

# Les Context managers

---

structure permettant de gérer des ressources (fichiers, connexions, verrous, etc.) de manière sécurisée à l'aide des blocs `with`.

## 1 Pourquoi utiliser un context manager plutôt qu'un simple `try/finally` ?

- Permet de discuter des avantages du context manager, comme l'amélioration de la lisibilité, la gestion automatique des ressources et l'encapsulation de la logique de nettoyage.

## 2 Quels sont les cas d'usage des context managers ?

- Cela peut faire émerger des discussions sur des usages avancés, comme la gestion du verrouillage ( `threading.Lock()` ), la temporisation ( `timeit` ), la modification temporaire d'un état (ex. `decimal.getcontext()` ), ou encore des outils comme `contextlib.ExitStack`.

## 3 Comment créer un context manager ?

- Cette question amène à parler de l'implémentation manuelle d'un context manager avec une classe et les méthodes `__enter__()` et `__exit__()`.

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

### 1. Avec une classe et `__enter__` / `__exit__`

C'est la manière la plus traditionnelle.

```

class MyContext:
    def __enter__(self):
        print("Entrée dans le contexte")
        return self # Valeur retournée accessible avec `as` dans `with`

    def __exit__(self, exc_type, exc_value, traceback):
        print("Sortie du contexte")
        if exc_type:
            print(f"Une exception {exc_type} a été levée : {exc_value}")
        return False # Propager ou non l'exception (False = propagée)

# Utilisation :
with MyContext() as ctx:
    print("Dans le bloc with")

```

## 2. Avec `contextlib.ContextDecorator` pour un décorateur

Cela permet d'utiliser le context manager sous forme de décorateur.

```

from contextlib import ContextDecorator

class MyDecoratorContext(ContextDecorator):
    def __enter__(self):
        print("Début du contexte")

    def __exit__(self, exc_type, exc_value, traceback):
        print("Fin du contexte")

@MyDecoratorContext()
def my_function():
    print("Exécution de la fonction")

# Utilisation
my_function()

```

## 3. Avec `contextlib.contextmanager` (fonction générateur)

Utilisation d'un décorateur et `yield` pour simplifier la gestion.

```
from contextlib import contextmanager

@contextmanager
def my_context():
    print("Entrée dans le contexte")
    yield # Pause ici jusqu'à la fin du bloc `with`
    print("Sortie du contexte")

# Utilisation :
with my_context():
    print("Dans le bloc with")
```

---

## 4. Avec `ExitStack` pour gérer plusieurs contextes dynamiquement

Très utile pour gérer plusieurs contextes imbriqués.

```
from contextlib import ExitStack

with ExitStack() as stack:
    file1 = stack.enter_context(open("file1.txt", "w"))
    file2 = stack.enter_context(open("file2.txt", "w"))
    print("Les fichiers sont ouverts")
```

---

## 5. Avec une classe utilisant `@dataclass`

Approche moderne avec `dataclasses` pour simplifier l'écriture.

```
from dataclasses import dataclass

@dataclass
class MyContext:
    name: str


    def __enter__(self):
        print(f"Entrée dans le contexte : {self.name}")
        return self


    def __exit__(self, exc_type, exc_value, traceback):
        print(f"Sortie du contexte : {self.name}")


# Utilisation
with MyContext("Test"):
    print("Dans le bloc with")
```

Voici un comparatif des différentes méthodes de création de context managers en Python, avec leurs cas d'utilisation, avantages et inconvénients.


## 1. Classe avec `__enter__` et `__exit__`

 **Cas d'utilisation** : - Lorsque l'on a besoin d'une gestion fine des ressources (fichiers, connexions, transactions...). - Lorsqu'on veut encapsuler un état ou des données associées au contexte.

 **Avantages** : - Offre un contrôle total sur l'entrée et la sortie du contexte. - Peut encapsuler des états complexes. - Compatible avec l'héritage pour réutiliser du code.

 **Inconvénients** : - Nécessite une classe et deux méthodes ( `__enter__` et `__exit__` ), ce qui alourdit un peu le code. - Plus verbeux que la solution avec `contextlib`.

## 2. `contextlib.contextmanager` (fonction générateur)

 **Cas d'utilisation** : - Lorsqu'on veut un context manager simple et léger. - Idéal pour encapsuler des ressources sans maintenir d'état interne complexe.

 **Avantages** : - Syntaxe concise (pas besoin de définir une classe). - Plus lisible pour des

cas simples. - Moins de code répétitif.

**✗ Inconvénients** : - Moins flexible qu'une classe (difficile de gérer des états internes complexes). - Le `yield` coupe l'exécution, ce qui peut rendre le flux moins intuitif.

---

### 3. `ContextDecorator` pour les décorateurs

---

**📌 Cas d'utilisation** : - Quand on veut encapsuler du code dans un contexte sans modifier le code appelant. - Très utile pour la mesure du temps, la gestion de logs, ou la gestion de connexions.

**✓ Avantages** : - Permet d'appliquer un context manager à une fonction sans `with`. - Utile pour des tâches répétitives où un contexte est toujours requis.

**✗ Inconvénients** : - Ne fonctionne que pour les fonctions/méthodes décorées. - Moins flexible si le contexte doit être paramétrable dynamiquement.

---

### 4. `ExitStack` pour gérer plusieurs contextes

---

**📌 Cas d'utilisation** : - Lorsque plusieurs context managers doivent être ouverts et fermés dynamiquement. - Idéal pour ouvrir plusieurs fichiers, sockets ou connexions à la fois.

**✓ Avantages** : - Gère plusieurs context managers en une seule instruction. - Fonctionne même si le nombre de contextes n'est pas connu à l'avance.

**✗ Inconvénients** : - Introduit une légère complexité supplémentaire. - Moins utile pour des cas simples avec un seul contexte.

---

### 5. `dataclass` avec `__enter__` et `__exit__`

---

**📌 Cas d'utilisation** : - Quand le contexte encapsule des données structurées. - Utile pour des objets qui nécessitent des valeurs paramétrées lors de l'instanciation.

**✓ Avantages** : - Simplifie l'initialisation des attributs avec `dataclass`. - Réduit le code répétitif.

**✗ Inconvénients** : - Pas de bénéfice majeur si l'on ne manipule pas des données structurées. - Pas plus léger que la solution classique avec une classe.

---

## Résumé sous forme de tableau

| Méthode   | Cas d'utilisation                                 | Avantages                                   | Inconvénients                                      |
|---|---|---|--|
| Classe <code>__enter__</code> / <code>__exit__</code>                   | Gestion fine des ressources, encapsulation d'état | Contrôle total, extensible                  | Verbeux, plus long à écrire                        |
| <code>contextlib.contextmanager</code>                                  | Simplicité, gestion de ressources légère          | Moins de code, plus lisible                 | Moins flexible pour les états complexes            |
| <code>ContextDecorator</code>   | Ajout de contexte à des fonctions                 | Facile à appliquer, évite <code>with</code> | Moins flexible pour des paramètres dynamiques      |
| <code>ExitStack</code>  | Gestion dynamique de plusieurs contextes          | Gère plusieurs contextes simultanément      | Introduit une complexité supplémentaire            |
| <code>dataclass</code> + <code>__enter__</code> / <code>__exit__</code> | Contexte structuré avec des données               | Réduit le code répétitif                    | Peu d'avantages par rapport à une classe classique |



## Conclusion : Quelle méthode choisir ?

Besoin d'un contrôle total → Classe `__enter__` / `__exit__`

Simplicité et lisibilité → `contextlib.contextmanager`

Automatisation avec un décorateur → `ContextDecorator`

Gestion de plusieurs contextes dynamiquement → `ExitStack`

Encapsulation de données avec état → `dataclass` + `__enter__` / `__exit__`

