Documentation du Code Python - Client

Contents

1 Importation des modules

Le script commence par l'importation de plusieurs modules essentiels à son fonctionnement. Ces modules permettent de gérer les délais, manipuler les fichiers, stocker les données et générer des graphiques.

- **time** : Permet de gérer les temporisations et les délais d'exécution dans le programme.
- **json** : Utilisé pour stocker et manipuler les données au format JSON, notamment pour le stockage local.
- **requests** : Utilisé pour envoyer des requêtes HTTP au serveur, notamment l'envoi des données collectées.
- **datetime, timedelta** : Fournit des outils pour manipuler les dates et gérer les intervalles de temps, ce qui est utile pour l'archivage et le suivi temporel.
- **pandas** : Librairie puissante pour la manipulation et l'analyse de données, utilisée ici pour archiver les données en fichiers CSV et Excel.
- **sys** : Utilisé pour reconfigurer l'encodage de la sortie standard en UTF-8, afin d'éviter les problèmes d'affichage des caractères spéciaux.
- **os** : Permet de manipuler les fichiers et les dossiers sur le système, notamment pour créer des répertoires et déplacer des fichiers d'archives.
- **matplotlib.pyplot** : Utilisé pour générer des graphiques permettant de visualiser l'évolution des données collectées.

2 Configuration des paramètres

Le script définit plusieurs paramètres globaux afin de gérer les chemins de stockage, les intervalles de collecte, d'envoi et d'archivage des données.

- **SERVER_URL** : Définition de l'URL du serveur Flask qui recevra les données via une requête HTTP POST.
- **LOCAL_STORAGE_FILE** : Nom du fichier JSON où sont stockées temporairement les données collectées avant envoi.
- **ENVOI_INTERVALLE** : Intervalle (en secondes) entre chaque tentative d'envoi des données au serveur.
- **ARCHIVE_INTERVALLE** : Intervalle (en secondes) entre chaque archivage des données en fichiers CSV et Excel.
- **ARCHIVE_DIRECTORY** : Chemin du répertoire où seront stockées les archives des données collectées.

- **BACKUP_DIRECTORY** : Chemin du répertoire où seront déplacées les anciennes archives avant suppression.
- **TEMP_ARCHIVE_FILE** : Nom du fichier temporaire servant d'intermédiaire pour l'archivage des données.
- **NETTOYAGE_INTERVALLE** : Fréquence (en secondes) à laquelle les anciennes archives seront déplacées vers le dossier de sauvegarde.
- **sys.stdout.reconfigure(encoding='utf-8')** : Configure l'encodage en UTF-8 afin d'éviter les erreurs d'affichage des caractères spéciaux lors des impressions en console.

3 Génération des données de capteurs

La fonction `generer_donnees()` est responsable de la génération des données de capteurs simulées. Elle crée un dictionnaire contenant une mesure de température et d'humidité avec un horodatage.

Explication du fonctionnement :

- La date et l'heure actuelles sont récupérées et formatées sous la forme `[jour/mois/année heure:minute:seconde]`.
- Les valeurs de température et d'humidité sont définies (dans cet exemple, elles sont fixes : 22°C et 50% d'humidité).
- Les données sont encapsulées dans un dictionnaire Python.
- La fonction retourne ces données sous forme de dictionnaire.

Code correspondant :

```
def generer_donnees():
    date_formatee = datetime.now().strftime("[%d/%b/%Y %H:%M:%S]")
    data = {
        "temperature": 22,
        "humidity": 50,
        "timestamp": date_formatee
    }
    return data
```

Ainsi, chaque appel de cette fonction retournera un dictionnaire contenant une entrée de données capteur avec une température, une humidité et un horodatage mis à jour.

4 Stockage des données localement

La fonction `stocker_localement()` permet de sauvegarder les données collectées dans un fichier JSON sur le disque local. Cela garantit que les informations ne sont pas perdues si l'envoi des données au serveur échoue.

Explication du fonctionnement :

- Vérifie si le fichier `LOCAL_STORAGE_FILE` existe déjà.
- Si le fichier existe, charge les données JSON stockées.
- Si le fichier n'existe pas, initialise une liste vide pour stocker les nouvelles données.
- Ajoute les nouvelles données collectées à la liste existante.
- Réécrit le fichier avec les données mises à jour pour garantir une conservation correcte.

Détails supplémentaires :

- `try / except` : Capture l'erreur si le fichier n'existe pas, évitant ainsi une exception qui bloquerait le programme.
- `json.load(fichier)` : Convertit les données du fichier JSON en une liste Python.
- `data.append(donnees)` : Ajoute les nouvelles données collectées à la liste.
- `json.dump(data, fichier, indent=4)` : Réécrit le fichier JSON avec les données mises à jour, en utilisant une indentation de 4 espaces pour une meilleure lisibilité.

Code correspondant :

```
def stocker_localement(donnees):
    try:
        with open(LOCAL_STORAGE_FILE, "r") as fichier:
            data = json.load(fichier)
    except FileNotFoundError:
        data = []
    data.append(donnees)
    with open(LOCAL_STORAGE_FILE, "w") as fichier:
        json.dump(data, fichier, indent=4)
```

Cette approche assure que chaque nouvelle donnée collectée est ajoutée à l'historique sans écraser les anciennes valeurs.

5 Envoi des données au serveur

La fonction `envoyer_donnees()` est responsable de l'envoi des données collectées vers le serveur via une requête HTTP POST. Si l'envoi échoue, les données non envoyées sont sauvegardées localement pour être retentées ultérieurement.

Explication du fonctionnement :

- Vérifie si le fichier `LOCAL_STORAGE_FILE` existe et contient des données.
- Si le fichier n'existe pas ou est vide, affiche un message indiquant qu'il n'y a aucune donnée à envoyer.
- Sauvegarde une copie temporaire des données dans `TEMP_ARCHIVE_FILE` avant de tenter l'envoi.
- Parcourt chaque entrée des données stockées :
  - Envoie les données au serveur en effectuant une requête HTTP POST.
  - Si l'envoi réussit (`status code 200`), un message de confirmation est affiché.
  - Si l'envoi échoue (erreur réseau ou autre `status code`), les données sont ajoutées à une liste `data_non_envoyee`.
- Après l'envoi, met à jour le fichier local en sauvegardant uniquement les données non envoyées pour éviter une perte d'informations.
- Affiche le nombre de données non envoyées qui ont été conservées.

Détails supplémentaires :

- `try / except` : Capture les erreurs en cas de fichier manquant ou de JSON corrompu.
- `requests.post(SERVER_URL, json=entry)` : Envoie chaque donnée en tant que requête POST au serveur.
- `data_non_envoyee` : Liste qui stocke les entrées dont l'envoi a échoué afin de ne pas les perdre.
- `json.dump(data_non_envoyee, fichier, indent=4)` : Met à jour le fichier JSON avec uniquement les données non envoyées.

Code correspondant :

```
def envoyer_donnees():
    try:
        with open(LOCAL_STORAGE_FILE, "r") as fichier:
            data = json.load(fichier)
```

```
except (FileNotFoundError, json.JSONDecodeError):
    print("Aucune donnée à envoyer.")
    return
with open(TEMP_ARCHIVE_FILE, "w") as temp_file:
    json.dump(data, temp_file, indent=4)
data_non_envoyee = []
for entry in data:
    try:
        response = requests.post(SERVER_URL, json=entry)
        if response.status_code == 200:
            print(f"Données envoyées avec succès : {entry}")
        else:
            print(f"Erreur d'envoi : {response.status_code}")
            data_non_envoyee.append(entry)
    except Exception as e:
        print(f"Erreur de connexion : {e}")
        data_non_envoyee.append(entry)
with open(LOCAL_STORAGE_FILE, "w") as fichier:
    json.dump(data_non_envoyee, fichier, indent=4)
print(f"{len(data_non_envoyee)} données non envoyées sauvegardées localement.")
```

Cette approche assure que toutes les données collectées finissent par être envoyées, même en cas de problèmes réseau temporaires.

6 Archivage des données

La fonction `archiver_donnees()` est responsable de la sauvegarde périodique des données collectées en les enregistrant sous forme de fichiers CSV et Excel. Cet archivage permet de conserver un historique des mesures et de faciliter leur analyse ultérieure.

Explication du fonctionnement :

- Vérifie si le fichier `TEMP_ARCHIVE_FILE` existe et contient des données.
- Si le fichier est introuvable ou corrompu, affiche un message indiquant qu'il n'y a aucune donnée à archiver.
- Si aucune nouvelle donnée n'est disponible, la fonction s'arrête immédiatement.
- Convertit les données JSON en un **DataFrame** `pandas`, une structure de données tabulaire facilitant l'exportation vers des fichiers CSV et Excel.
- Génère des noms de fichiers basés sur la date et l'heure actuelles pour identifier facilement les archives.

- Enregistre les données sous deux formats :
  - **Excel** (`.xlsx`) pour une exploitation dans des tableurs comme Microsoft Excel.
  - **CSV** (`.csv`) pour une compatibilité avec divers outils d'analyse.
- Affiche un message confirmant l'archivage réussi.
- Vide le fichier temporaire pour éviter des doublons lors du prochain archivage.

Détails supplémentaires :

- `try / except` : Capture les erreurs en cas de fichier manquant ou de JSON corrompu.
- `pandas.DataFrame` : Permet de manipuler les données sous forme de tableau avant exportation.
- `date_now.strftime()` : Génére un horodatage unique pour chaque fichier d'archive.
- `df.to_excel()` et `df.to_csv()` : Convertissent et enregistrent les données dans les formats Excel et CSV.
- `open(TEMP_ARCHIVE_FILE, "w").write("[]")` : Réinitialise le fichier temporaire après archivage.

Code correspondant :

```
def archiver_donnees():
    try:
        with open(TEMP_ARCHIVE_FILE, "r") as fichier:
            data = json.load(fichier)
    except (FileNotFoundError, json.JSONDecodeError):
        print("Aucune donnée à archiver.")
        return
    if not data:
        print("Aucune nouvelle donnée à archiver.")
        return
    df = pd.DataFrame(data)
    date_now = datetime.now()
    horodatage = date_now.strftime("%Y-%m-%d_%H-%M")
    heure_archive = date_now.strftime("%Hh%M")
    fichier_excel = f"{ARCHIVE_DIRECTORY}archive_{horodatage}_heure_{heure_archive}.xlsx"
    fichier_csv = f"{ARCHIVE_DIRECTORY}archive_{horodatage}_heure_{heure_archive}.csv"
    df.to_excel(fichier_excel, index=False, engine='openpyxl')
    df.to_csv(fichier_csv, index=False)
    print(f"Fichiers archivés : {fichier_excel} et {fichier_csv}")
    open(TEMP_ARCHIVE_FILE, "w").write("[]")
```

Cette approche assure une conservation organisée et efficace des données collectées, facilitant leur analyse ultérieure.

7 Nettoyage des anciennes archives

La fonction `nettoyer_archives()` permet de déplacer les anciens fichiers d'archives vers un dossier de sauvegarde dédié. Cela évite l'encombrement du dossier principal et facilite l'organisation des fichiers archivés.

Explication du fonctionnement :

- Vérifie si le dossier de sauvegarde (`BACKUP_DIRECTORY`) existe et le crée s'il est absent.
- Génère un graphique basé sur les données archivées avant le nettoyage pour conserver une trace visuelle.
- Parcourt tous les fichiers présents dans le dossier d'archives (`ARCHIVE_DIRECTORY`).
- Vérifie si chaque fichier est un fichier **CSV** ou **Excel**, indiquant une archive à déplacer.
- Déplace ces fichiers vers le dossier de sauvegarde `BACKUP_DIRECTORY` pour une conservation ordonnée.
- Gère les erreurs éventuelles, notamment si un fichier est en cours d'utilisation et ne peut pas être déplacé.

Détails supplémentaires :

- `os.makedirs(BACKUP_DIRECTORY, exist_ok=True)` : Vérifie si le dossier de sauvegarde existe, sinon le crée.
- `generer_graphique()` : Génère un graphique basé sur les données archivées avant le nettoyage.
- `os.listdir(ARCHIVE_DIRECTORY)` : Liste tous les fichiers présents dans le dossier d'archives.
- `fichier.endswith('.csv') or fichier.endswith('.xlsx')` : Sélectionne uniquement les fichiers de type archive.
- `os.rename(chemin_fichier, chemin_sauvegarde)` : Déplace les fichiers vers le dossier de sauvegarde.
- `except PermissionError` : Capture les erreurs si un fichier est en cours d'utilisation et ne peut pas être déplacé.

Code correspondant :

```
def nettoyer_archives():
    os.makedirs(BACKUP_DIRECTORY, exist_ok=True)
    generer_graphique()
    for fichier in os.listdir(ARCHIVE_DIRECTORY):
        chemin_fichier = os.path.join(ARCHIVE_DIRECTORY, fichier)
        chemin_sauvegarde = os.path.join(BACKUP_DIRECTORY, fichier)
        if os.path.isfile(chemin_fichier):
            try:
                if fichier.endswith('.csv') or fichier.endswith('.xlsx'):
                    os.rename(chemin_fichier, chemin_sauvegarde)
                    print(f"Fichier sauvegardé : {chemin_fichier} -> {chemin_sauvegarde}")
            except PermissionError:
                print(f"Impossible de déplacer (fichier utilisé) : {chemin_fichier}")
```

Cette approche garantit une gestion efficace des archives en évitant l'encombrement du dossier principal tout en conservant un historique des fichiers sauvegardés.

8 Collecte, stockage et envoi périodique des données

La fonction `collecter_et_envoyer()` orchestre l'ensemble du processus de collecte, de stockage, d'envoi et d'archivage des données de capteurs. Elle fonctionne en boucle infinie et exécute ces tâches à des intervalles de temps prédéfinis.

Explication du fonctionnement :

- Initialise trois variables temporelles :
  - `dernier_envoi` : Enregistre le moment du dernier envoi de données au serveur.
  - `dernier_archive` : Suit le dernier archivage de données.
  - `dernier_nettoyage` : Gère le dernier nettoyage des archives.
- Démarre une boucle infinie pour exécuter les actions périodiquement :
  - Génère une nouvelle mesure des capteurs en appelant `generer_donnees()`.
  - Stocke localement ces nouvelles données via `stocker_localement()`.
  - Vérifie si le temps écoulé depuis le dernier envoi dépasse `ENVOI_INTERVALLE`, puis envoie les données avec `envoyer_donnees()`.
  - Vérifie si l'intervalle d'archivage `ARCHIVE_INTERVALLE` est atteint et déclenche `archiver_donnees()`.
  - Vérifie si le nettoyage doit être effectué (toutes les 3 minutes) et appelle `nettoyer_archives()`.

- Attente de 5 secondes entre chaque itération pour limiter la charge CPU et éviter des exécutions trop fréquentes.

Détails supplémentaires :

- `time.time()` : Retourne l'heure actuelle en secondes depuis l'époque Unix, permettant le suivi du temps écoulé.
- `time.sleep(5)` : Introduit une pause de 5 secondes entre chaque cycle pour éviter une exécution continue qui surchargerait le système.
- `if time.time() - dernier_envoi >= ENVOI_INTERVALLE` :
  - Vérifie si suffisamment de temps s'est écoulé pour déclencher un nouvel envoi de données.
  - Utilisation du même principe pour l'archivage et le nettoyage.
- `while True` : Boucle infinie garantissant l'exécution continue du programme.

Code correspondant :

```
def collecter_et_envoyer():
    dernier_envoi = time.time()
    dernier_archive = time.time()
    dernier_nettoyage = time.time()
    while True:

        donnees = generer_donnees()
        print(f"Données collectées : {donnees}")

        stocker_localement(donnees)

        if time.time() - dernier_envoi >= ENVOI_INTERVALLE:
            print("Tentative d'envoi des données stockées...")
            envoyer_donnees()
            dernier_envoi = time.time()

        if time.time() - dernier_archive >= ARCHIVE_INTERVALLE:
            print("Archivage des données...")
            archiver_donnees()
            dernier_archive = time.time()

        if time.time() - dernier_nettoyage >= NETTOYAGE_INTERVALLE:
            print("Nettoyage des anciennes archives...")
            nettoyer_archives()
            dernier_nettoyage = time.time()
        time.sleep(5)
```

Cette approche assure que le système fonctionne en continu, collectant et envoyant périodiquement les données, tout en maintenant un archivage organisé et un nettoyage automatique.

9 Génération de graphiques pour les données archivées

La fonction `generer_graphique()` permet de visualiser les données archivées sous forme de graphique. Cette représentation facilite l'analyse des tendances de température et d'humidité au fil du temps.

Explication du fonctionnement :

- Recherche tous les fichiers CSV dans le dossier d'archives (`ARCHIVE_DIRECTORY`).
- Si aucun fichier CSV n'est trouvé, affiche un message et quitte la fonction.
- Charge le contenu de chaque fichier CSV dans un `DataFrame` **pandas**.
- Concatène tous les `DataFrame` pour regrouper toutes les données archivées en une seule structure tabulaire.
- Génère un graphique contenant :
  - Une courbe représentant l'évolution de la température.
  - Une courbe représentant l'évolution de l'humidité.
- Personnalise l'affichage du graphique :
  - Définit les labels de l'axe X (horodatage) et de l'axe Y (valeurs mesurées).
  - Ajoute une légende et un titre descriptif.
  - Applique une rotation aux labels de l'axe X pour une meilleure lisibilité.
- Enregistre le graphique sous forme d'image PNG dans le dossier d'archives.
- Ferme la figure pour libérer la mémoire.
- Affiche un message confirmant la création du graphique.

Détails supplémentaires :

- `os.listdir(ARCHIVE_DIRECTORY)` : Liste tous les fichiers dans le dossier d'archives.
- `df_list = [pd.read_csv(fichier) for fichier in fichiers_csv]` : Charge tous les fichiers CSV en tant que `DataFrame`.

- `pd.concat(df_list, ignore_index=True)` : Fusionne tous les `DataFrame` pour regrouper les données archivées.
- `plt.plot()` : Génère deux courbes (température et humidité) en fonction du temps.
- `plt.xticks(rotation=45)` : Fait pivoter les étiquettes de l'axe X pour améliorer la lisibilité.
- `plt.savefig()` : Sauvegarde le graphique sous forme de fichier image dans le dossier d'archives.
- `plt.close()` : Ferme la figure pour éviter une consommation excessive de mémoire.

Code correspondant :

```
def generer_graphique():
    fichiers_csv = [os.path.join(ARCHIVE_DIRECTORY, f)
    for f in os.listdir(ARCHIVE_DIRECTORY)
    if f.endswith('.csv')]
    if not fichiers_csv:
        print("Aucun fichier CSV disponible pour générer un graphique.")
        return

    df_list = [pd.read_csv(fichier) for fichier in fichiers_csv]
    df_combined = pd.concat(df_list, ignore_index=True)

    plt.figure(figsize=(10, 5))
    plt.plot(df_combined['timestamp'], df_combined['temperature'], label='Température')
    plt.plot(df_combined['timestamp'], df_combined['humidity'], label='Humidité')
    plt.xlabel("Horodatage")
    plt.ylabel("Valeurs")
    plt.title("Température et Humidité au cours du temps")
    plt.legend()
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f"{ARCHIVE_DIRECTORY}graphique_combined.png")
    plt.close()
    print(f"Graphique généré : {ARCHIVE_DIRECTORY}graphique_combined.png")
```

Cette approche permet de visualiser l'évolution des mesures environnementales et d'identifier des tendances ou anomalies potentielles.

10 Démarrage du client

Le script se termine par un bloc conditionnel qui permet de lancer le programme lorsqu'il est exécuté directement. Ce mécanisme garantit que le script fonctionne

comme une application autonome.

Explication du fonctionnement :

- Vérifie si le script est exécuté directement via la condition `if __name__ == "__main__"`.
- Crée le dossier d'archivage (`ARCHIVE_DIRECTORY`) s'il n'existe pas déjà.
- Affiche un message en console pour signaler le démarrage du programme.
- Lance la fonction principale `collecter_et_envoyer()` qui exécute en boucle les tâches de collecte, stockage, envoi, archivage et nettoyage des données.

Détails supplémentaires :

- `if __name__ == "__main__"` :
  - Cette condition permet d'exécuter le script uniquement s'il est lancé directement, et non s'il est importé comme un module dans un autre programme.
- `os.makedirs(ARCHIVE_DIRECTORY, exist_ok=True)` :
  - Vérifie si le dossier d'archivage existe.
  - S'il n'existe pas, le crée automatiquement.
  - `exist_ok=True` empêche les erreurs si le dossier est déjà présent.
- `print("Démarrage du client...")` :
  - Affiche un message dans la console pour informer l'utilisateur que le programme a bien été lancé.
- `collecter_et_envoyer()` :
  - Démarre la boucle principale du programme qui collecte, stocke, envoie et archive les données en continu.

Code correspondant :

```
if __name__ == "__main__":
    os.makedirs(ARCHIVE_DIRECTORY, exist_ok=True)
    print("Démarrage du client...")
    collecter_et_envoyer()
```

Cette structure permet d'assurer que le programme démarre correctement et fonctionne de manière autonome dès son exécution.