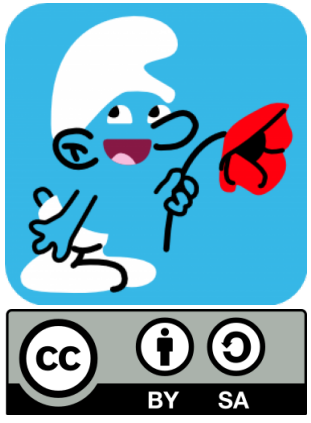
Plongée au cœur du modèle asynchrone Python

Sans boire la tasse !

Antoine “entwanne” Rozo



Plongée au cœur du modèle asynchrone Python

- `asyncio` n'est pas le seul moteur asynchrone
- `async` et `await` ne lui sont pas intrinsèquement liés
- Comment réécrire `asyncio` ?
- https://github.com/entwanne/presentation_python_plongee_asynchrone

Un monde de coroutines

Coroutines

- Définition d'une coroutine depuis Python 3.5 :

```
async def simple_print(msg):
    print(msg)
```

- `simple_print` est une fonction renvoyant une coroutine

```
simple_print
```

```
simple_print('Hello')
```

Coroutines

- Le contenu est exécuté par le moteur asynchrone, ici à l'aide d'`await`

```
await simple_print('Hello')
```

- En dehors d'un *REPL* asynchrone, il faudrait utiliser `asyncio.run`

```
asyncio.run(simple_print('Hello'))
```

- Ou encore interagir directement avec la boucle événementielle :

```
loop = asyncio.new_event_loop()
loop.run_until_complete(simple_print('Hello'))
```

- Cette boucle exécute et cadence les différentes tâches
- Elle permet une utilisation concurrente

Coroutines - introspection

- De quoi est faite une coroutine ?
- C'est un objet avec une méthode `__await__`

```
coro = simple_print('Hello')
dir(coro)
```

Coroutines - introspection

- Cette méthode renvoie un itérateur (`coroutine_wrapper`)

```
aw = coro.__await__()
aw

dir(aw)
```

Coroutines - itération

- On peut donc itérer manuellement sur une coroutine

```
for _ in simple_print('Hello').__await__():
    pass
```

Coroutines - itération

- De même avec une coroutine plus complexe

```
async def complex_work():
    await simple_print('Hello')
    await asyncio.sleep(0)
    await simple_print('World')


for _ in complex_work().__await__():
    pass
```

Coroutines - itération

- Plusieurs itérations sont bien parcourues

```
it = complex_work().__await__()
next(it)

next(it)
```

- La boucle reprend le contrôle à chaque interruption
- Le `await asyncio.sleep(0)` a pour effet de `yield`
- `await` est équivalent à `yield from`

Attendez-moi !

Awaitables

- Les coroutines sont des tâches asynchrones (*awaitables*)
- Un awaitable est un objet avec une méthode `__await__`
- Tâche équivalente à la coroutine `complex_work` :

```
class ComplexWork:
    def __await__(self):
        print('Hello')
        yield
        print('World')
```

- Le `yield` rend la méthode génératrice, qui renvoie donc un itérateur

```
await ComplexWork()
```

Awaitables - itération

- Notre tâche respecte le protocole établi

```
it = ComplexWork().__await__()
next(it)

next(it)
```

Awaitables

- Les tâches autres que les coroutines sont peu courantes
- Mais sont utiles pour conserver un état associé à la tâche

```
class Waiter:
    def __init__(self):
        self.done = False

    def __await__(self):
        while not self.done:
            yield
```

Awaitables - synchronisation

- `Waiter` permet à deux tâches de se synchroniser

```
waiter = Waiter()

async def wait_job(waiter):
    print('start')
    await waiter # wait for count_up_to to be finished
    print('finished')

async def count_up_to(waiter, n):
    for i in range(n):
        print(i)
        await asyncio.sleep(0)
    waiter.done = True

await asyncio.gather(wait_job(waiter), count_up_to(waiter,
```

Boucle d'or et les trois tâches

Boucles événementielles

- Premier prototype de boucle événementielle

```
def run_task(task):
    it = task.__await__()

    while True:
        try:
            next(it)
        except StopIteration:
            break
```

Boucles évenementielles

- Peu d'utilité pour n'exécuter qu'une seule tâche
- Version améliorée pouvant cadencer plusieurs tâches
- Utilisation d'une file pour connaître la prochaîne tâche à itérer

```
def run_tasks(*tasks):
    tasks = [task.__await__() for task in tasks]

    while tasks:
        # On prend la première tâche disponible
        task = tasks.pop(0)
        try:
            next(task)
        except StopIteration:
            # La tâche est terminée
            pass
        else:
            # La tâche continue, on la remet en queue de l
            tasks.append(task)
```

Boucles évenementielles - exécution

- Quelques exemples d'exécution concurrente

```
run_tasks(simple_print(1), ComplexWork(), simple_print(2),

waiter = Waiter()
run_tasks(wait_job(waiter), count_up_to(waiter, 5))
```

Environnement asynchrone

- Tâche unitaire simple pour rendre la main à la boucle

```
class interrupt:
    def __await__(self):
        yield
```

- Qui nous permet de développer d'autres utilitaires

```
import time

async def sleep_until(t):
    while time.time() < t:
        await interrupt()

async def sleep(duration):
    await sleep_until(time.time() + duration)
```

Environnement asynchrone - exemple

- Et d'en profiter dans notre environnement

```
async def print_messages(*messages, sleep_time=1):
    for msg in messages:
        print(msg)
        await sleep(sleep_time)

run_tasks(print_messages('foo', 'bar', 'baz'),
    print_messages('aaa', 'bbb', 'ccc', sleep_time=0.7))
```

Boucles événementielles - interactions

- Une boucle événementielle est plus utile si nous pouvons interagir avec elle une fois lancée
- Prototype d'une nouvelle boucle pouvant programmer des tâches à la volée (`add_task`)

```
class Loop:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        if hasattr(task, '__await__'):
            task = task.__await__()
        self.tasks.append(task)

    def run(self):
        while self.tasks:
            task = self.tasks.pop(0)
            try:
                next(task)
```

Boucles événementielles - interactions

- Ajout d'une méthode pour faciliter le lancement

```
class Loop:
    [...]

    def run_task(self, task):
        self.add_task(task)
        self.run()

loop = Loop()
loop.run_task(print_messages('foo', 'bar', 'baz'))
```

Boucles événementielles - interactions

- Ajout de `Loop.current` pour rendre la boucle accessible depuis nos tâches

```
class Loop:
    [...]

    current = None

    def run(self):
        Loop.current = self
        while self.tasks:
            task = self.tasks.pop(0)
            try:
                next(task)
            except StopIteration:
                pass
            else:
                self.add_task(task)
```

Boucles événementielles - utilitaires

- Implémentation de `gather`, utilitaire permettant d'attendre simultanément plusieurs tâches
- Amélioration de notre classe `Waiter` pour attendre plusieurs validations

```
class Waiter:
    def __init__(self, n=1):
        self.i = n

    def set(self):
        self.i -= 1

    def __await__(self):
        while self.i > 0:
            yield
```

Boucles événementielles - utilitaires

- Utilisée par gather pour attendre N tâches

```
async def gather(*tasks):
    waiter = Waiter(len(tasks))

    async def task_wrapper(task):
        await task
        waiter.set()

    for t in tasks:
        Loop.current.add_task(task_wrapper(t))
    await waiter

loop = Loop()
loop.run_task(gather(print_messages('foo', 'bar', 'baz'),
    print_messages('aaa', 'bbb', 'ccc', sleep_time=0.7)))
```

Boucles événementielles - utilitaires réseau

- Autre utilitaire : gestion de *sockets* asynchrones
- Utilisation de `select` pour savoir quand la *socket* est disponible
- Renvoi à la boucle événementielle le cas échéant

Boucles événementielles - utilitaires réseau

```
import select

class AIOSocket:
    def __init__(self, socket):
        self.socket = socket
        self.pollin = select.epoll()
        self.pollin.register(self, select.EPOLLIN)
        self.pollout = select.epoll()
        self.pollout.register(self, select.EPOLLOUT)

    def close(self):
        self.socket.close()

    def fileno(self):
        return self.socket.fileno()

    def __enter__(self):
        return self
```

Boucles événementielles - utilitaires réseau

```
class AIOSocket:
    [...]

    async def bind(self, addr):
        while not self.pollin.poll():
            await interrupt()
        self.socket.bind(addr)

    async def listen(self):
        while not self.pollin.poll():
            await interrupt()
        self.socket.listen()

    async def connect(self, addr):
        while not self.pollin.poll():
            await interrupt()
        self.socket.connect(addr)
```

Boucles événementielles - utilitaires réseau

```
class AIOSocket:
    [...]

    async def accept(self):
        while not self.pollin.poll(0):
            await interrupt()
        client, _ = self.socket.accept()
        return self.__class__(client)

    async def recv(self, bufsize):
        while not self.pollin.poll(0):
            await interrupt()
        return self.socket.recv(bufsize)

    async def send(self, bytes):
        while not self.pollout.poll(0):
            await interrupt()
        return self.socket.send(bytes)
```

Boucles événementielles - utilitaires réseau

```
import socket

def aiosocket(family=socket.AF_INET, type=socket.SOCK_STREA
    return AIOSocket(socket.socket(family, type, proto, fi
```

Boucles événementielles - utilitaires réseau

```
async def server_coro():
    with aiosocket() as server:
        await server.bind(('localhost', 8080))
        await server.listen()
        with await server.accept() as client:
            msg = await client.recv(1024)
            print('Received from client', msg)
            await client.send(msg[::-1])

async def client_coro():
    with aiosocket() as client:
        await client.connect(('localhost', 8080))
        await client.send(b'Hello World!')
        msg = await client.recv(1024)
        print('Received from server', msg)

loop = Loop()
loop.run_task(gather(server_coro(), client_coro()))
```

No Future

Futures

- L'implémentation précédente de `sleep` est inefficace
- La tâche est sans cesse reprogrammée pour rien
- De même pour la tâche `Waiter` qui n'a besoin d'être lancée qu'une fois sa condition validée

Futures - asyncio

- `asyncio` utilise un mécanisme de *futures* :

```
async def test():
    await asyncio.sleep(1)

loop = Loop()
loop.run_task(test())
```

- Le `yield` utilisé pour rendre la main à la boucle peut être accompagné d'une valeur

Futures - exemple

- Cette *future* ne doit être relancée qu'une fois sa condition validée

```
class Future:
    def __await__(self):
        yield self
        assert self.done
```

Futures - exemple

- On ajoute une méthode de validation qui reprogramme la tâche

```
class Future:
    def __init__(self):
        self._done = False
        self.task = None

    def __await__(self):
        yield self
        assert self._done

    def set(self):
        self._done = True
        if self.task is not None:
            Loop.current.add_task(self.task)
```

Futures - boucle événementielle

- Détection des *futures* par la boucle événementielle

```
class Loop:
    [...]

    def run(self):
        Loop.current = self
        while self.tasks:
            task = self.tasks.pop(0)
            try:
                result = next(task)
            except StopIteration:
                continue

            if isinstance(result, Future):
                result.task = task
            else:
                self.tasks.append(task)
```

Futures - événements temporels

- L'idée est d'associer une *future* à un temps
- On intègre pour cela une gestion d'événéments temporels

```
from functools import total_ordering

@total_ordering
class TimeEvent:
    def __init__(self, t, future):
        self.t = t
        self.future = future

    def __eq__(self, rhs):
        return self.t == rhs.t

    def __lt__(self, rhs):
        return self.t < rhs
```

Futures - événements temporels

- Ajout d'une méthode `call_later`

```
import heapq

class Loop:
    [...]

    handlers = []

    def call_later(self, t, future):
        heapq.heappush(self.handlers, TimeEvent(t, future))
```

Futures - événements temporels

- Prise en compte des événements temporels par la boucle

```
class Loop:
    [...]

    def run(self):
        Loop.current = self
        while self.tasks or self.handlers:
            if self.handlers and self.handlers[0].t <= time
                handler = heapq.heappop(self.handlers)
                handler.future.set()

            if not self.tasks:
                continue
            task = self.tasks.pop(0)
            try:
                result = next(task)
            except StopIteration:
                continue
```

Futures - utilitaires

- Ce qui nous permet une meilleure version de `sleep`

```
import time

async def sleep(t):
    future = Future()
    Loop.current.call_later(time.time() + t, future)
    await future

async def foo():
    print('before')
    await sleep(5)
    print('after')

loop = Loop()
loop.run_task(foo())
```

Et pour quelques outils de plus

Autres outils

- Nouveaux outils pour profiter de l'environnement asynchrone
- Nouveaux blocs : `for` et `with` asynchrones (`async for`, `async with`)

Itérables et générateurs asynchrones

- Un itérable asynchrone possède une méthode `__aiter__` renvoyant un itérateur asynchrone
- L'itérateur asynchrone a une méthode-coroutine `__anext__` renvoyant le prochain élément
- La méthode lève une exception `StopAsyncIteration` en fin d'itération

Itérables asynchrones

- Exemple : équivalent asychrone à `range`

```
class ARange:
    def __init__(self, stop):
        self.stop = stop

    def __aiter__(self):
        return ARangeIterator(self)


class ARangeIterator:
    def __init__(self, arange):
        self.arange = arange
        self.i = 0

    async def __anext__(self):
        if self.i >= self.arange.stop:
            raise StopAsyncIteration
        await sleep(1)
```

Itérables asynchrones

- Exécution au sein de notre moteur asynchrone

```
async def test_for():
    async for val in ARange(5):
        print(val)

loop = Loop()
loop.run_task(test_for())
```

Générateurs asynchrones

- On peut de façon similaire définir un générateur asynchrone (Python 3.6)

```
async def arange(stop):
    for i in range(stop):
        await sleep(1)
        yield i
```

Gestionnaires de contexte asynchrones

- Contexte asynchrone défini par ses méthodes `__aenter__` et `__aexit__`

```
class Server:
    def __init__(self, addr):
        self.socket = aiosocket()
        self.addr = addr

    async def __aenter__(self):
        await self.socket.bind(self.addr)
        await self.socket.listen()
        return self.socket

    async def __aexit__(self, *args):
        self.socket.close()
```

Gestionnaires de contexte asynchrones

- Exécution au sein de notre moteur asynchrone

```
async def test_with():
    async with Server(('localhost', 8080)) as server:
        with await server.accept() as client:
            msg = await client.recv(1024)
            print('Received from client', msg)
            await client.send(msg[::-1])

loop = Loop()
loop.run_task(gather(test_with(), client_coro()))
```

Gestionnaires de contexte asynchrones

- Contextes asynchrones intégrés à la `contextlib` (Python 3.7)

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def server(addr):
    socket = aiosocket()
    try:
        await socket.bind(addr)
        await socket.listen()
        yield socket
    finally:
        socket.close()
```

Conclusion

Conclusion

- Il n'est pas question de remplacer `asyncio` par ces exemples
- Le but est d'étudier comment cela fonctionne
- Retrouvez la présentation à l'adresse suivante :
  - https://entwanne.github.io/presentation_python_plongee_asynchrone/

Questions ?