Cas pratique : Gestion d’un thread en Python

Lancer et arrêter un thread manuellement peut être risqué si on oublie de le fermer proprement. Un **context manager** permet de garantir que le thread se termine proprement, même en cas d’erreur.

---

1. Sans context manager (gestion manuelle d'un thread)

```
import threading
import time

class MyThread:
    def __init__(self):
        self.thread = None
        self.running = False

    def start(self):
        """Démarre le thread"""
        if self.thread is None:
            self.running = True
            self.thread = threading.Thread(target=self.run)
            self.thread.start()

    def run(self):
        """Tâche exécutée par le thread"""
        while self.running:
            print("Thread en cours d'exécution...")
            time.sleep(1)

    def stop(self):
        """Arrête le thread proprement"""
        self.running = False
        if self.thread:
            self.thread.join()  # Attendre la fin du thread
            self.thread = None

# Utilisation sans context manager
my_thread = MyThread()
my_thread.start()
time.sleep(3)  # Laisser le thread tourner un peu
my_thread.stop()  # Arrêter le thread proprement
```

❌ Problèmes de cette approche :

1. **Risque d'oubli** : Il faut explicitement appeler `stop()`, sinon le thread continue de tourner.
2. **Gestion manuelle du thread** : Nécessite de vérifier si `start()` et `stop()` sont bien appelés.
3. **Moins sécurisé** : Si une exception est levée avant `stop()`, le thread peut ne jamais s'arrêter.

---

2. Avec un context manager (gestion automatique

du thread)

On va transformer la classe en **context manager** pour s'assurer que le thread est toujours arrêté après son utilisation.

```
class ThreadManager:
    def __init__(self):
        self.thread = None
        self.running = False

    def __enter__(self):
        """Démarre automatiquement le thread"""
        self.running = True
        self.thread = threading.Thread(target=self.run)
        self.thread.start()
        return self  # Retourne l'instance pour pouvoir appeler `log()`

    def run(self):
        """Tâche exécutée par le thread"""
        while self.running:
            print("Thread en cours d'exécution...")
            time.sleep(1)

    def __exit__(self, exc_type, exc_value, traceback):
        """Arrête automatiquement le thread à la sortie du `with`"""
        self.running = False
        self.thread.join()  # Attendre la fin propre du thread

# Utilisation avec `with`
with ThreadManager():
    time.sleep(3)  # Laisser le thread tourner un peu
    print("Fin du bloc `with`, arrêt automatique du thread")
```

✅ Avantages du context manager pour un thread :

1. **Arrêt automatique** du thread lorsque le bloc `with` se termine.
2. **Évite les oublis** de `stop()` en assurant que `__exit__` sera toujours exécuté.
3. **Plus sécurisé** : Si une exception est levée, `__exit__` arrête le thread proprement.
4. **Moins de code répétitif** : Plus besoin d'appeler `start()` et `stop()` séparément.

Cas d'erreur : Pourquoi le context manager est plus robuste ?

Prenons un cas où une exception survient :

Sans context manager (peut poser problème)

```
my_thread.start()
raise RuntimeError("Erreur imprévue")  # Le thread ne sera jamais arrêté !
my_thread.stop()
```

Ici, **le thread continue de tourner** après l'erreur, ce qui peut poser problème.

Avec context manager (problème évité)

```
with ThreadManager():
    raise RuntimeError("Erreur imprévue")  # Pas de problème : `__exit__`
```

Grâce au **context manager**, le thread sera **toujours arrêté** proprement, même en cas d'exception.

📌 Conclusion

- **Besoin d'exécuter un thread temporairement ?** → Utiliser un **context manager** garantit son arrêt propre.
- **Moins de risques d'oublis** → `__exit__` s'exécute toujours.
- **Code plus lisible et sûr** → Plus besoin d'appeler manuellement `start()` et `stop()`.