

Cas pratique : Gestion d'un fichier journal (log)

Imaginons que nous ayons besoin d'écrire des logs dans un fichier. Nous allons comparer deux implémentations : 1. **Sans context manager** : gestion manuelle de l'ouverture et de la fermeture du fichier. 2. **Avec un context manager** : automatisation de la gestion des ressources.

////////////////////////////////////

1. Sans context manager (gestion manuelle)

Voici comment on pourrait gérer un fichier de log **sans** context manager :

```
class Logger:
    def __init__(self, filename):
        self.filename = filename
        self.file = None

    def open(self):
        """Ouvre le fichier pour écrire les logs"""
        self.file = open(self.filename, "a")

    def log(self, message):
        """Écrit un message dans le fichier log"""
        if self.file is None:
            raise RuntimeError("Le fichier n'est pas ouvert !")
        self.file.write(message + "\n")

    def close(self):
        """Ferme le fichier log"""
        if self.file:
            self.file.close()
            self.file = None

# Utilisation
logger = Logger("app.log")

# Il faut penser à ouvrir et fermer correctement le fichier
logger.open()
logger.log("Démarrage de l'application")
logger.log("Une action a été effectuée")
logger.close()
```

❌ Problèmes de cette approche :

1. **Oubli de fermeture** : Si une erreur survient entre `open()` et `close()`, le fichier peut rester ouvert indéfiniment.
2. **Répétition de code** : On doit appeler `open()` avant `log()` et `close()` après.
3. **Moins robuste** : On doit s'assurer que `close()` est bien appelé, même en cas d'erreur.

2. Avec un context manager

On va maintenant transformer la classe en **context manager** pour gérer automatiquement l'ouverture et la fermeture du fichier.

```
class Logger:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        """Ouverture automatique du fichier"""
        self.file = open(self.filename, "a")
        return self

    def log(self, message):
        """Écrit un message dans le fichier log"""
        self.file.write(message + "\n")

    def __exit__(self, exc_type, exc_value, traceback):
        """Fermeture automatique du fichier"""
        self.file.close()

# Utilisation avec `with`
with Logger("app.log") as logger:
    logger.log("Démarrage de l'application")
    logger.log("Une action a été effectuée")
```

✅ Avantages de l'approche avec context manager :

1. **Plus sûr** : La fermeture du fichier est garantie, même si une exception survient.
 2. **Moins de code répétitif** : Plus besoin d'appeler `open()` et `close()` manuellement.
 3. **Plus lisible** : L'ouverture et la fermeture sont gérées de manière implicite.
-

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

Prenons un cas où une erreur survient après l'ouverture du fichier.

Sans context manager (peut poser problème)

```
logger.open()  
logger.log("Démarrage")  
raise RuntimeError("Erreur imprévue") # Le fichier ne sera jamais fermé !  
logger.close()
```

Le fichier **reste ouvert** car `close()` ne sera jamais atteint.

Avec context manager (problème évité)




```
with Logger("app.log") as logger:  
    logger.log("Démarrage")  
    raise RuntimeError("Erreur imprévue") # Pas de problème : __exit__
```

Même si une exception survient, **le fichier sera fermé proprement**.



Conclusion

Si une ressource (fichier, connexion, verrou, etc.) **doit être libérée après usage, le context manager est la meilleure approche** :

-  Il assure la fermeture automatique de la ressource.
-  Il simplifie le code et évite les oublis.
-  Il est plus robuste en cas d'erreurs.