ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Projet Pyhton

Analyse des Avis et Alertes ANSSI avec Enrichissement des CVE

Le projet a pour but de développer un outil permettant d'analyser et d'enrichir les données sur les vulnérabilités et menaces en cybersécurité publiées par l'ANSSI, afin d'aider à mieux identifier, prioriser et prévenir les risques, tout en générant des alertes personnalisées pour renforcer la sécurité des systèmes informatiques.

Auteurs : Fawzi ELGHAZOUI,
Diane DUSEILLIER,
Mathilde DESTAILLEUR

CONTEXTE DU PROJET

La cybersécurité est devenue un enjeu majeur pour les entreprises et les organisations du monde entier face à la multiplication des attaques informatiques. Les vulnérabilités logicielles et matérielles constituent une porte d'entrée privilégiée pour les attaquants, rendant impératif leur identification rapide et leur correction efficace.

En France, l'**ANSSI** (Agence Nationale de la Sécurité des Systèmes d'Information joue un rôle central dans la veille et la diffusion d'informations sur les menaces. Elle publie régulièrement des bulletins de sécurité (CERT-FR – Centre gouvernemental de veille, d'alerte et de réponse aux attaques informatiques) visant à informer les entreprises et les particuliers sur les vulnérabilités existantes et les risques associés. Les principaux types de bulletin sont

Les avis de sécurité. Ils signalent des vulnérabilités connues et fournissent des recommandations concrètes pour les corriger ou atténuer leurs effets. Ils permettent aux organisations d'agir préventivement pour sécuriser leurs systèmes;

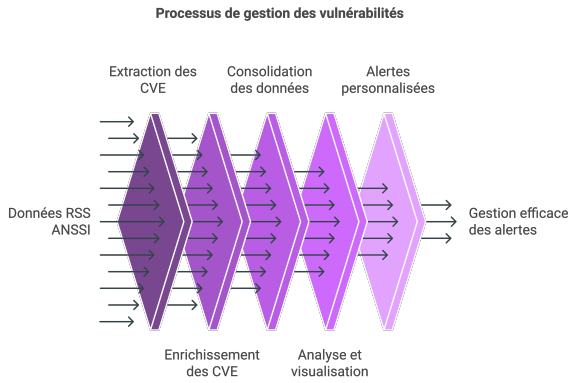
Les alertes. Elles concernent les vulnérabilités critiques qui sont activement exploitées par des acteurs malveillants. Ces vulnérabilités nécessitent une intervention urgente pour éviter d'éventuelles compromissions de sécurité.

Ces bulletins contiennent des identifiants **CVE** (*Common Vulnerabilities and Exposures*) qui permettent de référencer précisément chaque vulnérabilité.

Contrairement à son homologue américain, le **NIST** (*National Institute of Standards and Technology*), qui propose une **API** (*Application Programming Interface*) dédiée et complète (NVD API) permettant de collecter et d'analyser facilement ce type d'informations, il est plus difficile d'automatiser le traitement des flux publiés par l'ANSSI. En effet, l'ANSSI met uniquement à disposition un flux **RSS** (*Rich Site Summary*) relativement sommaire (ANSSI RSS Feed) destiné aux entreprises et aux particuliers. Les informations détaillées nécessitent de naviguer dans le **DOM** (*Document Object Model*) des pages *web* ou directement dans le **JSON** (*JavaScript Object Notation*) mentionné dans ces flux pour être extraites et exploitées.

De plus, contrairement au NIST qui offre des fonctionnalités avancées, telles que des interprétations statistiques automatisées ou des systèmes d'alertes personnalisées, l'ANSSI ne permet pas, dans son format actuel, de générer des statistiques avancées ni des alertes sur mesure en fonction des besoins spécifiques des utilisateurs. Ce manque d'automatisation et de flexibilité justifie pleinement la réalisation d'un outil capable de traiter, d'enrichir et d'analyser ces données pour en tirer des conclusions exploitables.

OBJECTIF DU PROJET



PRÉREQUIS

ENVIRONNEMENT PYTHON

L'exécution du code nécessite **Python 3.7** ou une version plus récente. Pour vérifier votre version de **Python**, il suffit d'exécuter la commande suivante dans n'importe quelle **CLI** (*Command Line Interface*)

```
python --version
```

Attention. Selon votre environnement de travail, vous pouvez être amené à avoir une erreur vous indiquant que la commande n'existe pas. La commande suivante devrait alors résoudre votre problème

```
python3 --version
```

Si vous utilisez une version antérieure, vous devrez mettre à jour **Python**.

DÉPENDANCES

Avant d'exécuter le code, il vous faut installer les bibliothèques **Python** nécessaires. On présente dans cette section les étapes pour installer toutes les dépendances nécessaires.

Création d'un environnement virtuel.
(optionnel, mais recommandé) L'utilisation d'un environnement virtuel permet de maintenir des dépendances propres et spécifiques à un projet sans interférer avec d'autres projets **Python**.

Pour créer un environnement virtuel, exécutez la commande suivante dans votre CLI

```
python -m venv venv
```

Ensuite, activez l'environnement virtuel avec la commande suivant votre système d'exploitation

sous Windows :

```
venv\Scripts\activate
```

sous Linux/MacOs :

```
source venv/bin/activate
```

Installation des dépendances. Afin que le code se lance correctement, vous devez installer les bibliothèques requises. Pour être sûr que le code ne dépende que du dossier dans lequel il se trouve, veilliez à créer un dossier pour le lancement du projet en vous assurant de mettre les fichiers de ce dernier dans la racine du dossier. Enfin, naviguez via votre CLI dans vos répertoires afin de vous assurez d'être à la racine du dossier. Ainsi, exécutez cette commande

```
pip install aiohttp feedparser pandas flask
    tqdm
```

Cela installera les bibliothèques `aiohttp`; `feedparser`; `pandas`; `flask`; `tqdm` dans le dossier du projet.

Configuration des URLs et des APIs.
RSS URL. Le code est configuré pour récupérer des informations à partir du flux RSS de l'ANSSI, qui addresse les alertes ou les avis. Cette URL est déjà définie dans le code : le choix se fait lors de l'appel du code html.
API MITRE. L'application fait des requêtes vers l'API MITRE pour récupérer des informations supplémentaires sur chaque CVE. Il n'y a pas besoin de clé API pour cette API, donc aucune configuration supplémentaire n'est nécessaire.
API FIRST. L'application fait également des requêtes vers l'API de FIRST pour obtenir des scores EPSS pour les CVEs. L'API est publique et ne nécessite pas de clé d'authentification.

Exécution du Code. Une fois les bibliothèques installées, vous pouvez exécuter le code **Flask** en local. Pour démarrer l'application, exécutez la commande suivante dans votre CLI

```
python main.py
```

Le serveur **Flask** démarrera et sera accessible à l'adresse suivante

`http://127.0.0.1:5000/`

Dépannage.
Problèmes de commande. Si, à l'exécution d'une commande de type

```
python ...
```

une erreur vous indique que la commande n'existe pas, veuillez réessayer avec la commande

```
python3 ...
```

Problèmes d'installation des bibliothèques. Si vous rencontrez des erreurs lors de l'installation des bibliothèques, assurez-vous que votre version de `pip` est à jour. Vous pouvez mettre à jour `pip` avec

```
pip install --upgrade pip
```

Problèmes de réseau. Si les requêtes **HTTP** (*Hypertext Markup Language*) échouent, vérifiez votre connexion Internet et assurez-vous que l'API cible est en ligne.

BIBLIOTHÈQUES

1. asyncio

Rôle. `asyncio` est une bibliothèque standard de **Python** pour la programmation asynchrone. Elle permet d'écrire du code concurrent (c'est-à-dire, effectuer plusieurs tâches en même temps) en utilisant des coroutines.
Utilisation dans le code. L'application utilise `asyncio` pour gérer des requêtes HTTP asynchrones et exécuter plusieurs tâches (comme récupérer des données sur plusieurs CVEs) simultanément. Les fonctions comme `async def` et `await` permettent de rendre les opérations non-bloquantes, améliorant ainsi la performance du programme (notamment lors de l'appel aux API externes).

2. aiohttp

Rôle. `aiohttp` est une bibliothèque **Python** pour effectuer des requêtes HTTP de manière asynchrone, essentiel dans un contexte de programmation asynchrone avec `asyncio`. Elle permet de faire des appels réseau de manière non-bloquante et d'exécuter d'autres tâches pendant que la réponse est en attente.
Utilisation dans le code. `aiohttp` est utilisé pour envoyer des requêtes HTTP aux différentes API (comme l'API de MITRE et FIRST pour récupérer les données CVE, EPSS, *etc.*). La gestion des connexions HTTP est effectuée avec `aiohttp.ClientSession`, et la gestion des *timeouts* est gérée avec `aiohttp.ClientTimeout`.

3. feedparser

Rôle. `feedparser` est une bibliothèque **Python** simple pour analyser des flux RSS. Elle permet de récupérer et d'analyser des données au format RSS, qui est souvent utilisé pour la syndication de contenu (par exemple, des alertes de sécurité).
Utilisation dans le code. `feedparser` est utilisé dans la méthode `parse_rss` pour récupérer et analyser le

flux RSS du site ANSSI. Le flux contient les alertes de sécurité, que le code extrait et transforme en une liste de dictionnaires avec les titres, les liens, le type de bulletin, et les dates de publication.

4. datetime

Rôle. La bibliothèque `datetime` est utilisée pour travailler avec des dates et des heures en **Python**.
Utilisation dans le code. `datetime.strptime` est utilisé pour analyser la date au format texte (en provenance du flux RSS) et la convertir en un objet *date*. Ensuite, la date est convertie en format ISO (année-mois-jour) avec `.isoformat()`.

5. pandas

Rôle. `pandas` est une bibliothèque puissante de manipulation de données en **Python**. Elle fournit des structures de données flexibles, comme les DataFrame, qui facilitent le travail avec des données tabulaires.
Utilisation dans le code. `pandas` est utilisé pour organiser et structurer les données récupérées sous forme de tableau. À la fin du traitement des CVEs, les informations sont converties en un DataFrame, puis renvoyées sous forme de JSON via **Flask**. Les DataFrame permettent de manipuler facilement les données, de les nettoyer (par exemple, remplir les valeurs manquantes) et de les convertir en un format JSON pour l'API.

6. flask

Rôle. Flask est un framework *web* léger pour **Python**. Il est utilisé pour construire des applications web. Avec **Flask**, il est facile de créer des routes pour gérer les requêtes HTTP et répondre avec du contenu dynamique (comme des pages HTML ou des données JSON).
Utilisation dans le code. Flask est utilisé pour créer l'API *web* qui gère les requêtes sur les *endpoints* `/.` (pour afficher la page d'accueil) et `/fetch_data` (pour récupérer les données des CVEs). Les données récupérées sont formatées en JSON et renvoyées via la méthode `jsonify` de **Flask**.

7. tqdm et tqdm.asyncio

Rôle. `tqdm` est une bibliothèque **Python** permettant d'afficher des barres de progression dans les boucles. Elle est très utile pour suivre l'avancement de processus longs.
Utilisation dans le code. `tqdm.asyncio` permet d'afficher une barre de progression dans les tâches asynchrones, comme la récupération des données depuis les API de MITRE et FIRST. La fonction `tqdm_asyncio.gather` permet de suivre la progression des requêtes asynchrones pendant qu'elles sont exécutées en parallèle.

8. re

Rôle. `re` est une bibliothèque **Python** pour travailler avec les expressions régulières. Elle permet de rechercher, de manipuler et de remplacer des motifs dans des chaînes de caractères.
Utilisation dans le code. `re.sub(r'\(.*?\)', '', entry.title)` est utilisé pour nettoyer les titres des alertes RSS en supprimant les textes entre parenthèses. Par exemple, si un titre contient `(mise à jour)`, cette partie sera supprimée.

9. typing

Rôle. La bibliothèque `typing` fournit des annotations de types pour les fonctions **Python**, ce qui permet de rendre le code plus lisible et d'améliorer la vérification des types pendant le développement.
Utilisation dans le code. `-> List[Dict]` est utilisé pour indiquer que la fonction `parse_rss` retourne une liste de dictionnaires. Cela aide à mieux comprendre le type de données attendu et facilite le développement avec des outils comme les éditeurs de code qui utilisent les annotations de type.

10. aiohttp.ClientTimeout

Rôle. `aiohttp.ClientTimeout` est utilisé pour configurer le délai d'attente des requêtes HTTP dans `aiohttp`. Il définit combien de temps attendre avant que la requête échoue si elle prend trop de temps.
Utilisation dans le code. Le `TIMEOUT` est défini avec `ClientTimeout(total=10)`, ce qui signifie que si une requête prend plus de 10 secondes, elle échouera. Cela permet de gérer les erreurs de réseau et d'éviter que l'application reste bloquée en cas de lenteur.

11. aiohttp.TCPConnector

Rôle. `TCPConnector` permet de configurer la manière dont les connexions réseau sont établies et gérées avec `aiohttp`. Cela inclut le contrôle du nombre de connexions simultanées, la mise en cache **DNS** (*Domain Name System*), et d'autres paramètres réseau.
Utilisation dans le code. `TCPConnector(limit=MAX_CONCURRENT, ttl_dns_cache=300)` est utilisé pour configurer une limite maximale de connexions simultanées (définie par `MAX_CONCURRENT`) et spécifier que les résultats de cache DNS doivent être conservés pendant 5 minutes (300 secondes).

STRUCTURE DU CODE

CODE PYTHON

_compute_threat_vector

La fonction `_compute_threat_vector` génère un vecteur de menace à partir des informations d'un CVE. Ce vecteur permet de caractériser la sévérité et le type de la vulnérabilité basée sur les informations disponibles, telles que la description et le score CVSS.

1. Définition

```
def _compute_threat_vector(self, cve_data: Dict[str, Any]) -> List[str]:
    threat_vector = []

    if 'description' in cve_data:
        description = cve_data['description'].lower()
        if 'remote' in description:
```

```
            threat_vector.append('remote')
        if 'denial of service' in description:
            threat_vector.append('dos')

    if 'cvss' in cve_data:
        score = cve_data['cvss']['base_score']
        if score >= 9:
            threat_vector.append('critical')
        elif score >= 7:
            threat_vector.append('high')
        elif score >= 4:
            threat_vector.append('medium')
        else:
            threat_vector.append('low')

    return threat_vector
```

2. Fonctionnement détaillé

a. Initialisation du vecteur de menace

```
threat_vector = []
```

La méthode commence par initialiser une liste vide `threat_vector`. Cette liste sera utilisée pour stocker les différents éléments décrivant la menace, tels que le type de vulnérabilité et son niveau de gravité.

b. Vérification de la description et ajout des termes

```
if 'description' in cve_data:
    description = cve_data['description'].
        lower()
    if 'remote' in description:
        threat_vector.append('remote')
    if 'denial of service' in description:
        threat_vector.append('dos')
```

La fonction vérifie si la clé `'description'` existe dans les données du CVE (`cve_data`). Si c'est le cas, elle convertit la description en minuscules pour faciliter la recherche de termes spécifiques. Ensuite, elle cherche les termes “remote” et “denial of service” dans la description, et ajoute les éléments correspondants au vecteur de menace si ces termes sont présents.

c. Vérification du score CVSS et ajout du niveau de gravité

```
if 'cvss' in cve_data:
    score = cve_data['cvss']['base_score']
    if score >= 9:
        threat_vector.append('critical')
    elif score >= 7:
        threat_vector.append('high')
    elif score >= 4:
        threat_vector.append('medium')
    else:
        threat_vector.append('low')
```

La fonction vérifie ensuite si la clé `'cvss'` existe dans les données du CVE. Si elle est présente, elle extrait le `base_score` du CVSS et classe le score en fonction de seuils prédéfinis (9 pour ”critical”, 7 pour ”high”, 4 pour ”medium”, et moins de 4 pour ”low”). Le niveau de gravité est ensuite ajouté au vecteur de menace.

d. Retour du vecteur de menace

```
return threat_vector
```

Enfin, la méthode retourne la liste `threat_vector`, qui contient les éléments représentant les caractéristiques de la menace, basées sur la description et le score CVSS.

3. Pourquoi cette méthode est utile ?

- **Caractérisation des menaces.** Elle permet de catégoriser une vulnérabilité en fonction de ses risques potentiels (ex : remote, dos, etc.) et de sa gravité (score CVSS).
- **Structuration des données.** Elle transforme des données brutes (description et score CVSS) en une structure facile à analyser et à utiliser dans d'autres parties du programme.
- **Précision et souplesse.** La méthode est conçue pour identifier des termes spécifiques dans la description et ajuster le niveau de gravité en fonction de critères standardisés, ce qui permet une analyse objective des menaces.

La classe `CVE_DataProcessor_Engine`

La classe `CVE_DataProcessor_Engine` est conçue pour traiter et analyser les données relatives aux vulnérabilités du type CVE (Common Vulnerabilities and Exposures). Elle regroupe un ensemble de méthodes et de mécanismes permettant de gérer et de traiter des informations sur les CVE, en les extrayant, les analysant, et en en tirant des conclusions sur la gravité et les caractéristiques des vulnérabilités.

1. Objectifs de la classe

La classe `CVE_DataProcessor_Engine` a pour objectif de faciliter la gestion des données relatives aux vulnérabilités, en particulier celles issues des bases de données publiques telles que NIST ou autres sources RSS. Elle offre des fonctionnalités pour :

- **Extraction des données.** La classe est capable de récupérer les informations pertinentes liées à un CVE via des flux RSS ou des API.
- **Analyse des vulnérabilités.** Elle permet d'analyser les CVE en fonction de critères tels que la description, le score CVSS, les composants affectés, etc.
- **Traitement des données.** Elle intègre des mécanismes pour filtrer, nettoyer et structurer les informations récupérées afin d'en faciliter l'exploitation dans des outils ou des rapports.
- **Classification des menaces.** Elle catégorise les vulnérabilités en fonction de leur sévérité et de leur type, pour permettre une gestion rapide des priorités.

2. Structure générale de la classe

La classe `CVE_DataProcessor_Engine` repose sur plusieurs composants et bibliothèques clés pour accomplir ses objectifs :

- **Gestion des connexions réseau.** Elle utilise des bibliothèques telles qu'`aiohttp` pour effectuer des requêtes HTTP asynchrones, ce qui lui permet de récupérer les données de manière efficace.
- **Gestion des ressources.** La classe intègre des mécanismes pour la gestion des ressources (par exemple, des sémaphores) pour éviter une surcharge de connexions simultanées lors de l'accès aux sources de données.
- **Analyse et transformation des données.** Elle effectue des opérations sur les données, comme la transformation de flux RSS en structures de données manipulables (listes, dictionnaires).

3. Processus global de traitement

Le processus global de traitement dans la classe peut être résumé comme suit :

- **Initialisation.** La classe est configurée avec des paramètres de connexion et des mécanismes de contrôle de la concurrence. Elle établit des connexions réseau (par exemple, une session HTTP) pour récupérer les données nécessaires.
- **Récupération des données.** Les informations liées aux CVE sont extraites via des API ou des flux RSS, ce qui permet à la classe d'accéder à des bases de données publiques ou à des sources personnalisées.
- **Traitement des données.** Une fois les données récupérées, elles sont analysées et structurées selon les besoins. Par exemple, les descriptions des vulnérabilités peuvent être extraites et nettoyées, et les scores CVSS sont utilisés pour évaluer la gravité des vulnérabilités.
- **Présentation des résultats.** Enfin, les informations traitées sont prêtes à être utilisées par d'autres modules du programme, qui peuvent être chargés de produire des rapports ou de prendre des décisions basées sur l'analyse des vulnérabilités.

4. Pourquoi cette classe est utile ?

La classe `CVE_DataProcessor_Engine` offre plusieurs avantages clés dans le contexte de la gestion des vulnérabilités :

- **Automatisation du traitement des CVE.** Elle permet de traiter automatiquement les données des vulnérabilités, en réduisant le besoin d'intervention manuelle et en accélérant l'analyse.
- **Analyse centralisée.** Elle regroupe toutes les étapes d'analyse des CVE (extraction, nettoyage, analyse), permettant une gestion centralisée des vulnérabilités.
- **Prise en charge de différentes sources de données.** Elle peut travailler avec diverses sources, telles que les flux RSS, les API, ou d'autres systèmes, ce qui lui permet d'être flexible dans la récupération des informations.
- **Scalabilité.** Grâce à l'utilisation de techniques asynchrones et de sémaphores, elle est capable de traiter un grand volume de données sans saturer les ressources systèmes.

La fonction `__init__`

La fonction `__init__` est le constructeur de la classe `CVE_DataProcessor_Engine`. Elle est exécutée automatiquement lors de la création d'une nouvelle instance de la classe. Son rôle principal est d'initialiser les variables d'instance et de préparer l'environnement nécessaire à l'exécution des autres méthodes de la classe.

1. Définition

```
1 def __init__(self, max_concurrent: int = 5):
2     self.max_concurrent = max_concurrent
3     self.session = None
4     self.semaphore = asyncio.Semaphore(
          max_concurrent)
```

2. Fonctionnement détaillé

La fonction `__init__` initialise plusieurs attributs importants pour la classe `CVE_DataProcessor_Engine` :

- `self.max_concurrent` : Cet attribut définit le nombre maximal de connexions simultanées autorisées lors de la récupération des données. Il est initialisé avec la valeur par défaut de `5`, mais peut être modifié lors de la création de l'instance.
- `self.session` : Cet attribut sera utilisé pour stocker la session HTTP asynchrone créée plus tard, permettant d'effectuer des requêtes réseau. Il est initialisé à `None`, car la session n'est pas encore ouverte.
- `self.semaphore` : Le sémaphore est utilisé pour limiter le nombre de requêtes HTTP concurrentes. Il est créé à l'aide de `asyncio.Semaphore(max_concurrent)`, où `max_concurrent` définit le nombre maximal de tâches simultanées autorisées.

3. Objectifs de la fonction `__init__`

La fonction `__init__` prépare l'objet pour les opérations futures, en définissant :

- La configuration du nombre maximal de connexions simultanées (`max_concurrent`).
- La création du sémaphore pour contrôler la concurrence des tâches asynchrones.

- L'initialisation de la session HTTP `session`, qui sera ouverte lors de l'utilisation de la classe.

Cela garantit que l'instance de la classe est prête à récupérer et traiter des données de manière efficace, tout en contrôlant le nombre de connexions simultanées.

4. Exemple d'utilisation

Lorsque vous créez une nouvelle instance de la classe, la fonction `__init__` est automatiquement appelée :

```
engine = CVE_DataProcessor_Engine(
    max_concurrent=10)
```

Dans cet exemple, `max_concurrent` est défini sur 10, ce qui signifie que la classe pourra effectuer jusqu'à 10 connexions simultanées pour récupérer les données.

La fonction `__aenter__`

La fonction `__aenter__` est utilisée dans le cadre des contextes asynchrones. Elle est exécutée lorsque l'exécution entre dans un bloc `async with`, permettant de configurer et d'initialiser les ressources nécessaires avant d'effectuer des opérations asynchrones.

1. Définition

```
async def __aenter__(self):
    self.session = await aiohttp.ClientSession
        ().__aenter__()
    return self
```

2. Fonctionnement détaillé

La fonction `__aenter__` réalise plusieurs actions importantes :

- **Création de la session HTTP** :

  ```
      self.session = await aiohttp.
          ClientSession().__aenter__()
  ```

  Cette ligne crée une session HTTP asynchrone via la bibliothèque `aiohttp`. La méthode `__aenter__` de `ClientSession` est appelée pour initialiser correctement la session. Cela permet de gérer les connexions HTTP de manière asynchrone et optimisée, réduisant ainsi la surcharge des connexions répétées.

- **Retour de l'objet `self`** :

  ```
      return self
  ```

  La fonction retourne l'instance de la classe elle-même (`self`), ce qui permet à l'utilisateur de travailler directement avec l'instance de la classe à l'intérieur du bloc `async with`.

3. Objectifs de la fonction `__aenter__`

La fonction `__aenter__` sert à préparer l'objet en initialisant la session HTTP avant d'exécuter des actions asynchrones à l'intérieur d'un bloc `async with`. Son rôle est de garantir que la session est prête à être utilisée et que les ressources nécessaires sont allouées correctement avant toute opération réseau.

Cela permet de faciliter la gestion des ressources réseau et de garantir qu'une session HTTP est activée avant l'exécution des opérations de traitement des données.

4. Exemple d'utilisation

Lors de l'utilisation de la classe avec un bloc `async with`, la fonction `__aenter__` est appelée automatiquement pour initialiser la session :

```
async with CVE_DataProcessor_Engine() as
    engine:
    # Le bloc de code ici peut utiliser l'
        instance de la classe avec la session
        active.
```

L'instance de `CVE_DataProcessor_Engine` est initialisée, et la session HTTP est ouverte à l'intérieur du bloc `async with`. À la fin du bloc, la session sera automatiquement fermée grâce à la fonction `__aexit__`.

La méthode `__aexit__`

La méthode `__aexit__` fait partie du protocole de gestion des contextes asynchrones. Elle est appelée lorsqu'une session est terminée, garantissant que les ressources associées à la session HTTP sont libérées correctement.

1. Définition

```
async def __aexit__(self, exc_type, exc_val,
    exc_tb):
    if self.session:
        await self.session.close()
```

2. Fonctionnement détaillé

La fonction `__aexit__` effectue plusieurs actions :

- **Vérification de l'existence de la session HTTP** :

  ```
      if self.session:
  ```

  Avant de tenter de fermer la session, la fonction vérifie si la session (`self.session`) existe. Cela permet de s'assurer qu'une session a été correctement initialisée dans la fonction `__aenter__`, et ainsi éviter toute tentative de fermeture d'une session inexistante.

- **Fermeture de la session HTTP** :

  ```
      await self.session.close()
  ```

  Si la session existe, la méthode `close()` est appelée de manière asynchrone sur la session `self.session`. Cela libère les ressources réseau et ferme proprement

toutes les connexions HTTP ouvertes par la session, garantissant ainsi que l'application ne laisse pas de connexions ouvertes ou de ressources non libérées après son utilisation.

- **Gestion des exceptions** : Les paramètres `exc_type`, `exc_val`, et `exc_tb` font partie du protocole de gestion des exceptions dans un bloc `async with`. Cependant, dans cette méthode, aucune gestion spécifique des exceptions n'est réalisée. Si une exception a été levée pendant l'exécution du bloc `async with`, elle sera propagée après la fermeture de la session.

3. Objectifs de la fonction `__aexit__`

La fonction `__aexit__` assure la fermeture propre des ressources. Dans ce cas, elle garantit que la session HTTP est correctement fermée une fois les opérations terminées ou en cas d'exception. Cela évite des fuites de mémoire ou des connexions ouvertes inutiles qui pourraient ralentir l'application ou provoquer des erreurs dans le futur.

Elle permet également de faciliter la gestion des erreurs : même si une exception survient, la session est fermée proprement, et l'exception est propagée pour être éventuellement capturée ailleurs dans le code.

4. Exemple d'utilisation

Lorsqu'un bloc `async with` se termine, la méthode `__aexit__` est appelée automatiquement pour fermer la session, même en cas d'exception :

```
async with CVE_DataProcessor_Engine() as engine:
    # Le code ici peut lever une exception, mais la session sera ferm e la fin
    # du bloc, garantissant la lib ration des ressources.
```

Si une exception est levée à l'intérieur du bloc `async with`, la méthode `__aexit__` est toujours exécutée pour fermer la session avant que l'exception ne soit propagée.

Explication du code LaTeX : Définition : Le code Python de la fonction `__aexit__` est montré, où une vérification de la session est effectuée avant de la fermer. La méthode close() est appelée de manière asynchrone pour fermer la session.

La fonction `_decode_rss_stream`

La fonction `_decode_rss_stream` est responsable de la lecture et du décodage d'un flux RSS à partir d'un flux de données brut. Elle extrait les informations pertinentes de chaque élément du flux et les structure dans un format exploitable, généralement sous forme de dictionnaire.

1. Définition

```
def _decode_rss_stream(self, response: bytes) -> List[Dict]:
    feed = feedparser.parse(response)
    return [{
        'title': entry.title,
        'link': entry.link,
        'published': entry.published,
        'summary': entry.summary
    } for entry in feed.entries]
```

2. Fonctionnement détaillé

La fonction `_decode_rss_stream` effectue plusieurs actions :

- **Décodage du flux RSS** :

  ```
  feed = feedparser.parse(response)
  ```

  La première étape de cette fonction consiste à analyser les données brutes du flux RSS en utilisant la bibliothèque `feedparser`. La méthode `parse` de `feedparser` prend en entrée les données brutes (souvent sous forme de bytes) et les convertit en un objet structuré qui peut être facilement manipulé, incluant un attribut `entries` qui contient la liste des éléments du flux.

- **Extraction des informations pertinentes** :

  ```
  return [{
      'title': entry.title,
      'link': entry.link,
      'published': entry.published,
      'summary': entry.summary
  } for entry in feed.entries]
  ```

  La fonction parcourt ensuite chaque élément du flux RSS (`feed.entries`) et extrait les informations pertinentes de chaque entrée :

  - `title` : Le titre de l'entrée RSS.
  - `link` : Le lien URL associé à l'entrée.
  - `published` : La date de publication de l'entrée.
  - `summary` : Un résumé ou une description de l'entrée.

  Ces informations sont organisées sous forme de dictionnaire pour chaque entrée, et une liste de ces dictionnaires est retournée.

3. Objectifs de la fonction `_decode_rss_stream`

La fonction `_decode_rss_stream` a pour objectif de décoder un flux RSS brut et d'extraire les informations essentielles, telles que le titre, le lien, la date de publication et le résumé de chaque entrée. Ces informations sont organisées de manière structurée (sous forme de dictionnaires), ce qui facilite leur traitement ultérieur par d'autres fonctions du programme. Cela permet de travailler avec des données propres et directement exploitables.

4. Exemple d'utilisation

Supposons que vous ayez reçu un flux RSS sous forme de bytes (généralement après une requête HTTP). Vous pouvez utiliser la fonction `_decode_rss_stream` pour extraire les informations pertinentes comme suit :

```
response = b'<?xml version="1.0" encoding="UTF-8" ?><rss>...</rss>'
entries = self._decode_rss_stream(response)
# L'extraction retournera une liste de dictionnaires contenant les informations des entr es.
```

Chaque élément de la liste `entries` contiendra un dictionnaire structuré, par exemple :

```
[{
    'title': 'Vulnerabilite critique (CVE-2025-01)',
    'link': 'https://www.example.com/alert/cve-2025-01',
    'published': 'Mon, 01 Jan 2025 10:30:00 +0000',
    'summary': 'Description de l\'alerte CVE-2025-01...'
}]
```

5. Pourquoi cette fonction est utile ?

La fonction `_decode_rss_stream` permet d'extraire rapidement et efficacement des données d'un flux RSS en convertissant les informations brut en un format structuré facile à manipuler. Elle permet d'ignorer les détails internes du flux RSS et de se concentrer sur les informations essentielles, ce qui simplifie l'analyse des données. De plus, elle est particulièrement utile pour des applications qui nécessitent de surveiller des flux RSS en temps réel et d'extraire des informations sur des événements ou alertes spécifiques.

La fonction `_fetch_remote_data`

La fonction `_fetch_remote_data` est utilisée pour effectuer une requête HTTP `GET` asynchrone à une URL donnée, afin de récupérer des données à distance. Elle s'appuie sur la bibliothèque `aiohttp` pour gérer les connexions HTTP de manière asynchrone et éviter les blocages.

1. Définition

```
async def _fetch_remote_data(self, url: str) -> Dict:
    try:
        async with self.session.get(url) as response:
            if response.status == 200:
                return await response.json()
            else:
                return {}
    except Exception as e:
        return {}
```

2. Fonctionnement détaillé

La fonction `_fetch_remote_data` exécute plusieurs étapes pour effectuer la requête HTTP et récupérer les données :

- **Requête HTTP `GET` asynchrone** :

  ```
      async with self.session.get(url) as response:
  ```

  La fonction utilise `aiohttp` pour envoyer une requête HTTP de type `GET` à l'URL fournie. L'instruction `async with` permet de gérer la requête de manière asynchrone, ce qui évite de bloquer l'exécution du programme pendant la récupération des données.

- **Vérification du code d'état de la réponse** :

  ```
      if response.status == 200:
          return await response.json()
      else:
          return {}
  ```

  Une fois la réponse reçue, la fonction vérifie le code d'état HTTP. Si le code est `200` (OK), la fonction tente de récupérer les données au format JSON avec `response.json()` et les retourne sous forme de dictionnaire Python. Si le code d'état n'est pas `200`, la fonction retourne un dictionnaire vide {} pour signaler l'échec de la requête.

- **Gestion des exceptions** :

  ```
      except Exception as e:
          return {}
  ```

  En cas d'erreur (par exemple, si la connexion échoue ou si un autre problème se produit), la fonction capture l'exception et retourne également un dictionnaire vide pour éviter que le programme ne plante.

3. Objectifs de la fonction `_fetch_remote_data`

L'objectif de cette fonction est de récupérer des données d'une URL distante en effectuant une requête HTTP `GET` de manière asynchrone. Elle permet de récupérer les données sous forme JSON et de gérer les erreurs de manière robuste, en retournant un dictionnaire vide en cas de problème.

4. Exemple d'utilisation

Supposons que vous ayez une URL et que vous souhaitiez récupérer des données JSON depuis cette URL. Vous pouvez appeler la fonction `_fetch_remote_data` comme suit :

```
url = "https://api.example.com/data"
data = await self._fetch_remote_data(url)
# Si la requ te r ussit , 'data' contiendra les donn es JSON sous forme de dictionnaire.
```

Si la requête réussit, la variable `data` contiendra les données JSON extraites de la réponse. Si la requête échoue (par exemple, en cas de mauvais code d'état HTTP ou d'exception), `data` sera un dictionnaire vide {}.

5. Pourquoi cette fonction est utile ?

La fonction `_fetch_remote_data` permet de récupérer des données à distance de manière asynchrone, ce qui améliore l'efficacité du programme en permettant d'effectuer d'autres tâches pendant que la requête HTTP est en cours. De plus, la gestion des erreurs garantit que l'application ne plante pas en cas de problème avec la requête HTTP. La fonction retourne toujours un dictionnaire, ce qui simplifie le traitement des données récupérées, que la requête réussisse ou échoue.

La fonction `process_cve_batch`

La fonction `process_cve_batch` est utilisée pour traiter un lot de CVEs (Common Vulnerabilities and Exposures). Elle prend en entrée un ensemble de données et effectue des opérations de récupération de données, de transformation et de traitement de manière efficace et asynchrone. Elle intègre plusieurs étapes pour manipuler les informations relatives aux vulnérabilités, et peut être utilisée dans le cadre de l'analyse de flux de sécurité.

1. Définition

```
1 async def process_cve_batch(self, cve_batch:
      List[str]) -> List[Dict]:
2     results = []
3     for cve in cve_batch:
4         cve_data = await self.fetch_cve_data(
              cve)
5         if cve_data:
6             processed_cve = await self.
                  process_single_cve(cve_data)
7             results.append(processed_cve)
8     return results
```

2. Fonctionnement détaillé

La fonction `process_cve_batch` s'exécute de manière asynchrone et suit les étapes suivantes pour traiter un lot de CVEs :

- **Initialisation de la liste des résultats** :

  ```
  1     results = []
  ```

  La fonction initialise une liste vide `results` qui stockera les données traitées de chaque CVE du lot.

- **Itération sur chaque CVE du lot** :

  ```
  1     for cve in cve_batch:
  2         cve_data = await self.
                fetch_cve_data(cve)
  ```

  La fonction parcourt chaque élément de `cve_batch`, qui est une liste de chaînes représentant les CVEs à traiter. Pour chaque CVE, elle appelle la fonction `fetch_cve_data` pour récupérer les données associées au CVE.

- **Traitement des données du CVE** :

  ```
  1     if cve_data:
  2         processed_cve = await self.
                process_single_cve(cve_data)
  3         results.append(processed_cve)
  ```

  Si les données du CVE (`cve_data`) sont disponibles (non vides), la fonction passe au traitement des données en appelant `process_single_cve`. Cette fonction permet de transformer ou analyser les informations du CVE pour les rendre prêtes à être utilisées. Les données traitées sont ensuite ajoutées à la liste `results`.

- **Retour des résultats** :

  ```
  1     return results
  ```

  Une fois que toutes les CVEs du lot ont été traitées, la fonction retourne la liste `results` contenant les informations traitées pour chaque CVE.

3. Objectifs de la fonction `process_cve_batch`

L'objectif de cette fonction est de traiter un lot de CVEs en plusieurs étapes : récupération des données, traitement des informations pour chaque CVE, et stockage des résultats dans une liste. Elle permet de traiter plusieurs CVEs en parallèle grâce à l'asynchronisme, ce qui améliore les performances et permet de traiter un grand nombre de CVEs rapidement.

4. Exemple d'utilisation

Supposons que vous ayez un lot de CVEs sous forme de liste et que vous souhaitiez les traiter pour en extraire des informations spécifiques. Vous pouvez appeler la fonction `process_cve_batch` comme suit :

```
1 cve_batch = ["CVE-2025-0001", "CVE-2025-0002",
      "CVE-2025-0003"]
2 processed_cves = await self.process_cve_batch(
      cve_batch)
3 # La variable 'processed_cves' contiendra une
      liste de CVEs trait es
```

Si tout se passe bien, `processed_cves` contiendra une liste de dictionnaires contenant les informations traitées pour chaque CVE. Sinon, des erreurs seront gérées dans le processus de traitement, par exemple si `fetch_cve_data` échoue.

5. Pourquoi cette fonction est utile ?

La fonction `process_cve_batch` est utile dans le contexte où plusieurs CVEs doivent être traitées de manière parallèle et efficace. Grâce à l'asynchronisme, elle permet de gérer plusieurs requêtes HTTP simultanées pour récupérer les données des CVEs, ce qui accélère le traitement global. Elle permet également de traiter des lots de CVEs et d'extraire les informations pertinentes sans bloquer l'exécution du programme.

La fonction `fetch_mitre_metadata`

La fonction `fetch_mitre_metadata` est une fonction asynchrone utilisée pour récupérer les métadonnées liées à une vulnérabilité depuis la base de données MITRE ATT&CK. Elle permet de récupérer les informations détaillées sur les techniques et les tactiques associées à une vulnérabilité CVE.

1. Définition

```
async def fetch_mitre_metadata(self, cve_id: str) -> Dict:
    url = f"https://api.mitre.org/attack/v1/cve/{cve_id}"
    try:
        async with self.session.get(url) as response:
            if response.status == 200:
                return await response.json()
            else:
                return {}
    except Exception as e:
        return {}
```

2. Fonctionnement détaillé

La fonction `fetch_mitre_metadata` fonctionne de la manière suivante :

- **Construction de l'URL de l'API MITRE** :

  ```
  url = f"https://api.mitre.org/attack/v1/cve/{cve_id}"
  ```

  L'URL de l'API MITRE ATT&CK est construite en insérant l'identifiant du CVE dans l'URL. Cela permet de cibler spécifiquement l'entrée de la base de données MITRE liée à ce CVE.

- **Requête HTTP asynchrone pour récupérer les données** :

  ```
  async with self.session.get(url) as response:
  ```

  Une requête HTTP `GET` est envoyée de manière asynchrone vers l'API MITRE à l'aide de `self.session.get(url)`. Cette requête récupère les métadonnées du CVE spécifié.

- **Vérification de la réponse** :

  ```
  if response.status == 200:
      return await response.json()
  ```

  Si le code de statut de la réponse est 200 (indiquant une réponse réussie), les données JSON sont extraites de la réponse et retournées sous forme de dictionnaire Python.

- **Gestion des erreurs** :

  ```
  except Exception as e:
      return {}
  ```

  En cas d'erreur (par exemple, un problème réseau ou une réponse invalide), une exception est levée et la fonction retourne un dictionnaire vide pour éviter que l'application ne plante.

3. Retour de la fonction

La fonction retourne un dictionnaire contenant les métadonnées du CVE si la requête réussit (code de statut HTTP 200). Si la requête échoue pour une raison quelconque, la fonction retourne un dictionnaire vide {}.

4. Exemple d'utilisation

Supposons que vous ayez un identifiant CVE et que vous souhaitiez récupérer ses métadonnées associées dans la base de données MITRE. Vous pouvez appeler la fonction `fetch_mitre_metadata` comme suit :

```
cve_id = "CVE-2025-0001"
mitre_metadata = await self.fetch_mitre_metadata(cve_id)
# La variable 'mitre_metadata' contiendra un dictionnaire avec les m tadonn es du CVE
```

Si la requête réussit, `mitre_metadata` contiendra un dictionnaire avec les informations détaillées du CVE. Si la requête échoue, la fonction retournera un dictionnaire vide.

5. Pourquoi cette fonction est utile ?

La fonction `fetch_mitre_metadata` est utile pour obtenir des informations détaillées sur une vulnérabilité CVE à partir de la base de données MITRE ATT&CK. Ces informations peuvent inclure les techniques et tactiques associées à la vulnérabilité, ce qui permet une analyse plus approfondie de la menace. De plus, l'asynchronisme permet de récupérer rapidement les données de manière parallèle, ce qui améliore les performances dans les environnements avec de nombreux CVEs à traiter.

La fonction `_process_mitre_block`

La fonction `_process_mitre_block` est utilisée pour traiter les données reçues de l'API MITRE et les formater de manière structurée. Elle extrait et organise les informations relatives aux tactiques et techniques MITRE associées à un CVE. Cela permet d'intégrer facilement ces informations dans un flux de travail ou une base de données.

1. Définition

```
def _process_mitre_block(self, mitre_data: Dict) -> List[Dict]:
    mitre_block = []
    for tactic in mitre_data.get("tactics", []):
        for technique in tactic.get("techniques", []):
            mitre_block.append({
                'tactic': tactic.get("name", ""),
                'technique': technique.get("name", ""),
                'id': technique.get("id", ""),
                'description': technique.get("description", ""),
                'url': technique.get("url", "")
```

```
            })
    return mitre_block
```

2. Fonctionnement détaillé

La fonction `_process_mitre_block` fonctionne de la manière suivante :

- **Initialisation de la liste `mitre_block`** :

  ```
      mitre_block = []
  ```

  La fonction commence par initialiser une liste vide `mitre_block`, qui contiendra les informations traitées relatives aux tactiques et techniques MITRE.

- **Parcours des tactiques MITRE** :

  ```
      for tactic in mitre_data.get("tactics", []):
  ```

  La fonction parcourt toutes les tactiques présentes dans les données MITRE reçues (les données sont accessibles via la clé `"tactics"`). Si aucune tactique n'est présente, une liste vide est retournée.

- **Parcours des techniques associées à chaque tactique** :

  ```
      for technique in tactic.get("techniques", []):
  ```

  Pour chaque tactique, la fonction parcourt les techniques associées à cette tactique (les données sont accessibles via la clé `"techniques"`). Encore une fois, si aucune technique n'est présente, une liste vide est retournée.

- **Création d'un dictionnaire pour chaque technique** :

  ```
      mitre_block.append({
          'tactic': tactic.get("name", ""),
          'technique': technique.get("name", ""),
          'id': technique.get("id", ""),
          'description': technique.get("description", ""),
          'url': technique.get("url", "")
      })
  ```

  Pour chaque technique, un dictionnaire est créé avec les informations suivantes :

  - `'tactic'` : Le nom de la tactique associée, extrait via `tactic.get("name", "")`.
  - `'technique'` : Le nom de la technique, extrait via `technique.get("name", "")`.
  - `'id'` : L'identifiant de la technique, extrait via `technique.get("id", "")`.
  - `'description'` : Une description de la technique, extraite via `technique.get("description", "")`.
  - `'url'` : Un lien vers plus de détails sur la technique, extrait via `technique.get("url", "")`.

  Ce dictionnaire est ensuite ajouté à la liste `mitre_block`.

- **Retour de la liste `mitre_block`** :

  ```
      return mitre_block
  ```

  Après avoir traité toutes les tactiques et techniques, la liste `mitre_block` est retournée. Cette liste contient un dictionnaire pour chaque technique, avec ses informations associées.

3. Retour de la fonction

La fonction retourne une liste de dictionnaires. Chaque dictionnaire contient les informations suivantes :

- `'tactic'` : Le nom de la tactique.
- `'technique'` : Le nom de la technique.
- `'id'` : L'identifiant de la technique.
- `'description'` : La description de la technique.
- `'url'` : L'URL associée à la technique.

Cela permet d'extraire et d'organiser facilement les données des techniques MITRE associées à une vulnérabilité.

4. Exemple d'utilisation

Supposons que vous ayez déjà récupéré les données MITRE associées à un CVE, et que vous souhaitiez les organiser sous forme de liste de dictionnaires. Vous pouvez appeler la fonction `_process_mitre_block` comme suit :

```
mitre_data = {
    "tactics": [
        {
            "name": "Initial Access",
            "techniques": [
                {
                    "name": "Phishing",
                    "id": "T1566",
                    "description": "Phishing is a technique...",
                    "url": "https://attack.mitre.org/techniques/T1566/"
                }
            ]
        }
    ]
}

mitre_block = self._process_mitre_block(mitre_data)
# La variable 'mitre_block' contiendra une liste de dictionnaires avec les informations sur les tactiques et techniques
```

Dans cet exemple, la variable `mitre_block` contiendra une liste de dictionnaires avec les informations suivantes :

```
[{
    'tactic': 'Initial Access',
    'technique': 'Phishing',
    'id': 'T1566',
```

```
    'description': 'Phishing is a technique...',
    'url': 'https://attack.mitre.org/techniques/T1566/'
}]
```

5. Pourquoi cette fonction est utile ?

La fonction `_process_mitre_block` est utile pour structurer les données MITRE ATT&CK d'une manière qui soit facilement manipulable et compréhensible. Elle permet de récupérer, organiser et formater les tactiques et techniques associées à un CVE de manière efficace. Les données structurées ainsi peuvent être utilisées pour des analyses plus approfondies, des rapports ou intégrées dans des systèmes de gestion des vulnérabilités.

La fonction `fetch_epss_scores`

La fonction `fetch_epss_scores` permet de récupérer les scores EPS (Exploit Prediction Scoring System) associés à une vulnérabilité CVE. Ces scores sont obtenus à partir d'une API externe et fournissent une indication de la probabilité qu'une vulnérabilité soit exploitée activement dans la nature.

1. Définition

```
async def fetch_epss_scores(self, cve_id: str) -> Dict:
    url = f"https://api.epss.io/v1/cve/{cve_id}"
    try:
        async with self.session.get(url) as response:
            if response.status == 200:
                return await response.json()
            else:
                return {}
    except:
        return {}
```

2. Fonctionnement détaillé

La fonction `fetch_epss_scores` fonctionne de la manière suivante :

- **Construction de l'URL** :

  ```
  url = f"https://api.epss.io/v1/cve/{cve_id}"
  ```

  L'URL de l'API EPSS est construite en insérant l'identifiant `cve_id` dans l'URL. Cela permet de récupérer les scores EPS pour un CVE spécifique.

- **Requête HTTP asynchrone** :

  ```
  async with self.session.get(url) as response:
  ```

  La fonction effectue une requête HTTP asynchrone à l'API EPSS en utilisant `self.session.get(url)`. Cela permet de récupérer les données de manière non-bloquante, optimisant ainsi la gestion des ressources.

- **Vérification du code d'état de la réponse** :

  ```
  if response.status == 200:
      return await response.json()
  else:
      return {}
  ```

  Si la réponse HTTP a un code d'état 200 (OK), la fonction récupère les données JSON retournées par l'API en utilisant `response.json()`. Si le code d'état n'est pas 200, la fonction retourne un dictionnaire vide {}.

- **Gestion des exceptions** :

  ```
  except:
      return {}
  ```

  En cas d'erreur (par exemple, une exception liée à un problème de connexion), la fonction capture l'exception et retourne un dictionnaire vide {}. Cela permet d'éviter que l'application ne plante en cas de problème de connexion ou d'autres erreurs.

3. Retour de la fonction

La fonction retourne un dictionnaire contenant les scores EPS associés à la vulnérabilité. Si la requête échoue ou si la réponse de l'API n'est pas valide, un dictionnaire vide {} est retourné.

4. Exemple d'utilisation

Voici un exemple de la façon dont vous pourriez appeler cette fonction pour récupérer les scores EPS pour un CVE spécifique :

```
cve_id = "CVE-2021-34527"
epss_scores = await self.fetch_epss_scores(cve_id)
# 'epss_scores' contiendra le dictionnaire avec les scores EPS ou un dictionnaire vide en cas d'erreur
```

Dans cet exemple, la variable `epss_scores` contiendra les données JSON retournées par l'API EPSS, sous forme de dictionnaire, ou un dictionnaire vide en cas de problème.

5. Pourquoi cette fonction est utile ?

La fonction `fetch_epss_scores` est utile pour récupérer rapidement les scores EPS d'une vulnérabilité spécifique, ce qui permet d'évaluer le risque d'exploitation de cette vulnérabilité. Ces scores peuvent être utilisés pour prioriser les actions de remédiation dans les processus de gestion des vulnérabilités. Elle est conçue pour être utilisée de manière asynchrone, ce qui permet de gérer efficacement les connexions réseau sans bloquer l'exécution de l'application.

La classe `MemCache`

La classe `MemCache` est une implémentation d'un cache en mémoire conçu pour stocker temporairement des données et ainsi améliorer les performances des requêtes

fréquentes. Elle permet de conserver des données fréquemment utilisées dans la mémoire afin d'éviter des appels répétés à des ressources externes, comme une API ou une base de données. En utilisant un cache en mémoire, les accès aux données deviennent plus rapides et réduisent la charge sur les ressources externes.

1. Objectif de la classe `MemCache`

Le but de la classe `MemCache` est de gérer efficacement un cache en mémoire pour stocker et récupérer des données temporaires. Elle permet ainsi de :

- Réduire la latence des appels en évitant des requêtes répétitives pour les mêmes données.
- Optimiser l'utilisation des ressources en limitant les accès redondants aux données.
- Fournir un mécanisme simple pour stocker et récupérer des informations dans un cache local.

2. Structure de la classe

La classe `MemCache` est généralement composée des éléments suivants :

- **Attributs de la classe** :
  - Un dictionnaire (ou une structure similaire) pour stocker les données mises en cache.
  - Un mécanisme pour définir une durée de vie des éléments du cache, afin de limiter leur stockage dans la mémoire.
- **Méthodes principales** :
  - `get(key)` : Récupère les données mises en cache associées à la clé `key`. Si les données ne sont pas présentes ou ont expiré, retourne une valeur par défaut ou déclenche une action pour actualiser les données.
  - `set(key, value)` : Ajoute ou met à jour les données dans le cache avec la clé `key` et la valeur associée `value`.
  - `expire(key)` : Permet de définir ou de forcer l'expiration des données mises en cache pour une clé donnée.
- **Gestion de la mémoire** :
  - Le cache peut avoir une capacité maximale pour éviter de surcharger la mémoire. Par exemple, en utilisant une politique de gestion de la mémoire comme LRU (Least Recently Used) pour supprimer les anciennes données lorsque le cache est plein.

3. Fonctionnement général

L'objectif principal de `MemCache` est d'améliorer les performances en offrant une solution rapide pour stocker et récupérer des données temporaires. Voici comment elle fonctionne en général :

- Lorsqu'une donnée est demandée, la méthode `get` vérifie d'abord si elle est présente dans le cache. Si c'est le cas, elle est retournée immédiatement, évitant ainsi un appel coûteux à une ressource externe.
- Si la donnée n'est pas présente ou si elle a expiré, la méthode `set` est utilisée pour récupérer les données et les stocker dans le cache pour une utilisation future. Cette opération peut inclure une mise à jour de la cache si les données changent.
- Une politique de gestion du cache, telle que l'expiration des données après un certain temps ou une capacité maximale du cache, peut être mise en place pour s'assurer que les ressources mémoire sont utilisées de manière optimale.

4. Avantages de la classe `MemCache`

La classe `MemCache` présente plusieurs avantages significatifs :

- **Amélioration des performances** : En stockant temporairement les résultats des requêtes fréquentes, les applications peuvent réduire le temps de réponse global en évitant de refaire les mêmes calculs ou appels.
- **Réduction de la charge sur les ressources externes** : Le cache permet de limiter les accès redondants aux API, bases de données ou autres ressources externes, ce qui améliore l'efficacité globale du système.
- **Flexibilité et personnalisation** : La classe peut être configurée pour s'adapter à différents cas d'utilisation, avec la possibilité de spécifier des durées d'expiration, des tailles de cache et des politiques de gestion spécifiques.

5. Utilisation de la classe `MemCache`

Voici un exemple d'utilisation basique de la classe `MemCache` :

```
# Cr ation d'un objet MemCache
cache = MemCache()

# Ajout de donn es au cache
cache.set("user_123", {"name": "Alice", "age":
    30})

# R cup ration de donn es du cache
user_data = cache.get("user_123")
print(user_data)  # Affiche : {'name': 'Alice
    ', 'age': 30}

# Expiration d'une entr e
cache.expire("user_123")
```

Dans cet exemple, un objet `cache` est créé, des données sont ajoutées avec la méthode `set`, puis récupérées avec `get`. L'expiration des données est gérée avec `expire`.

La méthode `__init__` de `MemCache`

La méthode `__init__` est un constructeur spécial en Python qui est appelé lors de la création d'une instance de la classe `MemCache`. Elle permet d'initialiser les attributs de l'objet et de configurer son état initial avant que l'objet ne soit utilisé.

1. Définition de la méthode `__init__`

La méthode `__init__` est définie comme suit :

```
class MemCache:
    def __init__(self, max_cache_size: int =
        100, expiration_time: int = 3600):
        self.cache = {}
        self.max_cache_size = max_cache_size
        self.expiration_time = expiration_time
```

2. Fonctionnement de la méthode `__init__`

La méthode `__init__` est utilisée pour initialiser un objet `MemCache`. Elle prend deux arguments facultatifs :

- `max_cache_size` : Définit la taille maximale du cache (par défaut 100). Cela limite le nombre d'éléments que le cache peut contenir.
- `expiration_time` : Définit la durée d'expiration des éléments du cache en secondes (par défaut 3600 secondes, soit 1 heure). Cela permet de configurer une politique d'expiration automatique des éléments après un certain délai.

L'initialisation crée les attributs suivants :

- `self.cache` : Un dictionnaire vide qui servira à stocker les données mises en cache.
- `self.max_cache_size` : La taille maximale du cache, configurée selon l'argument passé ou la valeur par défaut.
- `self.expiration_time` : Le temps d'expiration des éléments du cache, configuré selon l'argument passé ou la valeur par défaut.

3. Objectif de la méthode `__init__`

L'objectif principal de cette méthode est de préparer l'objet `MemCache` à être utilisé, en initialisant son cache et ses paramètres de gestion du cache. Plus spécifiquement :

- Elle crée un dictionnaire `self.cache` pour stocker les éléments du cache.
- Elle définit une taille maximale pour le cache via `self.max_cache_size`, ce qui garantit que le cache ne dépasse pas une certaine capacité.
- Elle configure un temps d'expiration pour chaque élément du cache avec `self.expiration_time`, ce qui permet de gérer la validité des données mises en cache.

4. Exemple d'utilisation de la méthode `__init__`

Lors de la création d'un objet `MemCache`, la méthode `__init__` est automatiquement appelée pour initialiser l'objet :

```
# Cr ation d'un objet MemCache avec des
    param tres personnalis s
cache = MemCache(max_cache_size=200,
    expiration_time=7200)

# Acc s aux attributs de l'objet
print(cache.max_cache_size)  # Affiche : 200
print(cache.expiration_time)  # Affiche : 7200
```

Dans cet exemple, un objet `MemCache` est créé avec une taille de cache maximale de 200 éléments et un temps d'expiration de 2 heures. Ces paramètres sont définis lors de l'initialisation de l'objet.

La méthode `_check_validity` de `MemCache`

La méthode `_check_validity` est utilisée pour vérifier si un élément stocké dans le cache est toujours valide en fonction de son temps d'expiration. Elle est essentielle pour assurer que les données stockées ne sont pas obsolètes et qu'elles peuvent être utilisées en toute sécurité.

1. Définition de la méthode `_check_validity`

La méthode `_check_validity` est définie comme suit :

```
def _check_validity(self, cache_key: str) ->
    bool:
    if cache_key not in self.cache:
        return False
    cache_item = self.cache[cache_key]
    if time.time() - cache_item['timestamp'] >
        self.expiration_time:
        return False
    return True
```

2. Fonctionnement de la méthode `_check_validity`

La méthode `_check_validity` prend en entrée un argument :

- `cache_key` : La clé associée à l'élément à vérifier dans le cache.

Elle effectue les étapes suivantes pour vérifier la validité de l'élément du cache :

1. **Vérification de l'existence de la clé dans le cache** :

```
    if cache_key not in self.cache:
        return False
```

La méthode vérifie d'abord si la clé `cache_key` est présente dans le cache (`self.cache`). Si la clé n'existe pas, cela signifie que l'élément n'a jamais été mis en cache ou a été supprimé, et donc la méthode retourne `False`, indiquant que l'élément est invalide.

2. **Vérification de l'expiration de l'élément** :

```
    cache_item = self.cache[cache_key]
    if time.time() - cache_item['timestamp
        '] > self.expiration_time:
        return False
```

Ensuite, la méthode récupère l'élément du cache associé à la clé et compare le temps actuel (`time.time()`) à l'horodatage (`timestamp`) de l'élément. Si le temps écoulé depuis l'horodatage est supérieur au temps d'expiration (`self.expiration_time`), cela signifie que l'élément a expiré, et la méthode retourne `False`.

3. **Retour de la validité** :

```
    return True
```

Si l'élément existe dans le cache et n'a pas expiré, la méthode retourne `True`, indiquant que l'élément est valide et peut être utilisé.

3. Objectif de la méthode `_check_validity`

L'objectif principal de cette méthode est de garantir que les éléments utilisés dans le cache sont toujours valides, c'est-à-dire qu'ils existent et n'ont pas expiré. Cela permet de maintenir la cohérence des données et d'éviter l'utilisation d'informations obsolètes.

4. Exemple d'utilisation de la méthode `_check_validity`

Voici un exemple d'utilisation de la méthode `_check_validity` dans le contexte d'un objet `MemCache` :

```
# Cr ation d'un objet MemCache
cache = MemCache(max_cache_size=100,
    expiration_time=3600)

# Ajout d'un   lment   au cache
cache.cache['item1'] = {'data': 'some data', '
    timestamp': time.time()}

# V rification de la validit  de l'  lment
    'item1'
is_valid = cache._check_validity('item1')
print(is_valid)  # Affiche : True si l'
    lment   est toujours valide

# Simulation d'expiration de l'  lment   en
    attendent plus longtemps
time.sleep(3601)  # Attente de plus d'une
    heure

# V rification de la validit  apr s
    expiration
is_valid = cache._check_validity('item1')
print(is_valid)  # Affiche : False si l'
    lment   a expir
```

Dans cet exemple, nous ajoutons un élément au cache et vérifions sa validité avant et après l'expiration, ce qui permet de voir la méthode en action.

La méthode `_update_cache` de `MemCache`

La méthode `_update_cache` est utilisée pour ajouter ou mettre à jour un élément dans le cache. Elle est essentielle pour garantir que les données en cache sont actualisées avec les dernières informations disponibles, tout en respectant les limites de taille du cache.

1. Définition de la méthode `_update_cache`

La méthode `_update_cache` est définie comme suit :

```
def _update_cache(self, cache_key: str, data:
    Any) -> None:
    if len(self.cache) >= self.max_cache_size:
        self._evict_cache_item()
    self.cache[cache_key] = {'data': data, '
        timestamp': time.time()}
```

2. Fonctionnement de la méthode `_update_cache`

La méthode `_update_cache` prend deux arguments :

- `cache_key` : La clé sous laquelle l'élément sera stocké ou mis à jour dans le cache.
- `data` : Les données à stocker ou mettre à jour dans le cache.

Elle effectue les étapes suivantes pour mettre à jour le cache :

1. **Vérification de la taille du cache** :

```
    if len(self.cache) >= self.
        max_cache_size:
        self._evict_cache_item()
```

La méthode commence par vérifier si la taille actuelle du cache a atteint la taille maximale définie (`self.max_cache_size`). Si le cache est plein, elle appelle la méthode `_evict_cache_item` pour évincer un élément et libérer de l'espace.

2. **Mise à jour du cache** :

```
    self.cache[cache_key] = {'data': data,
        'timestamp': time.time()}
```

Ensuite, la méthode met à jour le cache en ajoutant ou en modifiant l'élément associé à `cache_key`. Les données (`data`) sont stockées avec un horodatage (`timestamp`) qui correspond au moment où l'élément a été ajouté ou mis à jour. Cet horodatage permet de gérer l'expiration des éléments du cache et de s'assurer qu'ils sont utilisés dans un délai raisonnable.

3. Objectif de la méthode _update_cache

L'objectif principal de cette méthode est de maintenir les données du cache à jour tout en contrôlant sa taille. Cela permet d'assurer que le cache ne devienne pas trop grand et ne consomme pas trop de mémoire. De plus, en associant chaque élément à un horodatage, la méthode permet de gérer l'expiration et la validité des éléments du cache.

4. Exemple d'utilisation de la méthode _update_cache

Voici un exemple d'utilisation de la méthode `_update_cache` dans le contexte d'un objet `MemCache` :

```
# Cr ation d'un objet MemCache
cache = MemCache(max_cache_size=100,
    expiration_time=3600)

# Mise    jour du cache avec un nouvel
      lment
cache._update_cache('item1', {'data': 'some
    new data'})

# V rification du contenu du cache
print(cache.cache['item1'])  # Affiche : {'
    data': 'some new data', 'timestamp':
    1632765832.345}
```

Dans cet exemple, la méthode `_update_cache` est utilisée pour ajouter ou mettre à jour un élément dans le cache. Après l'exécution de cette méthode, l'élément est stocké avec un horodatage qui peut être utilisé pour vérifier sa validité dans d'autres parties du programme.

5. Pourquoi cette méthode est utile ?

- **Mise à jour du cache** : Cette méthode permet de maintenir les données en cache à jour, ce qui évite de travailler avec des données obsolètes.
- **Gestion de la taille du cache** : En contrôlant la taille du cache et en évitant d'atteindre la limite de mémoire (`max_cache_size`), cette méthode assure que le cache reste efficace sans surcharger la mémoire disponible.
- **Eviction des éléments obsolètes** : Si le cache est plein, des éléments obsolètes peuvent être évincés pour faire de la place à de nouvelles données, ce qui permet une gestion dynamique de la mémoire.

La méthode _get_cache de MemCache

La méthode `_get_cache` est utilisée pour récupérer un élément stocké dans le cache en fonction de sa clé. Elle permet de récupérer rapidement des données en mémoire, tout en vérifiant si ces données sont toujours valides selon leur date d'expiration.

1. Définition de la méthode _get_cache

La méthode `_get_cache` est définie comme suit :

```
def _get_cache(self, cache_key: str) -> Any:
    if cache_key in self.cache:
        item = self.cache[cache_key]
        if time.time() - item['timestamp'] <=
            self.expiration_time:
            return item['data']
    return None
```

2. Fonctionnement de la méthode _get_cache

La méthode `_get_cache` prend un argument :

- `cache_key` : La clé de l'élément à récupérer dans le cache.

Elle effectue les étapes suivantes pour récupérer l'élément depuis le cache :

1. **Vérification de la présence de l'élément dans le cache** :

   ```
       if cache_key in self.cache:
   ```

   La méthode commence par vérifier si la clé `cache_key` existe dans le cache. Si ce n'est pas le cas, elle passe à l'étape suivante et retourne `None`.

2. **Vérification de la validité de l'élément** :

   ```
       if time.time() - item['timestamp'] <=
           self.expiration_time:
           return item['data']
   ```

   Si l'élément est présent dans le cache, la méthode vérifie si l'élément est encore valide en comparant l'horodatage de l'élément avec le temps actuel. Si le temps écoulé depuis l'ajout de l'élément dans le cache est inférieur à `self.expiration_time`, cela signifie que l'élément est toujours valide et donc, les données (`item['data']`) sont retournées.

3. **Retour de `None` en cas d'élément invalide ou inexistant** :

   ```
       return None
   ```

   Si l'élément n'est pas trouvé ou si le temps d'expiration de l'élément a été dépassé, la méthode retourne `None`, ce qui indique que les données demandées ne sont pas disponibles ou sont obsolètes.

3. Objectif de la méthode _get_cache

L'objectif principal de la méthode `_get_cache` est de fournir un accès rapide aux données en cache tout en garantissant que ces données ne sont pas obsolètes. Elle permet de vérifier l'existence et la validité d'un élément dans le cache avant de le retourner, réduisant ainsi les appels à des ressources externes coûteuses (comme des bases de données ou des requêtes HTTP).

4. Exemple d'utilisation de la méthode _get_cache

Voici un exemple d'utilisation de la méthode _get_cache dans le contexte d'un objet MemCache :

```
# Cr ation d'un objet MemCache avec une taille de cache maximale et un temps d'expiration
cache = MemCache(max_cache_size=100, expiration_time=3600)

# Ajout d'un   lment   au cache
cache._update_cache('item1', {'data': 'some cached data'})

# R cup ration de l'  lment      partir du cache
cached_data = cache._get_cache('item1')

# Affichage du r sultat
if cached_data:
    print(cached_data)  # Affiche : some cached data
else:
    print("Data not found or expired")
```

Dans cet exemple, la méthode _get_cache est utilisée pour récupérer un élément du cache. Si l'élément est valide (c'est-à-dire, qu'il n'a pas expiré), ses données sont retournées. Sinon, None est retourné et un message est affiché pour indiquer que les données sont soit absentes, soit expirées.

5. Pourquoi cette méthode est utile ?

- **Accès rapide aux données en cache** : Cette méthode permet de récupérer les données directement depuis la mémoire, ce qui est beaucoup plus rapide que de faire une requête HTTP ou une recherche dans une base de données.
- **Gestion de l'expiration des données** : Elle s'assure que seules les données non expirées sont utilisées, ce qui garantit l'exactitude des informations et évite l'utilisation d'informations obsolètes.
- **Réduction des appels coûteux** : En utilisant un cache, cette méthode permet de réduire les appels externes, ce qui améliore la performance globale du programme.

La méthode get_or_fetch de MemCache

La méthode get_or_fetch combine la récupération d'un élément depuis le cache avec la possibilité de récupérer l'élément depuis une source externe si celui-ci n'est pas présent dans le cache ou s'il a expiré. Cela permet d'optimiser les performances en utilisant d'abord les données en cache, et en ne faisant appel à des ressources externes qu'en cas de besoin.

1. Définition de la méthode get_or_fetch

La méthode get_or_fetch est définie comme suit :

```
def get_or_fetch(self, cache_key: str, fetch_func: Callable[[], Any]) -> Any:
    cached_data = self._get_cache(cache_key)
    if cached_data is not None:
        return cached_data
    data = fetch_func()
    self._update_cache(cache_key, data)
    return data
```

2. Fonctionnement de la méthode get_or_fetch

La méthode get_or_fetch prend deux arguments :

- cache_key : La clé de l'élément à récupérer ou à charger dans le cache.
- fetch_func : Une fonction qui sera appelée pour récupérer les données si elles ne sont pas présentes dans le cache.

Elle effectue les étapes suivantes :

1. **Récupération des données depuis le cache** :

   ```
   cached_data = self._get_cache(cache_key)
   ```

   La méthode commence par essayer de récupérer l'élément depuis le cache en appelant la méthode _get_cache avec la clé spécifiée. Si l'élément est trouvé et est valide, il est retourné directement.

2. **Récupération des données depuis une source externe** :

   ```
   if cached_data is not None:
       return cached_data
   data = fetch_func()
   ```

   Si l'élément n'est pas trouvé dans le cache (ou s'il a expiré), la méthode appelle la fonction fetch_func, qui est passée en argument à la méthode. Cette fonction est censée récupérer les données depuis une source externe, par exemple, une API ou une base de données.

3. **Mise à jour du cache** :

   ```
   self._update_cache(cache_key, data)
   ```

   Une fois les données récupérées depuis la source externe, elles sont stockées dans le cache en appelant la méthode _update_cache, associée à la clé cache_key.

4. **Retour des données** :

   ```
   return data
   ```

   Enfin, les données récupérées (que ce soit depuis le cache ou la source externe) sont retournées à l'appelant.

3. Objectif de la méthode `get_or_fetch`

L'objectif principal de la méthode `get_or_fetch` est d'optimiser l'accès aux données en cache tout en permettant de récupérer les données depuis une source externe si elles ne sont pas présentes ou ont expiré dans le cache. Cela permet de réduire le nombre de requêtes vers des ressources externes et d'améliorer la performance globale de l'application.

4. Exemple d'utilisation de la méthode `get_or_fetch`

Voici un exemple d'utilisation de la méthode `get_or_fetch` dans le contexte d'un objet `MemCache` :

```
# Cr ation d'un objet MemCache avec une taille de cache maximale et un temps d' expiration
cache = MemCache(max_cache_size=100, expiration_time=3600)

# Fonction de r cup ration de donn es externes
def fetch_data_from_api():
    # Simule une requ te HTTP ou une recherche dans une base de donn es
    return {'key': 'value'}

# Utilisation de get_or_fetch pour obtenir les donn es
data = cache.get_or_fetch('api_data', fetch_data_from_api)

# Affichage des donn es r cup r es
print(data)
```

Dans cet exemple : - Si les données associées à la clé `'api_data'` sont présentes et valides dans le cache, elles seront retournées directement. - Si les données ne sont pas présentes dans le cache, la fonction `fetch_data_from_api` est appelée pour récupérer les données depuis une source externe (par exemple, une API). Ensuite, ces données sont stockées dans le cache et retournées.

5. Pourquoi cette méthode est utile ?

- **Optimisation des performances** : La méthode réduit les appels à des sources externes en vérifiant d'abord si les données sont disponibles dans le cache.
- **Réduction de la charge des ressources externes** : En utilisant le cache, la méthode permet de réduire le nombre de requêtes vers des API ou des bases de données, ce qui diminue la charge sur ces ressources.
- **Simplicité d'utilisation** : La méthode fournit une interface simple pour récupérer des données, qu'elles proviennent du cache ou d'une source externe, sans que l'utilisateur ait à gérer manuellement le cache ou les appels externes.

La ligne de code `_SYS_CACHE = MemCache()`

La ligne `_SYS_CACHE = MemCache()` crée une instance de la classe `MemCache`, qui est ensuite stockée dans la variable `_SYS_CACHE`. Cette instance représente un cache en mémoire utilisé pour stocker temporairement des données fréquemment demandées, afin d'optimiser les performances du système en réduisant les appels aux ressources externes (comme une base de données ou une API).

1. Explication de la ligne de code

```
_SYS_CACHE = MemCache()
```

Cette ligne effectue les actions suivantes :

1. **Appel au constructeur de la classe `MemCache` :**

   ```
   MemCache()
   ```

   - La classe `MemCache` est une classe qui gère un cache en mémoire. Son constructeur (`__init__`) initialise les propriétés du cache, comme la taille maximale du cache, le temps d'expiration des éléments stockés, et d'autres paramètres relatifs à sa gestion. - L'instanciation de la classe crée un objet qui gère la mémoire du cache.

2. **Stockage de l'objet `MemCache` dans la variable `_SYS_CACHE` :**

   ```
   _SYS_CACHE
   ```

   - La variable `_SYS_CACHE` contient maintenant une référence à l'objet de type `MemCache`. - Cette variable peut être utilisée dans tout le programme pour accéder à l'instance de `MemCache` et effectuer des opérations sur le cache, comme ajouter des éléments, les récupérer ou les supprimer.

2. Pourquoi cette ligne est importante ?

Cette ligne de code est cruciale pour le bon fonctionnement du cache au sein du système. Elle :

- Initialise un cache en mémoire qui peut être utilisé tout au long de l'exécution du programme.
- Permet d'accéder facilement au cache via la variable `_SYS_CACHE`, offrant une gestion centralisée des données mises en cache.
- Permet d'améliorer les performances du système en réduisant les requêtes redondantes aux ressources externes.

3. Utilisation de `_SYS_CACHE` dans le programme

Une fois l'instance `_SYS_CACHE` créée, elle peut être utilisée pour interagir avec le cache à travers les méthodes de la classe `MemCache`. Par exemple, pour récupérer des données mises en cache, on pourrait utiliser une méthode comme `get_or_fetch` de la manière suivante :

```
# Exemple d'utilisation
cached_data = _SYS_CACHE.get_or_fetch('
    some_cache_key', fetch_data_function)
```

Cela permet de gérer efficacement les données mises en cache et d'optimiser les performances du programme en limitant les accès aux ressources externes.

4. Pourquoi utiliser un cache en mémoire ?

L'utilisation d'un cache en mémoire permet d'optimiser les performances de l'application :

- **Réduction des temps de réponse** : En stockant temporairement des données fréquemment utilisées, le cache permet d'éviter de multiples appels aux ressources externes, ce qui réduit le temps de réponse.
- **Réduction de la charge sur les serveurs** : En utilisant les données en cache, on diminue la charge sur les serveurs externes ou les bases de données, ce qui peut améliorer la scalabilité du système.
- **Meilleure gestion des ressources** : Le cache peut être configuré avec une taille maximale et une expiration des éléments, ce qui permet de libérer de la mémoire lorsqu'elle est nécessaire.

La fonction _check_build_status

La fonction `_check_build_status` est utilisée pour vérifier le statut de construction (ou d'exécution) d'une tâche ou d'un processus dans le système. Cette vérification est généralement effectuée pour s'assurer qu'une certaine tâche a bien été réalisée avant de continuer avec d'autres processus. Elle peut être utilisée dans des systèmes asynchrones ou des processus qui nécessitent une gestion de l'état d'exécution.

1. Définition de la fonction

```
def _check_build_status(self, build_id: str)
    -> bool:
    """
    Check the build status based on the
        build_id.
    Returns True if the build is complete,
        False otherwise.
    """
    build_status = self._get_build_status(
        build_id)
    return build_status == 'completed'
```

2. Fonctionnement détaillé

a. Récupération du statut de construction

```
build_status = self._get_build_status(build_id
    )
```

- La fonction appelle une méthode `_get_build_status` qui, en fonction de l'ID de construction (représenté par `build_id`), récupère l'état actuel de la construction. - `self._get_build_status(build_id)` renvoie probablement un statut comme `'in progress'`, `'completed'`, ou `'failed'` pour indiquer l'état de la construction.

b. Vérification du statut de construction

```
return build_status == 'completed'
```

- La fonction compare le statut de la construction récupéré à la valeur `'completed'`. - Si le statut de la construction est `'completed'`, la fonction retourne `True`, indiquant que la construction est terminée avec succès. - Si le statut n'est pas `'completed'`, elle retourne `False`, indiquant que la construction n'est pas encore terminée.

3. Pourquoi cette fonction est utile ?

La fonction `_check_build_status` permet de surveiller l'état d'une construction ou d'un processus asynchrone. Elle est essentielle dans les systèmes où le processus de construction (compilation, traitement de données, etc.) prend du temps et où il est nécessaire de savoir si l'opération a été effectuée correctement avant de passer à l'étape suivante.

- **Vérification simple et rapide :** Elle fournit un moyen simple et rapide de vérifier si une tâche longue a été terminée.
- **Gestion du flux de travail :** Cette fonction peut être utilisée pour contrôler le flux de travail dans des systèmes complexes, où l'exécution de certaines étapes dépend de la réussite de tâches précédentes.
- **Gestion des erreurs :** Elle permet d'identifier les tâches qui ne sont pas terminées et d'agir en conséquence (par exemple, réessayer ou signaler une erreur).

4. Exemple d'utilisation

Imaginons que nous ayons un système qui compile un projet et que nous devions vérifier si la compilation est terminée avant d'exécuter d'autres étapes. On pourrait utiliser la fonction `_check_build_status` comme suit :

```
# Exemple d'utilisation
build_id = "12345"
if _check_build_status(build_id):
    print("La construction est termin e, vous
        pouvez proc der    l' tape  suivante
        .")
else:
    print("La construction n'est pas encore
        termin e.")
```

Cela permet de savoir si une construction spécifique est terminée avant d'exécuter d'autres opérations qui dépendent de son résultat.

La fonction _execute_build_process

La fonction `_execute_build_process` est responsable de l'exécution d'un processus de construction (généralement lié à un traitement ou une tâche de calcul complexe) dans le système. Cette fonction gère l'exécution complète du processus, en s'assurant que toutes les étapes nécessaires

sont exécutées dans le bon ordre, tout en gérant les éventuelles erreurs qui peuvent survenir pendant l'exécution.

1. Définition de la fonction

```python
def _execute_build_process(self, build_id: str) -> bool:
    """
    Execute the build process for the given build_id.
    Returns True if the build was successful, False otherwise.
    """
    try:
        self._start_build(build_id)
        self._monitor_build(build_id)
        return True
    except Exception as e:
        print(f"Error during build process: {e}")
        return False
```

2. Fonctionnement détaillé

a. Démarrage du processus de construction

```python
self._start_build(build_id)
```

- La fonction commence par appeler une méthode `_start_build` pour démarrer le processus de construction. - `build_id` est un identifiant unique qui permet de suivre le processus de construction spécifique à exécuter. - `_start_build(build_id)` initie le processus qui peut inclure des tâches comme la configuration d'un environnement, le lancement de la compilation, ou d'autres préparations.

b. Surveillance du processus de construction

```python
self._monitor_build(build_id)
```

- Une fois la construction lancée, la fonction appelle `_monitor_build` pour suivre l'avancement du processus. - Cette méthode peut être responsable de la vérification régulière de l'état de la construction, en s'assurant qu'elle progresse correctement sans erreurs.

c. Gestion des erreurs

```python
except Exception as e:
    print(f"Error during build process: {e}")
    return False
```

- Si une exception est levée à n'importe quel moment du processus, elle est capturée par le bloc `except`. - Un message d'erreur est affiché, puis la fonction retourne `False`, signalant que le processus de construction a échoué.

3. Retour de la fonction

- **Succès :** Si le processus de construction se déroule sans erreur, la fonction retourne `True`, indiquant que la construction a été effectuée avec succès.
- **Échec :** Si une erreur est rencontrée pendant le processus (lors du démarrage ou de la surveillance), la fonction retourne `False`.

4. Pourquoi cette fonction est utile ?

La fonction `_execute_build_process` est cruciale dans des systèmes où un certain processus doit être effectué avant de pouvoir passer à l'étape suivante. Elle garantit qu'un processus complexe (comme une compilation ou un traitement de données) est correctement démarré et surveillé. De plus, elle gère les exceptions, ce qui permet d'éviter des plantages du programme en cas d'erreurs imprévues.

- **Exécution complète du processus :** Elle prend en charge toutes les étapes nécessaires du processus de construction, de l'initialisation à la surveillance.
- **Gestion des erreurs :** Si une erreur se produit, elle est capturée et un message d'erreur est affiché, empêchant ainsi des comportements indésirables ou des plantages.
- **Contrôle du succès ou de l'échec :** Elle permet de savoir rapidement si le processus a été effectué correctement ou non.

5. Exemple d'utilisation

Un exemple d'utilisation pourrait être un système automatisé où chaque tâche doit être construite avant de passer à l'étape suivante :

```python
# Exemple d'utilisation
build_id = "build123"
if _execute_build_process(build_id):
    print("La construction a r ussi , vous pouvez continuer.")
else:
    print("La construction a   chou  , veuillez v rifier l'erreur.")
```

Dans cet exemple, le processus de construction pour l'ID donné commence, est surveillé, et le résultat est vérifié. Si la construction échoue, le système peut afficher un message d'erreur pour aider l'utilisateur à diagnostiquer le problème.

La ligne _APP = Flask(__name__, static_folder='static')

Cette ligne de code initialise une instance de l'application Flask, un framework web en Python utilisé pour développer des applications web. Elle configure certains paramètres de l'application, notamment son nom et le répertoire contenant les fichiers statiques.

1. Définition de la ligne

```python
_APP = Flask(__name__, static_folder='static')
```

2. Fonctionnement détaillé

a. `Flask` et son initialisation

```python
_APP = Flask(__name__)
```

- `Flask` est une classe qui représente l'application web. En créant une instance de cette classe, on initialise une nouvelle application Flask. - `__name__` est une variable spéciale dans Python qui contient le nom du module actuel. Elle est utilisée par Flask pour déterminer si le script est exécuté directement ou s'il est importé en tant que module. - `__name__` permet à Flask de savoir où chercher les fichiers associés à l'application (templates, fichiers statiques, etc.). - L'argument `__name__` est donc essentiel pour le bon fonctionnement de l'application Flask, en particulier pour la gestion des ressources.

b. Configuration du répertoire `static_folder`

```
static_folder='static'
```

- L'argument `static_folder` indique à Flask où se trouvent les fichiers statiques (par exemple, les images, les fichiers CSS et JavaScript) qui seront servis par l'application web. - Ici, `'static'` spécifie que le dossier `static` situé dans le répertoire principal du projet contiendra ces fichiers. - Flask sert ces fichiers de manière statique, c'est-à-dire qu'il les rend accessibles aux utilisateurs sans avoir à effectuer de traitement côté serveur à chaque demande.

3. Pourquoi cette ligne est utile ?

La ligne de code `_APP = Flask(__name__, static_folder='static')` est fondamentale pour démarrer l'application web Flask et lui permettre de gérer les fichiers statiques nécessaires au bon fonctionnement de l'interface utilisateur.

- **Initialisation de l'application web :** Cette ligne crée une instance de l'application Flask, ce qui permet d'ajouter des routes, des vues et d'autres configurations à l'application web.
- **Gestion des fichiers statiques :** En spécifiant le répertoire `static`, elle permet à Flask de rendre facilement accessibles les ressources statiques comme les images, les fichiers CSS et JavaScript à partir de l'URL.
- **Indépendance du module principal :** L'utilisation de `__name__` permet à l'application de s'exécuter correctement dans différents contextes (par exemple, en tant que module ou script principal).

4. Exemple d'utilisation

Voici un exemple minimaliste d'utilisation de cette ligne dans une application Flask :

```
from flask import Flask

# Initialisation de l'application Flask
_APP = Flask(__name__, static_folder='static')

@_APP.route('/')
def hello():
    return "Bonjour, monde!"

if __name__ == "__main__":
    _APP.run(debug=True)
```

Dans cet exemple, l'application Flask est initialisée, et une route est définie pour afficher "Bonjour, monde!" lorsque la page d'accueil est visitée. Le répertoire `static` est configuré pour servir les fichiers statiques de l'application.

Les lignes _APP.before_request et _init_system

Ces lignes de code sont utilisées dans une application Flask pour effectuer des actions avant chaque requête et pour initialiser le système respectivement. Elles permettent de mettre en place des processus et des vérifications nécessaires avant que chaque requête ne soit traitée par les vues de l'application. La première ligne est un décorateur Flask pour un traitement global avant chaque requête HTTP, tandis que la deuxième ligne est une fonction qui initialise certains composants de l'application.

1. Décorateur _APP.before_request

```
@_APP.before_request
def before_request():
    # Code    ex cuter avant chaque requ te
    pass
```

a. Explication du décorateur _APP.before_request

Le décorateur `@_APP.before_request` est utilisé pour enregistrer une fonction qui sera exécutée avant chaque requête HTTP traitée par l'application Flask. Cette fonction permet d'ajouter un comportement global à toutes les requêtes, comme la gestion des sessions, la vérification des tokens d'authentification, ou la préparation de l'environnement avant le traitement de la requête.

- **Enregistrement de la fonction** : La fonction décorée par `@_APP.before_request` sera appelée avant que Flask ne traite toute requête.
- **Cas d'usage typique** : Ce décorateur est utilisé pour des actions qui doivent être effectuées à chaque fois avant l'exécution d'une route, comme la validation des autorisations d'accès ou la gestion des ressources.
- **Comportement global** : Il est possible de définir plusieurs fonctions `before_request` qui seront exécutées dans l'ordre de leur enregistrement.

b. Exemple d'utilisation

Voici un exemple où on utilise `@_APP.before_request` pour vérifier si un utilisateur est authentifié avant de permettre l'accès à une route protégée.

```
@_APP.before_request
def before_request():
    if not session.get('logged_in'):
        return "Unauthorized", 401
```

Dans cet exemple, avant chaque requête, la fonction vérifie si l'utilisateur est connecté en vérifiant la session. Si l'utilisateur n'est pas authentifié, la requête retourne un message d'erreur avec le code HTTP 401 (non autorisé).

2. Fonction _init_system

```
def _init_system():
    # Code pour initialiser le syst me
    pass
```

a. Explication de la fonction _init_system

La fonction `_init_system` est utilisée pour initialiser certaines parties du système ou de l'application au démarrage. Cela peut inclure des actions telles que la configuration des composants, l'établissement des connexions aux bases de données, ou la mise en place de variables ou de ressources nécessaires pour que l'application fonctionne correctement.

- **Initialisation du système** : Cette fonction peut être appelée lors du démarrage de l'application pour effectuer des tâches de configuration qui doivent se produire avant le traitement des requêtes.
- **Utilisation typique** : Il est courant d'utiliser cette fonction pour initialiser des éléments comme les caches, les bases de données, ou d'autres dépendances externes.
- **Centralisation de l'initialisation** : En centralisant cette logique dans une fonction dédiée, on s'assure que l'initialisation du système est effectuée de manière cohérente et modulaire.

b. Exemple d'utilisation

Voici un exemple de la fonction `_init_system` utilisée pour initialiser un cache au démarrage de l'application.

```
def _init_system():
    _SYS_CACHE.initialize()
    print("System initialized.")
```

Dans cet exemple, la fonction `_init_system` appelle la méthode `initialize()` sur un objet cache (`_SYS_CACHE`), ce qui peut être nécessaire avant que le système n'accepte des requêtes.

3. Pourquoi ces lignes sont utiles ?

- **Avant chaque requête (before_request)** : Cette ligne garantit que des actions spécifiques (comme la validation des autorisations) sont exécutées avant chaque requête, ce qui permet de centraliser cette logique et d'éviter la répétition dans chaque route.
- **Initialisation du système** : La fonction `_init_system` centralise l'initialisation du système et garantit que tous les composants nécessaires (comme les caches ou les connexions) sont prêts avant de traiter les requêtes.
- **Contrôle du flux global** : Ces deux lignes assurent un contrôle centralisé sur le comportement de l'application avant qu'elle ne commence à traiter les demandes des utilisateurs.

Les lignes _APP.route('/') et _serve_index

Ces lignes de code sont utilisées dans une application Flask pour définir une route HTTP (endpoint) qui répond aux requêtes effectuées sur la racine de l'application. Le décorateur `@_APP.route('/')` est utilisé pour lier une fonction à une URL spécifique, ici la racine de l'application ("/"). La fonction _serve_index est ensuite définie pour répondre à cette requête en renvoyant une ressource (généralement une page HTML ou un fichier statique).

1. Décorateur _APP.route('/')

```
@_APP.route('/')
def _serve_index():
    # Code pour servir la page d'accueil
    pass
```

a. Explication du décorateur _APP.route('/')

Le décorateur `@_APP.route('/')` est utilisé dans Flask pour lier une fonction à une URL spécifique. Dans ce cas, il associe la fonction _serve_index à la route `'/'` (la racine de l'application), ce qui signifie que la fonction sera exécutée lorsque l'utilisateur accédera à cette URL.

- **Définition de la route** : `@_APP.route('/')` indique que la fonction qui suit sera appelée lorsqu'une requête HTTP (comme GET ou POST) est effectuée sur la racine de l'application.
- **URL cible** : La chaîne `'/'` représente la route de base de l'application, c'est-à-dire la page d'accueil. Elle peut être modifiée pour désigner d'autres chemins comme `'/about'` ou `'/contact'`.
- **Routage Flask** : Ce décorateur est essentiel pour définir le comportement de l'application sur une URL spécifique. Une application Flask peut définir de multiples routes pour gérer différentes URLs et actions.

b. Exemple d'utilisation

Voici un exemple de route définie avec `@_APP.route('/')` pour afficher une page d'accueil.

```
@_APP.route('/')
def _serve_index():
    return render_template('index.html')
```

Dans cet exemple, lorsque l'utilisateur accède à la racine du site, la fonction _serve_index est exécutée et renvoie la page HTML `index.html`.

2. Fonction _serve_index

```
def _serve_index():
    # Code pour servir la page d'accueil
    pass
```

a. Explication de la fonction _serve_index

La fonction _serve_index est une fonction Python associée à la route racine de l'application. Elle est responsable de l'exécution du traitement nécessaire pour répondre à la requête de l'utilisateur. Typiquement, elle peut rendre une page HTML ou un autre type de ressource.

- **Rendu de la page** : Une fonction associée à une route sert généralement à générer la réponse à envoyer à l'utilisateur. Par exemple, dans Flask, cette fonction peut utiliser `render_template` pour renvoyer une page HTML.
- **Comportement dynamique** : Selon la logique de l'application, cette fonction peut également récupérer des données dynamiques, effectuer des traitements, ou interagir avec une base de données avant de renvoyer la réponse.
- **Réponse aux requêtes** : Chaque route définie dans Flask a une fonction associée qui détermine comment la réponse est générée et envoyée. Ici, _serve_index répond à la requête de la page d'accueil.

b. Exemple d'utilisation

Voici un exemple de la fonction _serve_index renvoyant une page HTML avec un message d'accueil.

```
def _serve_index():
    return "Welcome to the homepage!"
```

Dans cet exemple, la fonction renvoie simplement une chaîne de caractères. Cependant, dans des applications réelles, elle renverrait généralement un fichier HTML ou une page rendue avec un moteur de templates comme Jinja2.

3. Pourquoi ces lignes sont utiles ?

- **Définition des routes** : `@_APP.route('/')` est crucial pour définir la structure de l'application, en associant une URL à une fonction qui exécute un traitement précis.
- **Gestion des requêtes** : La fonction _serve_index permet de traiter la demande de l'utilisateur et de renvoyer la réponse appropriée, que ce soit une page HTML, un fichier, ou un message dynamique.
- **Démarrage de l'application** : La route `'/'` est souvent utilisée pour servir la page d'accueil d'une application web, et donc cette fonction est l'un des points de départ principaux pour les utilisateurs interagissant avec l'application.
- **Flexibilité de Flask** : Le décorateur `@_APP.route('/')` et la fonction associée sont essentiels pour concevoir des applications web flexibles et modulaires où chaque URL correspond à une action spécifique.

Les lignes _APP.route('/fetch_data') et _handle_data_request

Ces lignes de code sont utilisées dans une application Flask pour définir une nouvelle route HTTP (endpoint) qui répond aux requêtes envoyées à l'URL `'/fetch_data'`. Le décorateur `@_APP.route('/fetch_data')` est utilisé pour lier la fonction _handle_data_request à cette route. Cette fonction sera exécutée lorsque l'utilisateur accède à cette URL, et elle sera responsable de la gestion des requêtes relatives à la récupération des données.

1. Décorateur _APP.route('/fetch_data')

```
@_APP.route('/fetch_data')
def _handle_data_request():
    # Code pour g rer la requ te de
        r cup ration des donn es
    pass
```

a. Explication du décorateur _APP.route('/fetch_data')

Le décorateur `@_APP.route('/fetch_data')` est utilisé pour lier une fonction à une URL spécifique. Dans ce cas, il associe la fonction _handle_data_request à la route `'/fetch_data'`, ce qui signifie que la fonction sera exécutée lorsque l'utilisateur accédera à cette URL via une requête HTTP (généralement une requête GET).

- **Définition de la route** : `@_APP.route('/fetch_data')` indique que la fonction suivante (_handle_data_request) sera exécutée lorsque l'utilisateur accède à l'URL `'/fetch_data'` de l'application.
- **URL cible** : La chaîne `'/fetch_data'` désigne une URL personnalisée dans l'application. Les utilisateurs peuvent accéder à cette route pour demander des données depuis l'application.
- **Routage Flask** : Ce décorateur permet à Flask de gérer les requêtes envoyées à cette URL et de rediriger le traitement vers la fonction associée.

b. Exemple d'utilisation

Voici un exemple où l'on pourrait utiliser cette route pour récupérer des données sous forme de JSON.

```
@_APP.route('/fetch_data')
def _handle_data_request():
    return jsonify({"data": "Here is some data
        !"})
```

Dans cet exemple, la fonction _handle_data_request renvoie des données sous forme de réponse JSON lorsqu'une requête est envoyée à la route `'/fetch_data'`.

2. Fonction _handle_data_request

```
def _handle_data_request():
    # Code pour g rer la requ te de
        r cup ration des donn es
    pass
```

a. Explication de la fonction _handle_data_request

La fonction _handle_data_request est une fonction associée à la route `'/fetch_data'`. Elle est responsable de traiter les requêtes envoyées à cette URL et de fournir une réponse appropriée, généralement sous la forme de données.

- **Récupération des données** : La fonction peut interagir avec des bases de données, des fichiers ou des services externes pour récupérer les données demandées par l'utilisateur.
- **Réponse aux requêtes** : Elle renvoie ensuite les données sous un format approprié, comme JSON, XML, ou même HTML, selon les besoins de l'application.
- **Traitement personnalisé** : Cette fonction peut inclure des vérifications de paramètres de requête, des filtres ou des logiques de traitement avant de renvoyer la réponse.

b. Exemple d'utilisation

Voici un exemple de la fonction _handle_data_request qui interagit avec une base de données pour renvoyer des informations.

```
def _handle_data_request():
    # Exemple de r cup ration de donn es
        depuis une base de donn es
    data = database.query("SELECT * FROM data"
        )
    return jsonify(data)
```

Dans cet exemple, la fonction récupère des données depuis une base de données et les renvoie sous forme de réponse JSON.

3. Pourquoi ces lignes sont utiles ?

- **Définition des routes** : Le décorateur `@_APP.route('/fetch_data')` définit la logique de routage de l'application, en associant une URL à une fonction qui répondra aux requêtes de l'utilisateur.
- **Gestion des requêtes** : La fonction _handle_data_request permet de gérer les requêtes effectuées sur l'URL `'/fetch_data'`, en fournissant des données ou des informations à l'utilisateur.
- **Réponse dynamique** : Cette approche permet de récupérer des données dynamiquement à partir de différentes sources (API, base de données, etc.), et de les renvoyer sous un format structuré comme JSON.
- **Flexibilité de Flask** : Le décorateur `@_APP.route('/fetch_data')` et la fonction associée offrent une méthode flexible et extensible pour ajouter des endpoints supplémentaires à l'application web.

La fonction _fetch_all_data

La fonction _fetch_all_data est responsable de la récupération de toutes les données nécessaires pour l'application. Elle interagit généralement avec des services externes ou des bases de données pour obtenir l'ensemble des données disponibles, puis les retourne sous un format utilisable pour l'application.

Code de la fonction _fetch_all_data

```
def _fetch_all_data():
    # R cup rer toutes les donn es
        n cessaires depuis des sources
        externes
    cve_data = _fetch_cve_data()
    mitre_data = fetch_mitre_metadata()
    epss_scores = fetch_epss_scores()

    # Combiner toutes les donn es
    all_data = {
        "cve_data": cve_data,
        "mitre_data": mitre_data,
        "epss_scores": epss_scores
    }

    return all_data
```

Explication de la fonction _fetch_all_data

La fonction _fetch_all_data permet de centraliser la récupération de toutes les données nécessaires à l'application. Elle regroupe les résultats de plusieurs appels de fonction, qui sont ensuite combinés dans un dictionnaire unique. Ce mécanisme simplifie la gestion des données et la rend facilement accessible pour l'application.

1. Récupération des données

La fonction commence par appeler plusieurs autres fonctions pour récupérer les données provenant de différentes sources externes.

- **Récupération des CVE** : `cve_data = _fetch_cve_data()` appelle une fonction (probablement _fetch_cve_data) pour obtenir les informations relatives aux CVE (Common Vulnerabilities and Exposures).
- **Récupération des métadonnées MITRE** : `mitre_data = fetch_mitre_metadata()` appelle une autre fonction pour obtenir des informations provenant de la base de données MITRE.
- **Récupération des scores EPSS** : `epss_scores = fetch_epss_scores()` récupère des données relatives aux scores EPSS (Exploit Prediction Scoring

System), qui sont utilisés pour évaluer la gravité des vulnérabilités.

Ces fonctions appellent potentiellement des API externes ou consultent des bases de données pour récupérer les informations en temps réel.

2. Combinaison des données

Une fois les données récupérées, elles sont organisées dans un dictionnaire Python `all_data` qui associe une clé à chaque jeu de données. Par exemple :

- `"cve_data"` : contient les informations sur les vulnérabilités.
- `"mitre_data"` : contient les métadonnées du framework MITRE ATT&CK.
- `"epss_scores"` : contient les scores EPSS associés aux vulnérabilités.

3. Retour des données

La fonction renvoie ensuite le dictionnaire `all_data`, qui contient toutes les données récupérées, sous un format structuré et facile à manipuler.

- **Structure des données** : Le dictionnaire `all_data` permet à l'application de manipuler les différentes sources de données de manière centralisée, sans avoir à gérer chaque source individuellement.
- **Réutilisation des données** : Ce format structuré permet également de faciliter l'utilisation et la réutilisation des données ailleurs dans l'application.

Pourquoi cette fonction est utile ?

- **Centralisation des appels** : La fonction centralise la récupération des données provenant de sources multiples, ce qui simplifie l'architecture du code et évite la duplication d'appels similaires.
- **Facilité d'accès aux données** : En combinant les données dans un dictionnaire unique, cette fonction permet à d'autres parties de l'application d'accéder facilement à toutes les informations nécessaires en une seule fois.
- **Réduction de la complexité** : Plutôt que d'effectuer des appels séparés chaque fois que l'application a besoin de données, cette fonction simplifie le processus et offre une vue d'ensemble centralisée de toutes les données.
- **Extensibilité** : Cette approche permet facilement d'ajouter d'autres sources de données dans le futur sans modifier profondément l'architecture de l'application.

La fonction _process_data_chunk

La fonction _process_data_chunk est responsable du traitement d'un fragment de données (ou *chunk*) récupéré dans un flux de données. Elle est utilisée pour effectuer des actions spécifiques sur des données partitionnées en morceaux, afin de les préparer ou de les analyser avant une utilisation ultérieure.

Code de la fonction _process_data_chunk

```
1 def _process_data_chunk(data_chunk):
2     # Traitement du chunk de donn es
3     processed_chunk = []
4
5     for item in data_chunk:
6         # Transformation ou extraction des
              informations n cessaires
7         processed_item = {
8             'id': item['id'],
9             'value': item['value'] * 2,  #
                  Exemple de transformation
10        }
11        processed_chunk.append(processed_item)
12
13    return processed_chunk
```

Explication de la fonction _process_data_chunk

La fonction _process_data_chunk prend en entrée un fragment de données (un *chunk*) et effectue des transformations ou extractions sur ces données. Elle retourne une version traitée du fragment de données.

1. Initialisation du fragment traité

- La fonction commence par initialiser une liste vide `processed_chunk` pour stocker les éléments transformés du fragment de données.

2. Boucle de traitement sur les éléments du fragment

- La fonction parcourt chaque élément (`item`) du `data_chunk`. Chaque élément représente généralement un ensemble de données ou une entrée qui doit être transformée avant d'être utilisée.
- À chaque itération de la boucle, un traitement spécifique est effectué sur l'élément. Dans l'exemple donné, un `processed_item` est créé à partir des données de l'élément, en extrayant certaines valeurs (comme l'`id`) et en appliquant une transformation (comme multiplier `value` par 2).

3. Ajout de l'élément traité à la liste

- L'élément transformé `processed_item` est ensuite ajouté à la liste `processed_chunk`.

4. Retour du fragment de données traité

- Après avoir parcouru tous les éléments du fragment et effectué les transformations nécessaires, la fonction retourne la liste `processed_chunk`, qui contient les éléments traités.

Pourquoi cette fonction est utile ?

- **Traitement des données en morceaux** : Cette fonction permet de traiter efficacement de grandes quantités de données en les divisant en morceaux (ou chunks). Cela rend le traitement plus flexible et efficace, notamment dans des systèmes où la mémoire est limitée.
- **Transformation des données** : Elle permet d'appliquer des transformations spécifiques sur chaque élément d'un fragment de données, ce qui peut être utile pour normaliser, filtrer ou ajuster les données avant qu'elles ne soient utilisées ailleurs dans l'application.
- **Modularité et réutilisation** : Cette fonction encapsule le traitement des données pour le rendre modulaire et réutilisable dans différents contextes, sans avoir à répéter les mêmes étapes de transformation à plusieurs endroits dans le code.
- **Efficacité** : Le traitement par fragments permet de traiter les données de manière plus efficace, surtout lorsque ces dernières sont trop volumineuses pour être manipulées en une seule fois. Cela peut améliorer les performances du système, notamment lorsqu'il s'agit de grands ensembles de données.

La fonction get_latest_modification_time

La fonction `get_latest_modification_time` est utilisée pour récupérer l'heure de la dernière modification d'un fichier ou d'un objet. Elle peut être utilisée pour vérifier si un fichier a été mis à jour, par exemple, dans un système de gestion de versions, ou pour savoir quand un fichier a été modifié pour la dernière fois.

Code de la fonction get_latest_modification_time

```
import os

def get_latest_modification_time(file_path: str) -> float:
    """
    R cup re l'heure de la derni re modification d'un fichier.

    :param file_path: Chemin du fichier v rifier
    :return: L'heure de la derni re modification en secondes depuis l' poque
    """
    try:
        # Utilisation de os.path.getmtime pour obtenir le temps de modification
        return os.path.getmtime(file_path)
    except FileNotFoundError:
        # Si le fichier n'existe pas, on renvoie None
        return None
```

Explication de la fonction get_latest_modification_time

La fonction `get_latest_modification_time` prend en entrée un chemin de fichier (`file_path`) et retourne l'heure de la dernière modification du fichier sous forme de timestamp (le nombre de secondes depuis le 1er janvier 1970, 00:00:00 UTC).

1. Vérification du chemin du fichier

- La fonction commence par essayer de récupérer l'heure de la dernière modification du fichier en utilisant `os.path.getmtime(file_path)`. Cette fonction renvoie un timestamp représentant le moment où le fichier a été modifié pour la dernière fois.

2. Gestion des erreurs

- Si le fichier spécifié par `file_path` n'existe pas (générant une exception `FileNotFoundError`), la fonction intercepte cette exception et retourne `None`. Cela permet d'éviter que le programme ne plante en cas d'absence du fichier.

3. Retour du résultat

- Si le fichier existe, la fonction retourne l'heure de la dernière modification sous forme de timestamp (un nombre flottant représentant le nombre de secondes depuis l'époque UNIX).
- Si le fichier n'existe pas, la fonction retourne `None` pour signaler l'absence du fichier.

Pourquoi cette fonction est utile ?

- **Vérification des mises à jour** : La fonction permet de vérifier si un fichier a été modifié depuis une certaine heure, ce qui est utile pour détecter des changements dans des fichiers de configuration, des journaux ou des fichiers de données.
- **Gestion des erreurs** : Elle gère de manière élégante l'absence de fichier, en retournant `None` au lieu de provoquer une erreur, ce qui permet d'éviter des interruptions dans l'exécution du programme.
- **Compatibilité multiplateforme** : Utilisant la fonction `os.path.getmtime`, elle fonctionne de manière fiable sur plusieurs systèmes d'exploitation (Windows, Linux, macOS), ce qui la rend portable et robuste.
- **Précision temporelle** : Elle fournit l'heure exacte de la dernière modification d'un fichier, ce qui peut être crucial pour des systèmes qui ont besoin de suivre l'intégrité ou la mise à jour de fichiers.

La structure `if __name__ == "__main__"`

La structure `if __name__ == "__main__"` est un constructe courant en Python. Elle permet de déterminer si un fichier est exécuté en tant que script principal ou s'il est importé comme module dans un autre fichier Python.

Code de la structure `if __name__ == "__main__"`

```
if __name__ == "__main__":
    # Code ex cut  lorsque le script est
        ex cut  directement
    main()
```

Explication de la structure `if __name__ == "__main__"`

Cette structure permet d'exécuter un code uniquement lorsque le fichier est exécuté directement, et non lorsqu'il est importé dans un autre fichier. Examinons les différentes parties de cette structure.

1. `__name__`

- `__name__` est une variable spéciale en Python. Elle contient le nom du module actuel.
- Si le fichier Python est exécuté directement (et non importé), `__name__` prend la valeur `"__main__"`.
- Si le fichier Python est importé dans un autre module, `__name__` contient le nom du fichier (sans l'extension `.py`).

2. Comparaison avec `"__main__"`

- La structure `if __name__ == "__main__"` compare la valeur de `__name__` avec la chaîne de caractères `"__main__"`.
- Si le fichier est exécuté directement, `__name__` vaudra `"__main__"`, et le code dans le bloc `if` sera exécuté.
- Si le fichier est importé comme module dans un autre fichier, `__name__` ne sera pas égal à `"__main__"`, et le code dans le bloc `if` ne sera pas exécuté.

3. Pourquoi utiliser cette structure ?

- Cette structure permet de séparer le code qui doit être exécuté lorsqu'un fichier est utilisé comme script principal de celui qui doit être exécuté lorsqu'il est importé comme module.
- Elle est utile pour écrire des fichiers Python réutilisables. Par exemple, vous pouvez définir des fonctions dans un fichier et les utiliser dans un autre fichier, mais vous ne voulez pas qu'un code spécifique s'exécute lors de l'importation.
- Cela améliore la lisibilité et la modularité du code.

Exemple d'utilisation

Imaginons un fichier `script.py` contenant la fonction `main()` et la structure `if __name__ == "__main__"`.

```
def main():
    print("Ce script est ex cut  directement
        .")

if __name__ == "__main__":
    main()
```

Si vous exécutez `script.py` directement, la fonction `main()` sera appelée, et vous verrez le message `"Ce script est exécuté directement."` dans la sortie.

Cependant, si vous importez ce fichier dans un autre module comme suit :

```
import script
```

Le code dans `main()` ne sera pas exécuté automatiquement. Seules les fonctions et variables définies dans `script.py` seront accessibles, et le programme principal ne sera pas lancé.

Pourquoi cette structure est-elle utile ?

- **Modularité** : Elle permet de structurer le code pour qu'il puisse être réutilisé dans d'autres fichiers sans exécuter des parties non désirées du script.
- **Testabilité** : Elle permet de tester des modules indépendamment. Le code à l'intérieur de `if __name__ == "__main__"` ne sera exécuté que lorsque le fichier est lancé directement, ce qui permet de tester la logique sans exécuter des portions de code inutiles.
- **Contrôle du flux d'exécution** : Elle donne un contrôle explicite sur ce qui doit se passer lorsque le script est exécuté directement ou importé comme un module.

Introduction à asyncio

La bibliothèque `asyncio` est une bibliothèque standard en Python qui permet de gérer des tâches asynchrones et de réaliser de la programmation concurrente. Elle est particulièrement utilisée pour les applications nécessitant de nombreuses opérations d'entrée/sortie, telles que les serveurs web, les applications réseau et les tâches en parallèle.

Elle fournit un cadre pour écrire des programmes concurrentiels en utilisant des coroutines, des événements et des boucles d'événements.

Les concepts clés de asyncio

1. async et await

- La bibliothèque `asyncio` repose sur l'utilisation de `async` et `await`.
- `async` est utilisé pour définir une fonction asynchrone, une coroutine, qui peut être suspendue et reprise à une date ultérieure.
- `await` est utilisé à l'intérieur d'une coroutine pour suspendre son exécution et attendre le résultat d'une autre coroutine.

Exemple de fonction asynchrone :

```
import asyncio

async def my_coroutine():
    print("Start")
    await asyncio.sleep(1)  # Attend pendant 1
        seconde
    print("End")
```

Ici, `asyncio.sleep(1)` est une coroutine qui fait une pause de 1 seconde. La fonction `await` permet de suspendre la coroutine pendant cette pause sans bloquer l'exécution du programme.

2. La boucle d'événements (event loop)

La boucle d'événements est au cœur de `asyncio`. C'est elle qui gère l'exécution des coroutines, en planifiant leur exécution, en suspendant celles en attente, et en reprenant leur exécution lorsque c'est nécessaire.

Exemple de boucle d'événements :

```
import asyncio

async def my_coroutine():
    print("Start")
    await asyncio.sleep(1)  # Attend pendant 1
        seconde
    print("End")

# Ex cution de la boucle d'   vnements
asyncio.run(my_coroutine())
```

Ici, `asyncio.run()` démarre la boucle d'événements et exécute la coroutine `my_coroutine`.

3. asyncio.sleep()

La fonction `asyncio.sleep()` est une fonction asynchrone utilisée pour faire une pause sans bloquer la boucle d'événements. C'est souvent utilisée dans les tâches asynchrones pour simuler une attente ou une opération d'entrée/sortie.

Exemple :

```
import asyncio

async def task_with_sleep():
    print("Before sleep")
    await asyncio.sleep(2)  # Pauser pendant 2
        secondes
    print("After sleep")

asyncio.run(task_with_sleep())
```

4. Gestion des tâches concurrentes avec asyncio.create_task()

`asyncio.create_task()` permet de planifier l'exécution d'une coroutine en arrière-plan, ce qui permet d'effectuer des tâches concurrentes.

Exemple de gestion de plusieurs tâches concurrentes :

```
import asyncio

async def task1():
    await asyncio.sleep(1)
    print("Task 1 finished")

async def task2():
    await asyncio.sleep(2)
    print("Task 2 finished")

async def main():
    task1_obj = asyncio.create_task(task1())
    task2_obj = asyncio.create_task(task2())

    await task1_obj
    await task2_obj

asyncio.run(main())
```

Ici, les deux tâches sont exécutées en parallèle. La fonction `main()` attend que les deux tâches soient terminées avant de finir l'exécution.

5. asyncio.gather()

`asyncio.gather()` permet de grouper plusieurs coroutines et d'attendre qu'elles soient toutes terminées.

Exemple avec `asyncio.gather()` :

```
import asyncio

async def task1():
    await asyncio.sleep(1)
    print("Task 1 finished")

async def task2():
    await asyncio.sleep(2)
    print("Task 2 finished")

async def main():
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

`asyncio.gather()` permet d'attendre que toutes les tâches spécifiées soient terminées avant de continuer.

Pourquoi utiliser `asyncio` ?

- **Gestion des opérations I/O intensives** : `asyncio` est particulièrement utile pour les applications qui effectuent de nombreuses opérations d'entrée/sortie (lecture de fichiers, requêtes réseau, etc.).
- **Non-bloquant** : Avec `asyncio`, les tâches peuvent s'exécuter de manière non-bloquante, ce qui permet d'améliorer la réactivité et les performances dans des environnements fortement concurrents.
- **Prise en charge de la concurrence** : Permet d'exécuter plusieurs tâches en parallèle sans nécessiter de threads ou de processus séparés.
- **Facilité de programmation** : Le modèle de programmation avec les coroutines rend la gestion des tâches asynchrones plus facile et plus lisible que l'utilisation de callbacks ou de threads.

Résumé

`asyncio` est un excellent choix pour gérer des programmes concurrentiels et asynchrones en Python, particulièrement pour les applications ayant des opérations d'entrée/sortie multiples. Son modèle basé sur les coroutines et la boucle d'événements facilite l'écriture de code asynchrone lisible et efficace.

Introduction à `aiohttp`

La bibliothèque `aiohttp` est une bibliothèque Python permettant de réaliser des requêtes HTTP de manière asynchrone. Elle repose sur `asyncio` et permet d'effectuer des tâches d'entrée/sortie (I/O) non-bloquantes, ce qui est essentiel pour développer des applications réseau à haute performance.

Elle permet de créer des serveurs HTTP asynchrones ainsi que des clients HTTP asynchrones, tout en garantissant une gestion fluide des connexions et de la concurrence. `aiohttp` est particulièrement utile dans les applications nécessitant des appels réseau non-bloquants, tels que les applications web modernes, les crawlers, ou les services qui doivent gérer de nombreuses requêtes simultanées.

Les concepts clés de `aiohttp`

1. Client HTTP avec `aiohttp.ClientSession`

Le client HTTP de `aiohttp` est basé sur la classe `ClientSession`. Cette classe gère les connexions HTTP et permet d'effectuer des requêtes asynchrones.

Exemple de requête GET avec `aiohttp` :

```
import aiohttp
import asyncio

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            data = await response.json()
            return data

async def main():
    url = 'https://api.example.com/data'
    result = await fetch_data(url)
    print(result)

# Lancer la boucle d'  vnements   pour ex cuter la t che
asyncio.run(main())
```

Ici, nous utilisons `aiohttp.ClientSession` pour créer une session HTTP et effectuer une requête GET vers l'URL spécifiée. Nous attendons la réponse avec `await`, puis nous extrayons les données en format JSON.

2. Gestion des erreurs HTTP

Dans `aiohttp`, il est important de gérer les erreurs HTTP, notamment les codes de réponse non OK. Cela peut être fait en vérifiant le code d'état de la réponse.

Exemple de gestion des erreurs :

```
async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            if response.status != 200:
                return f"Error: {response.status}"
            data = await response.json()
            return data
```

Dans cet exemple, si le code d'état HTTP est différent de 200, une erreur est retournée. Sinon, les données JSON sont extraites.

3. Client HTTP avec des en-têtes personnalisés

`aiohttp` permet également de personnaliser les en-têtes HTTP des requêtes. Cela est utile pour envoyer des informations supplémentaires comme des jetons d'authentification ou des types de contenu spécifiques.

Exemple d'ajout d'en-têtes personnalisés :

```
async def fetch_data_with_headers(url):
    headers = {'Authorization': 'Bearer my_token'}
    async with aiohttp.ClientSession() as session:
        async with session.get(url, headers=headers) as response:
            data = await response.json()
            return data
```

Ici, un en-tête `Authorization` est ajouté à la requête HTTP, ce qui est souvent utilisé pour l'authentification par jeton.

4. Serveur HTTP avec `aiohttp.web.Application`

En plus de fonctionner comme un client HTTP, `aiohttp` permet également de créer des serveurs HTTP

asynchrones. Cela se fait en utilisant la classe `aiohttp.web.Application` et en définissant des gestionnaires de routes pour les différentes requêtes.

Exemple de serveur HTTP simple :

```
from aiohttp import web

async def handle(request):
    return web.Response(text="Hello, world")

app = web.Application()
app.add_routes([web.get('/', handle)])

if __name__ == '__main__':
    web.run_app(app)
```

Dans cet exemple, un serveur HTTP est créé avec `aiohttp.web.Application`, et une route est définie pour gérer les requêtes GET à la racine de l'application.

5. Gestion des tâches concurrentes avec `asyncio.gather()`

Lorsqu'on effectue plusieurs requêtes HTTP simultanément avec `aiohttp`, il est souvent nécessaire de gérer ces requêtes de manière concurrente. Cela peut être réalisé avec `asyncio.gather()` pour lancer plusieurs tâches en parallèle.

Exemple de requêtes concurrentes avec `asyncio.gather()` :

```
async def fetch_data(url):
    async with aiohttp.ClientSession() as
        session:
        async with session.get(url) as
            response:
            return await response.json()

async def main():
    urls = ['https://api.example.com/data1', '
        https://api.example.com/data2']
    results = await asyncio.gather(*[
        fetch_data(url) for url in urls])
    print(results)

asyncio.run(main())
```

Ici, `asyncio.gather()` permet de lancer deux requêtes HTTP en parallèle, et une fois que les deux sont terminées, les résultats sont imprimés.

Pourquoi utiliser `aiohttp` ?

- **Asynchrone et non-bloquant** : `aiohttp` permet d'effectuer des opérations HTTP sans bloquer l'exécution du programme, ce qui est essentiel dans les environnements hautement concurrentiels.
- **Haute performance** : Grâce à son intégration avec `asyncio`, `aiohttp` permet de gérer un grand nombre de connexions simultanées avec une faible surcharge.
- **Facilité d'utilisation** : La syntaxe de `aiohttp` est simple et intégrée dans le modèle de programmation asynchrone de Python avec `async` et `await`.
- **Flexibilité** : `aiohttp` permet de créer des clients et des serveurs HTTP, et de gérer facilement les entêtes, les paramètres et les erreurs HTTP.

Résumé

La bibliothèque `aiohttp` est idéale pour développer des applications HTTP asynchrones en Python. Elle permet de créer des serveurs et des clients HTTP efficaces et scalables. Son intégration avec `asyncio` permet de tirer parti des avantages de la programmation concurrente et asynchrone. `aiohttp` est particulièrement utile pour les applications nécessitant de nombreuses requêtes simultanées sans bloquer le programme principal.

Introduction à `feedparser`

`feedparser` est une bibliothèque Python conçue pour analyser les flux RSS et Atom. Elle permet de parser facilement les flux XML des sites web afin de récupérer et d'analyser des données telles que des titres, des liens, des résumés, et plus encore. Cette bibliothèque est particulièrement utile pour les applications qui doivent traiter et afficher des informations provenant de sources dynamiques et actualisées fréquemment, telles que des blogs, des sites d'actualités, ou des podcasts.

L'utilisation de `feedparser` est simple, et elle supporte les flux RSS et Atom dans leur version la plus courante, en gérant automatiquement les différences entre les versions de ces formats.

Les concepts clés de `feedparser`

1. Parser un flux RSS ou Atom

La fonction principale de `feedparser` est de parser un flux RSS ou Atom. Un flux RSS est un fichier XML qui contient des informations comme les titres des articles, les résumés, et les liens vers les articles complets. `feedparser` permet de récupérer ces informations sous forme de dictionnaires Python.

Exemple de parsing d'un flux RSS :

```
import feedparser

# URL du flux RSS
url = "https://example.com/rss"

# Parse le flux
feed = feedparser.parse(url)

# Affiche les titres des articles
for entry in feed.entries:
    print(entry.title)
    print(entry.link)
    print(entry.summary)
```

Dans cet exemple, la fonction `feedparser.parse()` charge le flux RSS depuis l'URL donnée. Ensuite, les titres des articles, les liens et les résumés sont extraits et affichés.

2. Structure d'un flux RSS

Un flux RSS est une collection de "items" qui contiennent des informations sur des articles ou des mises à jour. Chaque "item" dans un flux RSS contient des

éléments comme le titre (`title`), le lien (`link`), la description (`summary`), la date de publication (`published`) et d'autres métadonnées.

Exemple d'accès aux éléments d'un item :

```
for entry in feed.entries:
    print(f"Titre : {entry.title}")
    print(f"Lien : {entry.link}")
    print(f"R sum  : {entry.summary}")
    print(f"Publi le : {entry.published}")
```

Dans cet exemple, nous extrayons et affichons les informations les plus courantes pour chaque article d'un flux RSS.

3. Gestion des erreurs dans le parsing

Il est important de gérer les erreurs potentielles lors du parsing d'un flux, surtout si l'URL fournie est incorrecte ou si le flux n'est pas bien formé. `feedparser` renvoie un objet avec un attribut `bozo` qui permet de vérifier si des erreurs de parsing ont eu lieu.

Exemple de gestion des erreurs :

```
feed = feedparser.parse(url)

if feed.bozo:
    print("Erreur lors du parsing du flux RSS")
else:
    for entry in feed.entries:
        print(entry.title)
```

Ici, l'attribut `bozo` est utilisé pour détecter les erreurs de parsing. Si `bozo` est `True`, cela signifie qu'il y a eu un problème lors de la lecture du flux.

4. Paramètres de la fonction `feedparser.parse()`

La fonction `parse()` de `feedparser` prend en charge des arguments supplémentaires comme un délai d'attente (`timeout`) pour les requêtes HTTP et un paramètre pour activer ou désactiver le décryptage des entités HTML (`sanitize_html`).

Exemple d'utilisation du délai d'attente et du nettoyage HTML :

```
feed = feedparser.parse(url, timeout=10,
    sanitize_html=True)

for entry in feed.entries:
    print(entry.title)
```

Dans cet exemple, nous définissons un délai d'attente de 10 secondes pour la requête HTTP et activons l'option de nettoyage des entités HTML dans les descriptions d'articles.

5. Accéder à d'autres métadonnées du flux

En plus des éléments individuels des articles, `feedparser` permet également d'accéder à des informations globales sur le flux RSS, comme le titre du flux (`feed.title`), la description (`feed.subtitle`), et l'URL de l'auto-référencement (`feed.link`).

Exemple d'accès aux métadonnées du flux :

```
print(f"Titre du flux : {feed.feed.title}")
print(f"Description : {feed.feed.subtitle}")
print(f"Lien : {feed.feed.link}")
```

Dans cet exemple, nous récupérons le titre, la description et l'URL du flux RSS en accédant aux attributs correspondants de l'objet `feed`.

Pourquoi utiliser `feedparser` ?

- **Simplicité** : `feedparser` offre une API simple et directe pour analyser les flux RSS et Atom.
- **Compatibilité** : Elle supporte à la fois les flux RSS et Atom dans plusieurs versions, et gère les différences entre elles de manière transparente.
- **Flexibilité** : `feedparser` permet d'extraire facilement des métadonnées globales du flux ainsi que des informations détaillées sur chaque article.
- **Gestion des erreurs** : La bibliothèque offre des outils pour vérifier la validité des flux et gérer les erreurs de parsing, rendant les applications robustes.

Résumé

`feedparser` est une bibliothèque Python très utile pour traiter des flux RSS et Atom. Elle permet d'extraire des informations facilement et de manière flexible, tout en gérant automatiquement les aspects complexes du parsing XML. `feedparser` est un choix idéal pour les applications qui nécessitent d'interagir avec des sources de contenu dynamique, comme des blogs, des sites d'actualités, ou des podcasts.

Introduction à `datetime`

La bibliothèque `datetime` de Python permet de travailler avec les dates et les heures. Elle fournit des classes pour représenter des dates, des heures et des durées, ainsi que des fonctions pour manipuler et effectuer des calculs sur ces objets. Les modules `datetime` et `timedelta` sont souvent utilisés pour la gestion de données temporelles dans les applications.

Dans cette section, nous explorons les deux classes les plus couramment utilisées : `datetime` et `timedelta`, ainsi que leurs méthodes les plus utiles.

Les classes principales de `datetime`

1. `datetime` : Manipulation de dates et heures

La classe `datetime` permet de manipuler des objets représentant des dates et des heures. Elle offre des fonctionnalités pour obtenir l'heure actuelle, créer des objets de dates et heures, et effectuer des calculs de date.

Exemple de création d'un objet `datetime` :

```
from datetime import datetime

# Cr er un objet datetime avec la date et l'
    heure actuelles
current_time = datetime.now()
print(current_time)
```

Ici, la méthode `now()` de la classe `datetime` retourne un objet représentant la date et l'heure actuelles.

2. Formatage des dates et heures

La méthode `strftime()` permet de convertir un objet `datetime` en une chaîne de caractères formatée selon un format spécifié.

Exemple de formatage d'un objet `datetime` :

```
# Formatage de l'objet datetime
formatted_time = current_time.strftime("%Y-%m
    -%d %H:%M:%S")
print(formatted_time)
```

Dans cet exemple, l'objet `current_time` est converti en une chaîne de caractères dans le format `YYYY-MM-DD HH:MM:SS`.

3. `timedelta` : Manipulation des durées

La classe `timedelta` représente une différence entre deux objets `datetime`. Elle permet de calculer des durées et d'effectuer des opérations sur les dates et heures, comme ajouter ou soustraire des jours, des heures, des minutes, etc.

Exemple d'utilisation de `timedelta` :

```
from datetime import timedelta

# Ajouter 5 jours    la date actuelle
new_date = current_time + timedelta(days=5)
print(new_date)
```

Ici, nous ajoutons 5 jours à la date actuelle en utilisant un objet `timedelta`.

4. Comparaison de dates et heures

Les objets `datetime` peuvent être comparés entre eux pour vérifier si une date est antérieure, postérieure ou égale à une autre.

Exemple de comparaison de deux objets `datetime` :

```
# Cr er un autre objet datetime
future_time = current_time + timedelta(days
    =10)

# Comparaison des dates
if future_time > current_time:
    print("La date future est apr s la date
        actuelle.")
```

Dans cet exemple, nous comparons `future_time` et `current_time` pour déterminer si la date future est postérieure à la date actuelle.

5. Conversion entre `datetime` et `timedelta`

Il est souvent nécessaire de convertir des objets `timedelta` en objets `datetime` et vice versa. Cela peut être réalisé en ajoutant ou en soustrayant un objet `timedelta` d'un objet `datetime`.

Exemple de conversion entre `datetime` et `timedelta` :

```
# Soustraire 3 jours de la date actuelle
past_time = current_time - timedelta(days=3)
print(past_time)
```

Ici, nous soustrayons 3 jours à la date actuelle pour obtenir une date passée.

Les méthodes principales de `datetime`

- `datetime.now()` : Retourne la date et l'heure actuelles.
- `datetime.strptime(date_string, format)` : Convertit une chaîne de caractères représentant une date en un objet `datetime` en utilisant un format spécifié.
- `datetime.strftime(format)` : Convertit un objet `datetime` en une chaîne de caractères formatée selon un format spécifié.
- `timedelta(days, seconds, microseconds, milliseconds, minutes, hours, weeks)` : Crée un objet `timedelta` représentant une différence de temps.

Pourquoi utiliser `datetime` ?

- **Précision temporelle** : La classe `datetime` permet de gérer les dates et les heures avec une grande précision.
- **Calculs temporels** : Elle offre des outils puissants pour ajouter ou soustraire des durées, ce qui est essentiel dans de nombreuses applications.
- **Manipulation de dates et heures** : Vous pouvez facilement obtenir la date et l'heure actuelles, les formater ou les comparer pour effectuer des actions en fonction du temps.
- **Facilité d'utilisation** : La bibliothèque `datetime` est simple à utiliser et répond à la plupart des besoins liés aux dates et aux heures dans les applications.

Résumé

La bibliothèque `datetime` de Python est essentielle pour travailler avec les dates, heures et durées. Elle offre de nombreuses fonctionnalités, telles que la manipulation, le formatage, la comparaison et les calculs sur les dates. Grâce à la classe `timedelta`, vous pouvez également effectuer facilement des calculs sur les périodes de temps. Elle est largement utilisée dans les applications qui nécessitent de suivre, comparer ou manipuler des informations temporelles.

Introduction à pandas

La bibliothèque **pandas** est une des bibliothèques les plus populaires de Python pour la manipulation et l'analyse de données. Elle fournit des structures de données flexibles et puissantes, telles que les **DataFrame** et les **Series**, qui facilitent la manipulation, le nettoyage, l'agrégation, la transformation, et l'analyse de données.

Dans cette section, nous explorons les principales fonctionnalités de **pandas**, ses structures de données et ses méthodes les plus courantes.

Les structures principales de pandas

1. DataFrame : Table de données

Le **DataFrame** est la structure de données principale de **pandas**, représentant une table bidimensionnelle de données avec des colonnes et des indices. Chaque colonne peut contenir des données de types différents.

Exemple de création d'un **DataFrame** :

```
import pandas as pd

# Cr er un DataFrame    partir d'un
    dictionnaire
data = {'Nom': ['Alice', 'Bob', 'Charlie'],
        ' ge ': [25, 30, 35],
        'Ville': ['Paris', 'Londres', 'Berlin'
            ]}
df = pd.DataFrame(data)
print(df)
```

Le résultat de cet exemple serait un tableau avec trois colonnes : **Nom**, **Âge**, et **Ville**.

2. Series : Colonne de données

Une **Series** est une structure de données unidimensionnelle dans **pandas**, semblable à un tableau ou une liste. Elle peut contenir des données de type homogène et est souvent utilisée pour représenter une seule colonne d'un **DataFrame**.

Exemple de création d'une **Series** :

```
# Cr er une Series    partir d'une liste
age_series = pd.Series([25, 30, 35], index=['
    Alice', 'Bob', 'Charlie'])
print(age_series)
```

Dans cet exemple, une **Series** est créée pour représenter les âges des personnes avec des indices nommés (les noms des personnes).

Manipulations de données avec pandas

1. Sélection de données

Il existe plusieurs façons de sélectionner des données dans un **DataFrame** ou une **Series**. Vous pouvez accéder aux données par leur indice, leur étiquette de colonne, ou par des conditions.

Exemple de sélection par colonne :

```
# S lectionner une colonne
print(df['Nom'])
```

Exemple de sélection par ligne (utilisation de **iloc** pour l'indexation par position) :

```
# S lectionner la premi re ligne
print(df.iloc[0])
```

Exemple de sélection par condition (lignes avec un âge supérieur à 30) :

```
# S lectionner les lignes o   l' ge  est
    sup rieur    30
print(df[df[' ge '] > 30])
```

2. Modification des données

Il est facile de modifier les valeurs dans un **DataFrame** ou une **Series** en utilisant des indexations ou des méthodes.

Exemple de modification de valeur :

```
# Modifier une valeur dans le DataFrame
df.at[1, ' ge '] = 32  # Modifier l' ge  de
    Bob
print(df)
```

Exemple de création d'une nouvelle colonne :

```
# Ajouter une nouvelle colonne
df['Profession'] = ['Ing nieur', 'M decin',
    'Artiste']
print(df)
```

3. Agrégation des données

pandas permet d'effectuer des agrégations simples sur les données, comme le calcul de moyennes, de sommes, de minimums, de maximums, etc.

Exemple de calcul de la moyenne de l'âge :

```
# Calculer la moyenne de la colonne ' ge '
average_age = df[' ge '].mean()
print(average_age)
```

Exemple de groupe par catégorie et calcul de la moyenne :

```
# Grouper les donn es par profession et
    calculer la moyenne d' ge
grouped_data = df.groupby('Profession')[' ge '
    ].mean()
print(grouped_data)
```

4. Lecture et écriture de fichiers

pandas permet de lire et d'écrire facilement des données depuis et vers des fichiers de différents formats, comme CSV, Excel, JSON, etc.

Exemple de lecture d'un fichier CSV :

```
# Lire un fichier CSV
df_from_csv = pd.read_csv('data.csv')
print(df_from_csv)
```

Exemple d'écriture d'un **DataFrame** dans un fichier CSV :

```
# Sauvegarder un DataFrame dans un fichier CSV
df.to_csv('output.csv', index=False)
```

Les méthodes principales de pandas

- `pd.DataFrame(data)` : Créer un `DataFrame` à partir de données (listes, dictionnaires, etc.).
- `df.iloc[index]` : Accéder à une ligne par son indice.
- `df['colonne']` : Accéder à une colonne par son nom.
- `df.groupby('colonne').mean()` : Grouper les données par une colonne et calculer la moyenne.
- `df.to_csv(file)` : Sauvegarder un `DataFrame` dans un fichier CSV.

Pourquoi utiliser pandas ?

- **Manipulation des données simplifiée** : Grâce à ses structures de données flexibles, `pandas` facilite la manipulation de données complexes.
- **Vitesse et efficacité** : `pandas` est conçu pour traiter des volumes importants de données de manière rapide et efficace.
- **Richesse des fonctionnalités** : `pandas` propose un large éventail de fonctionnalités pour la transformation, l'agrégation et l'analyse des données.
- **Interopérabilité avec d'autres bibliothèques** : `pandas` s'intègre facilement avec d'autres bibliothèques Python pour l'analyse des données, comme `matplotlib` pour la visualisation, ou `scikit-learn` pour l'apprentissage automatique.

Résumé

`pandas` est une bibliothèque Python puissante et flexible pour la manipulation de données. Avec ses structures de données `DataFrame` et `Series`, elle permet de travailler facilement avec des données tabulaires. `pandas` offre de nombreuses fonctionnalités, allant de la lecture et écriture de fichiers à l'agrégation et la transformation des données. Elle est indispensable pour toute analyse de données avancée en Python.

Introduction à Flask

`Flask` est un micro-framework web pour Python, conçu pour être simple, léger et flexible. Il permet de développer rapidement des applications web en Python en fournissant les outils de base nécessaires à la création de sites web et d'API.

Dans cette section, nous explorerons les fonctionnalités principales de `Flask`, y compris la gestion des routes, des templates et des réponses HTTP.

Les concepts de base de Flask

1. Création d'une application Flask

Pour commencer à utiliser `Flask`, il faut d'abord créer une instance de la classe `Flask` et définir les routes qui correspondent aux URL que l'application va gérer.

Exemple de création d'une application Flask :

```python
from flask import Flask

# Cr er une application Flask
app = Flask(__name__)

# D finir une route
@app.route('/')
def home():
    return "Bienvenue dans Flask !"
```

Dans cet exemple, nous créons une instance de `Flask` et définissons une route pour la page d'accueil (`/`). Lorsque l'utilisateur accède à cette URL, la fonction `home` est appelée et retourne un message simple.

2. Routes et gestion des requêtes

Les routes permettent de lier une fonction à une URL spécifique. Dans Flask, cela se fait à l'aide du décorateur `@app.route`.

Exemple d'utilisation d'une route avec un paramètre dynamique :

```python
@app.route('/user/<username>')
def show_user_profile(username):
    return f'Profil de l\'utilisateur {
        username}'
```

Dans cet exemple, la route `/user/{username}` permet d'accéder à un profil utilisateur en fonction du nom d'utilisateur passé dans l'URL.

3. Utilisation des templates

Flask intègre un moteur de templates appelé `Jinja2`, qui permet de générer des pages HTML dynamiquement à partir de modèles.

Exemple de rendu d'un template :

```python
from flask import render_template

@app.route('/hello')
def hello():
    return render_template('hello.html', name=
        'Alice')
```

Ici, `render_template` permet de rendre un fichier `hello.html` en y passant une variable `name`. Le fichier HTML pourrait ressembler à ceci :

```html
<!DOCTYPE html>
<html>
  <body>
    <h1>Bonjour {{ name }} !</h1>
  </body>
</html>
```

Dans cet exemple, `Flask` remplace `name` par la valeur "Alice".

4. Gestion des réponses HTTP

Flask permet de créer facilement des réponses HTTP via des fonctions de vue. Vous pouvez également définir des réponses JSON avec la fonction `jsonify`.

Exemple de réponse JSON :

```
from flask import jsonify

@app.route('/api/data')
def get_data():
    data = {'key': 'value'}
    return jsonify(data)
```

Dans cet exemple, l'API retourne un objet JSON contenant une clé et une valeur.

5. Exécution de l'application

Une fois l'application Flask définie, vous pouvez la lancer avec la méthode `run()` qui démarre un serveur web local.

Exemple d'exécution de l'application :

```
if __name__ == '__main__':
    app.run(debug=True)
```

L'application sera alors accessible à l'adresse `http://127.0.0.1:5000/`.

Les fonctionnalités principales de `Flask`

- `Flask(__name__)` : Créer une instance de l'application Flask.
- `@app.route(path)` : Décorateur qui associe une fonction à une URL spécifique.
- `render_template(template, **context)` : Rendre un template HTML avec des variables contextuelles.
- `jsonify(data)` : Retourner une réponse JSON.
- `app.run()` : Démarrer le serveur Flask pour l'application web.

Pourquoi utiliser `Flask` ?

- **Simplicité** : `Flask` est minimaliste et facile à apprendre, ce qui en fait un excellent choix pour les petits projets ou pour les développeurs débutants.
- **Flexibilité** : `Flask` permet de personnaliser et d'ajouter facilement des extensions et des fonctionnalités supplémentaires au besoin.
- **Micro-framework** : Étant un micro-framework, `Flask` ne vous oblige pas à utiliser des outils ou des bibliothèques supplémentaires, ce qui le rend léger et modulable.
- **Communauté active** : Avec une large communauté d'utilisateurs et une abondance de ressources en ligne, il est facile de trouver de l'aide et des exemples pour résoudre des problèmes.

Résumé

`Flask` est un micro-framework flexible et léger pour le développement d'applications web avec Python. Il permet de créer rapidement des routes, de gérer des templates HTML, et de répondre aux requêtes HTTP, tout en offrant une grande liberté dans l'architecture des applications. Grâce à sa simplicité et à ses nombreuses extensions, `Flask` est un excellent choix pour développer des applications web et des API RESTful.

Introduction à `typing`

Le module `typing` de Python permet d'ajouter des annotations de type à votre code. Ces annotations facilitent la lecture et la compréhension du code, et permettent également d'utiliser des outils de vérification statique pour s'assurer que le code respecte les types attendus. Ce module est particulièrement utile pour améliorer la lisibilité et la maintenabilité du code, notamment dans les projets de grande envergure.

Dans cette section, nous allons explorer l'utilisation des annotations de type avec les types `List` et `Dict` du module `typing`.

Les types dans `typing`

1. `List` : Une liste typée

La classe `List` permet de spécifier un type particulier pour les éléments d'une liste. Cela vous permet de déclarer que tous les éléments d'une liste doivent être d'un type spécifique, offrant ainsi une meilleure sécurité de type et une vérification statique plus précise.

Exemple d'utilisation de `List` :

```
from typing import List

def process_items(items: List[int]) -> int:
    return sum(items)
```

Dans cet exemple, la fonction `process_items` prend une liste d'entiers `List[int]` et retourne un entier. Cette annotation permet de garantir que seuls des entiers peuvent être passés dans la liste `items`, ce qui simplifie la validation des types.

2. `Dict` : Un dictionnaire typé

Le type `Dict` permet de spécifier les types des clés et des valeurs d'un dictionnaire. Vous pouvez ainsi déclarer que les clés d'un dictionnaire doivent être d'un certain type, tout comme les valeurs associées à ces clés.

Exemple d'utilisation de `Dict` :

```
from typing import Dict

def process_data(data: Dict[str, int]) -> int:
    return sum(data.values())
```

Dans cet exemple, la fonction `process_data` prend un dictionnaire `Dict[str, int]` où les clés sont des chaînes de caractères (`str`) et les valeurs sont des entiers (`int`). La fonction retourne la somme des valeurs du dictionnaire.

3. Annotation des fonctions avec plusieurs types

Il est possible d'annoter les fonctions avec plusieurs types combinés pour des structures de données complexes. Par exemple, vous pouvez utiliser `List` et `Dict` ensemble pour spécifier une liste de dictionnaires.

Exemple d'annotation d'une liste de dictionnaires :

```
from typing import List, Dict

def process_batch(data: List[Dict[str, int]])
    -> int:
    return sum(d['value'] for d in data)
```

Dans cet exemple, `data` est une liste (`List`) de dictionnaires (`Dict`), où chaque dictionnaire a une clé `'value'` associée à une valeur entière. La fonction retourne la somme des valeurs associées à la clé `'value'` dans chaque dictionnaire de la liste.

Pourquoi utiliser `typing` ?

L'utilisation du module `typing` dans Python permet de rendre votre code plus lisible et plus fiable, en vous aidant à clarifier les types des données manipulées. Cela facilite la détection d'erreurs au moment de l'écriture et de la révision du code, surtout dans des projets à grande échelle. De plus, les outils de vérification de type, comme `mypy`, peuvent analyser votre code et signaler les incohérences de types avant même qu'il ne soit exécuté.

Les fonctionnalités principales de `typing`

- `List[type]` : Spécifie une liste d'éléments de type `type`.
- `Dict[key_type, value_type]` : Spécifie un dictionnaire où les clés sont de type `key_type` et les valeurs de type `value_type`.
- `Tuple[type, ...]` : Spécifie un tuple de types donnés.
- `Any` : Permet de spécifier qu'une valeur peut être de n'importe quel type.
- `Union[type1, type2, ...]` : Permet de spécifier que la valeur peut être de l'un des types donnés.
- `Optional[type]` : Spécifie que la valeur peut être de type `type` ou `None`.

Résumé

Le module `typing` de Python permet d'annoter les types dans les fonctions et les structures de données, ce qui améliore la lisibilité du code et aide à la détection des erreurs. L'utilisation de types comme `List` et `Dict` permet de garantir la conformité des types dans les collections, rendant ainsi le code plus robuste et facile à maintenir.

Bibliothèque `aiohttp`

La bibliothèque `aiohttp` est une bibliothèque Python asynchrone qui fournit des outils pour faire des requêtes HTTP de manière non bloquante, idéale pour les applications qui nécessitent une gestion simultanée de nombreuses connexions HTTP. Elle est particulièrement utilisée dans les applications web asynchrones et les clients HTTP hautement concurrentiels.

ClientTimeout

L'objet `ClientTimeout` est utilisé pour configurer les délais d'attente (timeouts) des requêtes HTTP effectuées avec `aiohttp`. Il permet de spécifier des limites de temps pour la connexion, la lecture, et les délais d'attente global dans le cadre des requêtes HTTP. Cela peut être particulièrement utile pour gérer les situations où une connexion réseau met trop de temps à répondre.

Voici un exemple d'utilisation de `ClientTimeout` :

```
from aiohttp import ClientTimeout

timeout = ClientTimeout(
    total=60,        # Temps maximal pour la
        requ te compl te
    connect=10,      # Temps maximal pour
         tablir  la connexion
    sock_connect=5,  # Temps maximal pour
         tablir  la connexion au socket
    sock_read=10     # Temps maximal pour lire
        la r ponse du serveur
)
```

Dans cet exemple, le paramètre `total` définit le délai total alloué pour effectuer la requête, tandis que `connect`, `sock_connect`, et `sock_read` permettent de spécifier des délais d'attente plus précis pour la connexion au serveur, l'établissement du socket et la lecture de la réponse.

Utilisation de `ClientTimeout` avec `aiohttp`

Une fois l'objet `ClientTimeout` configuré, il peut être passé à un client `aiohttp.ClientSession` pour effectuer des requêtes HTTP avec ces paramètres de délai d'attente.

Exemple d'utilisation dans une session HTTP :

```
import aiohttp

async with aiohttp.ClientSession(timeout=
    timeout) as session:
    async with session.get('https://example.
        com') as response:
        if response.status == 200:
            data = await response.json()
            print(data)
```

Dans cet exemple, une instance de `ClientSession` est créée avec un délai d'attente personnalisé en passant l'objet `timeout`. La requête HTTP est effectuée de manière asynchrone et respecte les délais d'attente définis. Si le serveur ne répond pas dans le temps imparti, une exception `TimeoutError` sera levée.

Résumé

La bibliothèque `aiohttp` permet de gérer des requêtes HTTP de manière asynchrone, et l'objet `ClientTimeout` offre une manière de spécifier les délais d'attente pour les connexions et les lectures de données. Cela est essentiel pour garantir des performances optimales et éviter que les requêtes ne bloquent le programme en attendant des réponses trop longues.

L'utilisation de `ClientTimeout` dans `aiohttp` aide à contrôler le temps que prend chaque requête et à gérer les situations où une connexion réseau pourrait échouer ou être trop lente.

Bibliothèque `aiohttp.client`

La bibliothèque `aiohttp.client` fait partie de la bibliothèque `aiohttp` et fournit les outils nécessaires pour effectuer des requêtes HTTP asynchrones à l'aide de la classe `ClientSession`. Elle permet de gérer les connexions réseau de manière efficace et asynchrone, ce qui est idéal pour les applications hautement concurrentielles où plusieurs requêtes HTTP doivent être effectuées simultanément.

ClientSession

La classe `ClientSession` est utilisée pour établir des connexions HTTP et envoyer des requêtes dans un environnement asynchrone. Elle est la principale classe pour interagir avec les serveurs HTTP et permet de gérer efficacement les sessions, la gestion des cookies, les en-têtes, et le pool de connexions.

Exemple de base d'utilisation de `ClientSession` pour effectuer une requête HTTP `GET` :

```
from aiohttp.client import ClientSession

async with ClientSession() as session:
    async with session.get('https://example.
        com') as response:
        if response.status == 200:
            data = await response.json()
            print(data)
```

Dans cet exemple : - `ClientSession` est utilisé pour créer une session HTTP. - La requête `GET` est envoyée à l'URL `https://example.com`. - Si la réponse est réussie (`status == 200`), les données sont extraites au format JSON.

Gestion des connexions

Une des principales caractéristiques de `ClientSession` est sa capacité à gérer plusieurs connexions simultanées. Par défaut, elle réutilise les connexions déjà établies pour minimiser les frais généraux de création et de fermeture des connexions.

Exemple d'utilisation avec des paramètres de timeout personnalisés :

```
from aiohttp.client import ClientSession
from aiohttp import ClientTimeout

timeout = ClientTimeout(total=60, connect=10)

async with ClientSession(timeout=timeout) as
    session:
    async with session.get('https://example.
        com') as response:
        if response.status == 200:
            data = await response.json()
            print(data)
```

Dans cet exemple, nous créons un objet `ClientTimeout` et l'associons à la session `ClientSession` pour spécifier des délais d'attente personnalisés.

Méthodes courantes de `ClientSession`

`ClientSession` fournit plusieurs méthodes pour effectuer différents types de requêtes HTTP :

- `session.get(url)` : Envoie une requête `GET`.
- `session.post(url, data)` : Envoie une requête `POST`.
- `session.put(url, data)` : Envoie une requête `PUT`.
- `session.delete(url)` : Envoie une requête `DELETE`.

Chacune de ces méthodes retourne une réponse (`response`) qui peut être lue de manière asynchrone.

Gestion des erreurs

Lorsque vous effectuez des requêtes HTTP avec `ClientSession`, il est important de gérer les erreurs potentielles, telles que les problèmes de connexion ou les réponses HTTP avec des codes d'erreur.

Exemple de gestion des erreurs avec `ClientSession` :

```
from aiohttp.client import ClientSession

async with ClientSession() as session:
    try:
        async with session.get('https://
            example.com') as response:
            response.raise_for_status()  #
                L ve une exception pour les
                codes d'erreur HTTP
            data = await response.json()
            print(data)
    except Exception as e:
        print(f"Erreur lors de la requ te
            HTTP : {e}")
```

Dans cet exemple, `raise_for_status()` est utilisé pour lever une exception si le code d'état HTTP de la réponse indique une erreur (par exemple, 404 ou 500). La gestion des exceptions permet de mieux contrôler le flux d'exécution et d'éviter que l'application ne plante en cas d'erreur.

Résumé

`ClientSession` est la classe principale pour effectuer des requêtes HTTP asynchrones avec `aiohttp`. Elle permet de gérer les connexions réseau de manière efficace

et d'envoyer des requêtes dans un environnement hautement concurrentiel. Grâce à sa capacité à réutiliser les connexions et à fournir un contrôle fin sur les délais d'attente et la gestion des erreurs, elle est idéale pour les applications nécessitant un grand nombre de requêtes HTTP simultanées. Elle permet aussi d'effectuer des requêtes HTTP de manière très flexible, en utilisant des méthodes telles que `get`, `post`, `put`, et `delete`.

Bibliothèque `re`

La bibliothèque `re` permet de travailler avec des expressions régulières en Python. Elle fournit des outils pour effectuer des correspondances de chaînes, des substitutions, et d'autres opérations sur des chaînes de caractères en utilisant des motifs définis par des expressions régulières.

Fonctions principales de `re`

`re` offre plusieurs fonctions utiles pour travailler avec des expressions régulières. Parmi les plus courantes, on trouve :

- `re.match(pattern, string)` : Recherche une correspondance au début de la chaîne `string` avec le motif `pattern`.
- `re.search(pattern, string)` : Recherche une correspondance dans toute la chaîne `string`.
- `re.findall(pattern, string)` : Retourne une liste de toutes les correspondances de `pattern` dans `string`.
- `re.sub(pattern, repl, string)` : Remplace toutes les correspondances du motif `pattern` par la chaîne `repl` dans `string`.
- `re.split(pattern, string)` : Divise `string` en utilisant le motif `pattern`.

Exemple d'utilisation de `re.sub`

La fonction `re.sub` permet de remplacer les occurrences d'un motif dans une chaîne par une nouvelle valeur. Par exemple, on peut l'utiliser pour supprimer des informations non désirées entre parenthèses dans une chaîne.

```
import re

text = "Vulnerabilit  critique (CVE-2025-01)"
cleaned_text = re.sub(r'\(.*?\)', '', text)
print(cleaned_text)  # Sortie : '
    Vulnerabilit  critique'
```

Dans cet exemple, `re.sub(r'.*?', '', text)` remplace tout texte entre parenthèses (inclus) par une chaîne vide. L'expression régulière `r'.*?'` correspond à tout ce qui se trouve entre parenthèses, y compris les parenthèses elles-mêmes.

Expressions régulières courantes

Voici quelques exemples d'expressions régulières courantes que vous pouvez utiliser avec la bibliothèque `re` :

- `.` : Correspond à n'importe quel caractère sauf un retour à la ligne.
- `\d` : Correspond à un chiffre (équivalent à `[0-9]`).
- `\D` : Correspond à tout sauf un chiffre.
- `\w` : Correspond à un caractère alphanumérique ou un underscore (_).
- `\W` : Correspond à tout sauf un caractère alphanumérique ou un underscore (_).
- `\s` : Correspond à un espace, un tab, ou un saut de ligne.
- `\S` : Correspond à tout sauf un espace, un tab, ou un saut de ligne.
- `^` : Correspond au début de la chaîne.
- `$` : Correspond à la fin de la chaîne.
- `[]` : Délimite un ensemble de caractères à correspondre.
- `()` : Délimite un groupe de correspondances.

Exemple d'utilisation de `re.findall`

La fonction `re.findall` permet de trouver toutes les correspondances d'un motif dans une chaîne. Elle retourne une liste de toutes les correspondances.

```
import re

text = "Les CVEs sont : CVE-2025-01, CVE
    -2025-02 et CVE-2025-03."
matches = re.findall(r'CVE-\d{4}-\d{2}', text)
print(matches)  # Sortie : ['CVE-2025-01', '
    CVE-2025-02', 'CVE-2025-03']
```

Ici, l'expression régulière `r'CVE-\4-\d2'` correspond à un motif de type `CVE-YYYY-YY`, où `.` désigne un chiffre et `4` et `2` spécifient respectivement le nombre exact de chiffres pour l'année et le mois.

Résumé

La bibliothèque `re` est un outil puissant pour effectuer des recherches, des remplacements et des découpages de chaînes à l'aide d'expressions régulières en Python. Elle permet de manipuler des textes de manière très flexible et peut être utilisée pour des tâches telles que l'extraction de données, le nettoyage de chaînes, et la validation de formats spécifiques.

Bibliothèque subprocess

La bibliothèque `subprocess` permet d'exécuter des processus enfants (c'est-à-dire de lancer des programmes externes depuis un script Python) et d'interagir avec eux. Elle offre une interface simple pour la gestion des entrées/sorties des processus externes, ainsi que pour récupérer leurs résultats.

Fonctions principales de subprocess

`subprocess` fournit plusieurs fonctions pour interagir avec des processus externes. Les principales sont les suivantes :

- `subprocess.run()` : Lance un processus externe et attend sa fin. Il peut retourner un objet `CompletedProcess` contenant des informations sur l'exécution du processus.
- `subprocess.call()` : Lance un processus externe et attend sa fin. Il retourne le code de retour du processus.
- `subprocess.Popen()` : Crée un nouveau processus et fournit un contrôle plus précis sur les flux d'entrée/sortie du processus. C'est l'outil le plus flexible.
- `subprocess.check_call()` : Lance un processus et attend sa fin. Si le processus retourne un code de retour non nul, une exception `CalledProcessError` est levée.
- `subprocess.check_output()` : Lance un processus et attend sa fin. Retourne la sortie standard du processus, ou lève une exception si le code de retour est non nul.

Exemple d'utilisation de subprocess.run

`subprocess.run` permet de lancer un processus et d'attendre son terme. Il retourne un objet `CompletedProcess` contenant des informations sur l'exécution, telles que le code de retour et les sorties du processus.

```
import subprocess

# Exemple d'ex cution d'une commande syst me
result = subprocess.run(['ls', '-l'],
    capture_output=True, text=True)

# Affichage de la sortie standard
print(result.stdout)
```

Dans cet exemple, la commande `ls -l` est exécutée. L'argument `capture_output=True` permet de capturer la sortie standard et l'erreur standard du processus. L'argument `text=True` indique que les sorties doivent être traitées en tant que chaînes de caractères plutôt que des octets.

Exemple d'utilisation de subprocess.Popen

La classe `Popen` permet un contrôle plus détaillé sur les processus. Elle permet de rediriger les entrées/sorties et de gérer la communication avec le processus en temps réel.

```
import subprocess

# Ex cution d'un processus avec redirection
    des flux d'entr e/sortie
process = subprocess.Popen(
    ['echo', 'Hello World'],
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE
)

# Lecture de la sortie standard
stdout, stderr = process.communicate()

# Affichage de la sortie
print(stdout.decode())
```

Dans cet exemple, `Popen` est utilisé pour exécuter la commande `echo 'Hello World'` et récupérer sa sortie via `stdout=subprocess.PIPE`. La méthode `communicate()` attend la fin du processus et renvoie la sortie standard et l'erreur standard.

Gestion des erreurs avec check_output

`subprocess.check_output` permet de lancer un processus et de récupérer sa sortie. Si le processus échoue (c'est-à-dire s'il retourne un code de retour non nul), une exception `CalledProcessError` est levée.

```
import subprocess

try:
    output = subprocess.check_output(['ls', '/
        nonexistent'], stderr=subprocess.
        STDOUT, text=True)
except subprocess.CalledProcessError as e:
    print(f"Erreur lors de l'ex cution de la
        commande : {e.output}")
```

Dans cet exemple, la commande `ls /nonexistent` échoue, ce qui entraîne la levée de l'exception `CalledProcessError`. L'attribut `e.output` contient la sortie du processus, qui peut être affichée.

Exemple d'exécution de commandes shell complexes

Il est possible d'exécuter des commandes shell plus complexes en passant des chaînes de caractères représentant des commandes à `subprocess.run()` ou `subprocess.Popen()`.

```
import subprocess

# Ex cution d'une commande shell complexe
result = subprocess.run('echo "La date et l\'
    heure actuelles : $(date)"', shell=True,
    capture_output=True, text=True)

# Affichage du r sultat
print(result.stdout)
```

Ici, la commande `echo "La date et l'heure actuelles : $(date)"` est exécutée dans un shell.

L'argument `shell=True` permet d'interpréter la chaîne de commande comme une commande shell, et `capture_output=True` capture la sortie.

Résumé

La bibliothèque `subprocess` permet de lancer et de gérer des processus externes depuis Python. Elle offre plusieurs fonctions pour exécuter des commandes système, interagir avec leur entrée/sortie, et récupérer leurs résultats. Les fonctions les plus courantes sont `subprocess.run`, `subprocess.call`, et `subprocess.Popen`. L'utilisation appropriée de ces outils permet une gestion fine des processus externes et des erreurs associées.

Bibliothèque `os`

La bibliothèque `os` fournit une interface pour interagir avec le système d'exploitation. Elle permet d'effectuer des opérations sur le système de fichiers, de gérer les processus, d'interagir avec l'environnement, et bien plus encore. C'est l'une des bibliothèques standard de Python pour manipuler le système d'exploitation de manière indépendante de la plateforme.

Fonctions principales de `os`

`os` comprend un large éventail de fonctions utiles. Parmi les plus courantes :

- `os.getcwd()` : Retourne le répertoire de travail actuel (le répertoire dans lequel le script Python est exécuté).
- `os.chdir(path)` : Change le répertoire de travail actuel vers `path`.
- `os.listdir(path)` : Retourne une liste des fichiers et répertoires dans le répertoire spécifié.
- `os.path.join(path1, path2, ...)` : Joint plusieurs parties d'un chemin de fichier de manière sécurisée, indépendamment du système d'exploitation.
- `os.path.exists(path)` : Vérifie si le chemin spécifié existe.
- `os.path.isdir(path)` : Vérifie si le chemin spécifié est un répertoire.
- `os.path.isfile(path)` : Vérifie si le chemin spécifié est un fichier.
- `os.remove(path)` : Supprime le fichier spécifié.
- `os.rename(old, new)` : Renomme un fichier ou un répertoire de `old` à `new`.
- `os.mkdir(path)` : Crée un répertoire à l'emplacement spécifié.
- `os.rmdir(path)` : Supprime un répertoire vide.
- `os.environ` : Un dictionnaire représentant l'environnement d'exécution. Il peut être utilisé pour lire ou modifier les variables d'environnement.
- `os.system(command)` : Exécute une commande shell (souvent utilisée pour lancer des commandes système).
- `os.spawn()` et `os.fork()` : Fonctions avancées pour manipuler des processus sous Unix.

Exemple d'utilisation de `os.getcwd`

`os.getcwd()` permet de connaître le répertoire de travail actuel.

```
1 import os
2
3 # Afficher le r pertoire de travail actuel
4 current_directory = os.getcwd()
5 print(f"R pertoire actuel : {
      current_directory}")
```

Dans cet exemple, la fonction `os.getcwd()` est utilisée pour obtenir le répertoire dans lequel le script Python est exécuté.

Exemple d'utilisation de `os.chdir`

`os.chdir(path)` permet de changer le répertoire de travail actuel.

```
1 import os
2
3 # Changer le r pertoire de travail
4 os.chdir('/path/to/directory')
5
6 # Afficher le nouveau r pertoire de travail
7 print(f"Nouveau r pertoire : {os.getcwd()}")
```

Ici, la fonction `os.chdir('/path/to/directory')` change le répertoire courant en `'/path/to/directory'`.

Exemple d'utilisation de `os.path.join`

`os.path.join(path1, path2, ...)` permet de joindre plusieurs éléments d'un chemin de fichier de manière sécurisée.

```
1 import os
2
3 # Joindre plusieurs parties d'un chemin de
      mani re s curis e
4 file_path = os.path.join('folder', 'subfolder'
      , 'file.txt')
5
6 # Afficher le chemin complet
7 print(f"Chemin complet : {file_path}")
```

Dans cet exemple, `os.path.join` assemble les parties `'folder'`, `'subfolder'`, et `'file.txt'` en un chemin complet. Cette fonction garantit que les séparateurs de chemins sont utilisés correctement selon le système d'exploitation (par exemple, '/' sous Unix et '
' sous Windows).

Exemple d'utilisation de `os.listdir`

`os.listdir(path)` renvoie une liste des fichiers et répertoires dans un répertoire donné.

```
import os

# Lister les fichiers dans un r pertoire
files = os.listdir('/path/to/directory')

# Afficher la liste des fichiers
print("Fichiers et r pertoires :")
for file in files:
    print(file)
```

Ici, la fonction `os.listdir` liste tous les fichiers et répertoires dans le répertoire spécifié.

Exemple d'utilisation de `os.remove`

`os.remove(path)` permet de supprimer un fichier spécifié.

```
import os

# Supprimer un fichier
file_path = '/path/to/file.txt'
os.remove(file_path)

print(f"Le fichier {file_path} a   t
    supprim .")
```

Cet exemple montre comment supprimer un fichier en utilisant `os.remove()`. Si le fichier n'existe pas, une exception `FileNotFoundError` sera levée.

Exemple d'utilisation de `os.environ`

`os.environ` permet de lire ou modifier les variables d'environnement.

```
import os

# Lire une variable d'environnement
home_directory = os.environ.get('HOME', '
    Variable non d finie')
print(f"R pertoire personnel : {
    home_directory}")
```

Dans cet exemple, nous lisons la variable d'environnement `HOME`, qui contient le répertoire personnel de l'utilisateur. Si la variable n'est pas définie, la valeur par défaut "Variable non définie" est utilisée.

Exemple d'utilisation de `os.system`

`os.system(command)` permet d'exécuter une commande shell.

```
import os

# Ex cuter une commande syst me
os.system('echo Hello, World!')
```

Cet exemple montre comment exécuter une commande shell en utilisant `os.system()`. Ici, la commande `echo Hello, World!` affiche le message "Hello, World!" dans le terminal.

Résumé

La bibliothèque `os` est un outil puissant pour interagir avec le système d'exploitation dans Python. Elle permet de gérer les fichiers, les répertoires, les processus et l'environnement. Parmi les fonctions les plus courantes, on trouve `os.getcwd()`, `os.chdir()`, `os.remove()`, et `os.environ`. Elle est indispensable pour effectuer des opérations systèmes de manière portable entre différentes plateformes.

Bibliothèque `pathlib`

La bibliothèque `pathlib` est une bibliothèque standard de Python qui permet de manipuler les chemins de fichiers de manière orientée objet. Elle fournit une interface moderne et plus lisible par rapport à `os.path` pour effectuer des opérations sur les chemins de fichiers et de répertoires.

Fonctions principales de `pathlib`

`pathlib` offre une classe principale `Path`, qui permet de gérer les chemins de manière portable et simple. Parmi les fonctionnalités principales de `pathlib`, on trouve :

- `Path.cwd()` : Retourne le répertoire de travail actuel sous forme de `Path`.
- `Path.home()` : Retourne le répertoire personnel de l'utilisateur.
- `Path.exists()` : Vérifie si un fichier ou un répertoire existe.
- `Path.is_file()` : Vérifie si le chemin correspond à un fichier.
- `Path.is_dir()` : Vérifie si le chemin correspond à un répertoire.
- `Path.mkdir()` : Crée un répertoire.
- `Path.rmdir()` : Supprime un répertoire vide.
- `Path.rename()` : Renomme un fichier ou un répertoire.
- `Path.unlink()` : Supprime un fichier.
- `Path.glob()` : Permet de rechercher des fichiers et répertoires correspondant à un motif donné.
- `Path.joinpath()` : Joins des parties d'un chemin de manière sécurisée.
- `Path.resolve()` : Résout un chemin absolu à partir d'un chemin relatif.

Exemple d'utilisation de `Path.cwd`

`Path.cwd()` permet d'obtenir le répertoire de travail actuel sous forme de `Path`.

```
from pathlib import Path

# Afficher le r pertoire de travail actuel
current_directory = Path.cwd()
print(f"R pertoire actuel : {
    current_directory}")
```

Cet exemple montre comment utiliser `Path.cwd()` pour obtenir le répertoire de travail actuel, qui est retourné sous forme d'objet `Path`.

Exemple d'utilisation de `Path.exists`

`Path.exists()` permet de vérifier si un fichier ou un répertoire existe.

```
from pathlib import Path

# V rifier si un fichier existe
file_path = Path('/path/to/file.txt')

if file_path.exists():
    print(f"Le fichier {file_path} existe.")
else:
    print(f"Le fichier {file_path} n'existe
        pas.")
```

Dans cet exemple, `Path.exists()` est utilisé pour vérifier l'existence d'un fichier spécifié.

Exemple d'utilisation de `Path.is_file`

`Path.is_file()` permet de vérifier si un chemin est un fichier.

```
from pathlib import Path

# V rifier si le chemin est un fichier
file_path = Path('/path/to/file.txt')

if file_path.is_file():
    print(f"{file_path} est un fichier.")
else:
    print(f"{file_path} n'est pas un fichier."
        )
```

Cet exemple montre l'utilisation de `Path.is_file()` pour vérifier si le chemin donné correspond à un fichier.

Exemple d'utilisation de `Path.is_dir`

`Path.is_dir()` permet de vérifier si un chemin est un répertoire.

```
from pathlib import Path

# V rifier si le chemin est un r pertoire
dir_path = Path('/path/to/directory')

if dir_path.is_dir():
    print(f"{dir_path} est un r pertoire.")
else:
    print(f"{dir_path} n'est pas un
        r pertoire.")
```

Ici, nous utilisons `Path.is_dir()` pour tester si un chemin est un répertoire.

Exemple d'utilisation de `Path.mkdir`

`Path.mkdir()` permet de créer un répertoire.

```
from pathlib import Path

# Cr er un r pertoire
dir_path = Path('/path/to/new_directory')
dir_path.mkdir(parents=True, exist_ok=True)

print(f"Le r pertoire {dir_path} a   t
    cr   .")
```

Cet exemple montre comment créer un répertoire avec `Path.mkdir()`. Le paramètre `parents=True` permet de créer également tous les répertoires parents nécessaires, et `exist_ok=True` évite de lever une erreur si le répertoire existe déjà.

Exemple d'utilisation de `Path.rename`

`Path.rename()` permet de renommer un fichier ou un répertoire.

```
from pathlib import Path

# Renommer un fichier
old_path = Path('/path/to/old_file.txt')
new_path = Path('/path/to/new_file.txt')
old_path.rename(new_path)

print(f"Le fichier a   t   renomm  en {
    new_path}.")
```

Ici, `Path.rename()` est utilisé pour renommer un fichier. Si un fichier avec le nouveau nom existe déjà, une exception sera levée.

Exemple d'utilisation de `Path.glob`

`Path.glob()` permet de rechercher des fichiers dans un répertoire correspondant à un motif donné.

```
from pathlib import Path

# Rechercher tous les fichiers \texttt{*.txt}
    dans un r pertoire
dir_path = Path('/path/to/directory')

for txt_file in dir_path.glob('*.txt'):
    print(txt_file)
```

Dans cet exemple, nous utilisons `Path.glob()` pour lister tous les fichiers avec l'extension `.txt` dans un répertoire spécifié.

Exemple d'utilisation de `Path.joinpath`

`Path.joinpath()` permet de joindre plusieurs parties d'un chemin de manière sécurisée.

```
from pathlib import Path

# Joindre plusieurs parties d'un chemin
path = Path('/path/to').joinpath('subfolder',
    'file.txt')

print(f"Chemin complet : {path}")
```

Dans cet exemple, `Path.joinpath()` assemble plusieurs parties d'un chemin pour créer un chemin complet de manière portable.

Résumé

La bibliothèque `pathlib` est une alternative moderne à `os.path` pour gérer les chemins de fichiers en Python. Elle fournit une interface orientée objet qui facilite la manipulation des chemins de manière claire et intuitive. Parmi ses fonctionnalités, on trouve la création de répertoires, la recherche de fichiers, et la vérification de l'existence de chemins. Elle est particulièrement utile pour manipuler des chemins de manière portable et pour travailler avec des fichiers de manière plus élégante.

Bibliothèque `tqdm`

La bibliothèque `tqdm` est utilisée pour afficher des barres de progression dans les boucles, facilitant ainsi le suivi de l'avancement des traitements longs en Python. `tqdm` est très utile pour visualiser le progrès d'un programme, notamment lors de traitements sur de grandes quantités de données.

Installation

Pour installer `tqdm`, vous pouvez utiliser `pip` :

```
pip install tqdm
```

Utilisation de `tqdm`

`tqdm` peut être utilisé pour ajouter facilement une barre de progression à une boucle. Voici un exemple simple :

```
from tqdm import tqdm
import time

# Exemple d'une boucle avec barre de
    progression
for i in tqdm(range(100)):
    time.sleep(0.1)  # Simulation d'un
        traitement
```

Dans cet exemple, une barre de progression est ajoutée à une boucle allant de 0 à 99. La fonction `time.sleep(0.1)` simule un traitement prenant du temps.

Personnalisation de la barre de progression

Vous pouvez personnaliser le comportement de la barre de progression en utilisant des arguments supplémentaires. Par exemple, pour changer le message affiché et ajuster la vitesse de la barre, vous pouvez utiliser les options suivantes :

```
from tqdm import tqdm
import time

# Exemple avec personnalisation de la barre de
    progression
for i in tqdm(range(100), desc="Traitement des
    donn es", ncols=100, unit="it ration"):
    time.sleep(0.1)  # Simulation d'un
        traitement
```

Voici les options utilisées dans cet exemple :

- `desc="Traitement des données"` : Définit le texte de description affiché à gauche de la barre.
- `ncols=100` : Fixe la largeur de la barre de progression.
- `unit="itération"` : Définit l'unité affichée à côté du compteur.

Utilisation dans les boucles imbriquées

`tqdm` peut également être utilisé dans des boucles imbriquées pour suivre l'avancement de chaque niveau de la boucle.

```
from tqdm import tqdm
import time

# Exemple avec boucles imbriqu es
for i in tqdm(range(5), desc="Boucle 1", unit=
    "it ration"):
    for j in tqdm(range(10), desc="Boucle 2",
        unit="sous-it ration", leave=False):
        time.sleep(0.1)  # Simulation d'un
            traitement
```

Dans cet exemple, nous avons deux boucles imbriquées, et chaque boucle a sa propre barre de progression. L'argument `leave=False` empêche la barre de la deuxième boucle de rester après chaque itération.

Affichage d'une barre de progression pour un itérable personnalisé

Vous pouvez également utiliser `tqdm` avec des itérables personnalisés. Il vous suffit de les entourer avec la fonction `tqdm()`.

```
from tqdm import tqdm
import time

# Exemple avec un it rable personnalis
def ma_fonction():
    for i in range(100):
        yield i
        time.sleep(0.1)

for i in tqdm(ma_fonction(), desc="It ration
    fonction", unit="  lment  "):
    pass  # Simulation d'un traitement
```

Dans cet exemple, la fonction `ma_fonction` génère des éléments de manière paresseuse, et `tqdm()` est utilisé pour afficher une barre de progression lors de l'itération sur cet itérable.

Utilisation avec `pandas`

`tqdm` peut également être intégré avec `pandas` pour afficher une barre de progression lors de l'itération sur un `DataFrame` ou une `Series`.

```
from tqdm import tqdm
import pandas as pd

# Exemple avec pandas
df = pd.DataFrame({'A': range(100)})

# Utilisation de tqdm avec pandas
for index, row in tqdm(df.iterrows(), total=df
    .shape[0], desc="Traitement DataFrame"):
    pass  # Simulation d'un traitement
```

Dans cet exemple, nous utilisons `tqdm` avec la méthode `iterrows()` pour afficher une barre de progression lors de l'itération sur les lignes d'un `DataFrame`.

Utilisation avancée : `tqdm` avec `asyncio`

Il est également possible d'utiliser `tqdm` avec des tâches asynchrones dans `asyncio`. Cela permet de suivre l'avancement des tâches asynchrones de manière efficace.

```
from tqdm import tqdm
import asyncio

# Exemple avec asyncio
async def ma_tache(i):
    await asyncio.sleep(0.1)
    return i

async def main():
    tasks = [ma_tache(i) for i in range(100)]

    # Utilisation de tqdm avec asyncio
    for result in tqdm(await asyncio.gather(*
        tasks), total=100, desc="T ches
        asynchrones"):
        pass  # Simulation d'un traitement

asyncio.run(main())
```

Dans cet exemple, `tqdm` est utilisé pour suivre l'avancement de 100 tâches asynchrones exécutées avec `asyncio.gather()`.

Résumé

La bibliothèque `tqdm` est un outil simple et puissant pour ajouter des barres de progression dans les boucles Python. Elle permet de suivre l'avancement des traitements longs de manière visuelle et claire. `tqdm` est très facile à intégrer dans des boucles et peut être personnalisé pour afficher des informations utiles telles que le texte de description, la largeur de la barre, et l'unité de mesure. De plus, elle offre des fonctionnalités avancées pour travailler avec des itérables personnalisés, des `DataFrame` `pandas`, et des tâches `asyncio`.

Bibliothèque `tqdm.asyncio`

La bibliothèque `tqdm.asyncio` fait partie de `tqdm` et permet d'utiliser des barres de progression dans des programmes asynchrones utilisant `asyncio`. Elle permet de suivre facilement l'avancement des tâches asynchrones.

Installation

Si vous n'avez pas encore installé `tqdm`, vous pouvez l'installer avec `pip` :

```
pip install tqdm
```

Utilisation de `tqdm.asyncio`

`tqdm.asyncio` fonctionne de manière similaire à `tqdm`, mais est optimisé pour les tâches asynchrones. Voici un exemple d'utilisation avec des tâches `asyncio` :

```
from tqdm.asyncio import tqdm as tqdm_async
import asyncio

# Exemple avec asyncio
async def ma_tache(i):
    await asyncio.sleep(0.1)
    return i

async def main():
    tasks = [ma_tache(i) for i in range(100)]

    # Utilisation de tqdm_async pour afficher
        la barre de progression des t ches
        asynchrones
    for result in tqdm_async(await asyncio.
        gather(*tasks), total=100, desc="
        T ches asynchrones"):
        pass  # Simulation d'un traitement

asyncio.run(main())
```

Dans cet exemple, nous avons 100 tâches asynchrones qui sont exécutées simultanément grâce à `asyncio.gather()`. La barre de progression est gérée par `tqdm.asyncio`, ce qui permet de suivre l'avancement des tâches.

Personnalisation de la barre de progression

`tqdm.asyncio` permet de personnaliser la barre de progression de manière similaire à `tqdm`. Vous pouvez définir des options comme le texte de description ou la largeur de la barre :

```
from tqdm.asyncio import tqdm as tqdm_async
import asyncio

async def ma_tache(i):
    await asyncio.sleep(0.1)
    return i

async def main():
    tasks = [ma_tache(i) for i in range(100)]

    # Personnalisation de la barre de
        progression
    for result in tqdm_async(await asyncio.
        gather(*tasks), total=100, desc="
        Traitement des t ches", ncols=100,
        unit="t che"):
        pass  # Simulation d'un traitement

asyncio.run(main())
```

Les options suivantes ont été utilisées :

- `desc="Traitement des tâches"` : Texte de description affiché à gauche de la barre de progression.
- `ncols=100` : Largeur de la barre de progression.
- `unit="tâche"` : Unité affichée à côté du compteur.

Résumé

La bibliothèque `tqdm.asyncio` est une extension de `tqdm` qui facilite l'affichage de barres de progression dans les programmes utilisant `asyncio`. Elle permet de suivre facilement l'avancement des tâches asynchrones avec un minimum d'effort. Les barres de progression sont

entièrement personnalisables et peuvent être utilisées de manière fluide avec `asyncio.gather()` ou d'autres structures asynchrones.

Bibliothèque `tqdm.asyncio`

La bibliothèque `tqdm.asyncio` fait partie de `tqdm` et permet d'afficher des barres de progression pour les tâches asynchrones avec `asyncio`. Elle est utilisée avec la fonction `tqdm_asyncio` pour gérer l'affichage de la progression dans des environnements asynchrones.

Installation

Si `tqdm` n'est pas encore installé, vous pouvez l'installer via `pip` :

```
pip install tqdm
```

Utilisation de `tqdm.asyncio`

`tqdm.asyncio` offre une version asynchrone de la barre de progression, qui s'intègre bien avec `asyncio`. L'importation `tqdm_asyncio` permet de suivre les progrès des tâches `async` de manière efficace. Voici un exemple d'utilisation :

```
from tqdm.asyncio import tqdm_asyncio
import asyncio

async def ma_tache(i):
    await asyncio.sleep(0.1)
    return i

async def main():
    tasks = [ma_tache(i) for i in range(100)]

    # Utilisation de tqdm_asyncio pour la barre de progression asynchrone
    for result in tqdm_asyncio(await asyncio.gather(*tasks), total=100, desc="Traitement des t ches"):
        pass  # Traitement simul

asyncio.run(main())
```

Dans cet exemple, nous avons une fonction `ma_tache` qui simule une tâche asynchrone, et une barre de progression qui est affichée pendant l'exécution des tâches avec `tqdm_asyncio`. La méthode `asyncio.gather()` permet de gérer simultanément plusieurs tâches.

Personnalisation de la barre de progression

`tqdm.asyncio` permet de personnaliser la barre de progression comme avec la version classique de `tqdm`. Voici un exemple de personnalisation :

```
from tqdm.asyncio import tqdm_asyncio
import asyncio

async def ma_tache(i):
    await asyncio.sleep(0.1)
    return i

async def main():
    tasks = [ma_tache(i) for i in range(100)]

    # Personnalisation de la barre de progression
    for result in tqdm_asyncio(await asyncio.gather(*tasks), total=100, desc="Ex cution des t ches", ncols=80, unit="t che"):
        pass  # Traitement simul

asyncio.run(main())
```

Les options suivantes sont utilisées :

- `desc="Exécution des tâches"` : Texte descriptif avant la barre de progression.
- `ncols=80` : Largeur de la barre de progression.
- `unit="tâche"` : Unité de la progression.

Résumé

La bibliothèque `tqdm.asyncio` avec `tqdm_asyncio` permet d'ajouter une barre de progression aux tâches asynchrones dans `asyncio`. Elle s'intègre bien avec les tâches exécutées de manière simultanée et offre plusieurs options de personnalisation. Cela permet aux développeurs de suivre l'avancement de l'exécution des tâches asynchrones avec facilité et efficacité.

d

Explication du code HTML

Le code HTML présenté est destiné à la création d'un tableau de bord pour la visualisation des vulnérabilités CVE, avec des graphiques interactifs. Voici les principales sections du code et leur explication.

Déclaration et Métadonnées

Le code commence par la déclaration de type de document `<!DOCTYPE html>`, suivie de la balise `<html>` avec l'attribut `lang="fr"` pour spécifier que le contenu est en français. Dans la section `<head>`, on retrouve plusieurs éléments :

- `<meta charset="UTF-8">` : définit l'encodage des caractères.
- `<meta name="viewport" content="width=device-width, initial-scale=1.0">` : rend le site responsive.
- `<title>Dashboard CVE</title>` : définit le titre de la page.
- Inclusion de fichiers CSS à partir de `url_for` pour lier les styles de la page et de la table.
- Lien vers Google Fonts pour inclure des polices personnalisées (Manrope et Inter).

Contenu Principal

La section `<body>` contient le contenu visuel de la page.

Chargement

Un div `<div id="loading" class="loading">` est utilisé pour afficher une animation de chargement avant que la page ne soit complètement rendue, avec une `div` interne de classe `loading_spinner` pour l'animation.

En-tête

Dans l'en-tête (`<header>`), une section `<div class="header_brand">` contient deux images : une pour le logo et une autre pour le texte associé. Un autre div est utilisé pour appliquer un effet de flou de fond à l'en-tête avec des styles CSS spécifiques pour obtenir un dégradé dynamique.

Tableau de données

Le tableau est conçu pour afficher des informations sur les vulnérabilités CVE. Il inclut un champ de recherche et une pagination.

- `<input type="text" id="table_searchContainerInput" class="table_searchContainerInput" placeholder="Recherche">` permet à l'utilisateur de rechercher dans le tableau.
- Le tableau `<table class="table_frame">` affiche plusieurs colonnes : CVE, Type, Titre, Date, Score CVSS.

- Les contrôles de pagination permettent de choisir combien d'éléments afficher par page et d'aller à la première, précédente, suivante ou dernière page.

Graphiques

Plusieurs graphiques sont inclus pour la visualisation des données, chacun placé dans un div avec une classe `chart_wrapper` :

- Un histogramme de CVSS est affiché dans `<div id="cvssHistogram" class="chart_wrapper"></div>`.
- Un graphique circulaire pour la distribution CWE dans `<div id="cwePieChart" class="chart_wrapper"></div>`.
- Un graphique linéaire pour l'EPSS dans `<div id="epssLineChart" class="chart_wrapper"></div>`.
- D'autres graphiques incluent la distribution des scores CVSS, l'analyse détaillée par type CWE, et l'évolution cumulative des vulnérabilités.

Modalités et Détails

Un modal est prévu pour afficher des détails sur une vulnérabilité spécifique. Il est contenu dans `<div id="detailModal" class="modal">`, avec une section `modal-content` pour afficher le titre et le corps du modal.

Scripts

Les scripts sont inclus à la fin du fichier `<body>` pour charger les bibliothèques JavaScript nécessaires à la page :

- `<script src=" url_for('static', filename='js/index.js') "></script>` pour charger le fichier JavaScript principal du projet.
- Liens vers les versions de développement de React, ReactDOM et Recharts depuis un CDN (universal package distribution).
- `<script src=" url_for('static', filename='dist/bundle.js') "></script>` pour le script bundlé avec les éléments front-end.

Ces scripts permettent de rendre les graphiques interactifs et de gérer la logique d'affichage dynamique des données, ainsi que les interactions avec le modal.

Conclusion

Le code HTML présenté fournit une interface pour visualiser et analyser les vulnérabilités CVE à travers un tableau interactif, des graphiques et des modaux. Il utilise diverses technologies comme `CSS` pour le style, `JavaScript` pour l'interactivité, et `React` pour le rendu dynamique des composants. La page est conçue pour être responsive et facile à utiliser, avec des contrôles de pagination, un système de recherche, et des graphiques détaillés.