Context Manager Asynchrone en Python

En Python, un **context manager asynchrone** permet de gérer des ressources asynchrones proprement avec `async with`, notamment pour : - La gestion de connexions réseau (`asyncio`, `aiohttp`, `aiomysql`, etc.). - L'accès à des bases de données asynchrones. - La gestion de tâches asynchrones.

1. Création d'un context manager asynchrone avec `__aenter__` et `__aexit__`

Python propose des méthodes spéciales `__aenter__` et `__aexit__` pour les context managers asynchrones.

```
import asyncio

class AsyncContext:
    async def __aenter__(self):
        """Exécuté au début du bloc `async with`"""
        print("Entrée dans le contexte asynchrone")
        await asyncio.sleep(1)  # Simulation d'une tâche asynchrone
        return self  # Permet d'accéder à l'instance dans le `async with`

    async def __aexit__(self, exc_type, exc_value, traceback):
        """Exécuté à la sortie du bloc `async with`"""
        print("Sortie du contexte asynchrone")
        await asyncio.sleep(1)  # Simulation de nettoyage asynchrone

# Utilisation avec `async with`
async def main():
    async with AsyncContext():
        print("Dans le bloc async with")
        await asyncio.sleep(2)  # Simulation d'une tâche

asyncio.run(main())
```

✅ **Pourquoi cette méthode ?** - Utilise `asyncio.sleep()` pour simuler une tâche asynchrone. - Gère proprement l'entrée et la sortie du contexte avec `async with`.

2. Utilisation de `contextlib.asynccontextmanager` (plus simple)

Une alternative plus concise consiste à utiliser `contextlib.asynccontextmanager`, qui permet d'écrire un context manager asynchrone sous forme de fonction avec `yield`.

```
import asyncio
from contextlib import asynccontextmanager

@asynccontextmanager
async def async_manager():
    print("Entrée dans le contexte")
    await asyncio.sleep(1)  # Simulation d'une tâche d'initialisation
    yield
    print("Sortie du contexte")
    await asyncio.sleep(1)  # Simulation de nettoyage

# Utilisation :
async def main():
    async with async_manager():
        print("Dans le bloc async with")
        await asyncio.sleep(2)  # Simulation d'une autre tâche

asyncio.run(main())
```

✅ **Pourquoi utiliser `asynccontextmanager` ?** - **Moins de code** qu'avec `__aenter__` et `__aexit__`. - **Facile à comprendre** et rapide à écrire. - Permet **d'éviter la création d'une classe** inutile.

---

3. Cas d'utilisation : Gestion d'une connexion HTTP asynchrone avec `aiohttp`

Les **context managers asynchrones** sont très utilisés dans la gestion des connexions réseau.

```
import aiohttp
import asyncio

class AsyncHTTPClient:
    async def __aenter__(self):
        """Crée une session HTTP asynchrone"""
        print("Ouverture de la session HTTP")
        self.session = aiohttp.ClientSession()
        return self.session

    async def __aexit__(self, exc_type, exc_value, traceback):
        """Ferme la session HTTP proprement"""
        print("Fermeture de la session HTTP")
        await self.session.close()

# Utilisation :
async def fetch():
    async with AsyncHTTPClient() as session:
        async with session.get("https://jsonplaceholder.typicode.com/tod
            data = await response.json()
            print("Données reçues :", data)

asyncio.run(fetch())
```

✅ **Pourquoi utiliser un context manager ici ?** - Assure **l'ouverture et la fermeture propre de la connexion HTTP**. - **Évite les fuites de ressources** en s'assurant que `.close()` est toujours appelé. - **Facilite la lecture et la maintenance du code**.

📌 Comparatif des approches

Méthode	Cas d'utilisation	Avantages	Inconvénients
Classe avec `__aenter__` et `__aexit__`	Connexions, tâches asynchrones complexes	Plus de contrôle	Plus de code
`asynccontextmanager`	Tâches simples	Syntaxe concise	Moins de flexibilité
Gestion HTTP avec `aiohttp`	Requêtes réseau asynchrones	Fermeture propre de la session	Nécessite `aiohttp`

📌 Conclusion

- **Besoin d'un context manager asynchrone simple ?** → `asynccontextmanager`
- **Besoin d'un contrôle plus fin sur les ressources ?** → Classe avec `__aenter__` et `__aexit__`
- **Gestion de connexions HTTP, bases de données, sockets ?** → Utiliser un context manager asynchrone pour éviter les fuites de ressources.