Brief_1_current_weather

1 Utilisation d'une API publique pour récupérer des données et les afficher de manière lisible

Pour récupérer des données en utilisant une API publique, vous allez utiliser l'API de "OpenWeatherMap" qui fournit des informations météorologiques.

1. **Obtenir une clé API :** Pour accéder à l'API d'OpenWeatherMap, il vous faut une clé API. Vous pouvez obtenir cette clé en vous inscrivant sur OpenWeatherMap ou bien en cliquant sur Get API key de l'offre gratuite (Free).
2. **Faire une requête à l'API :** Utilisons l'API pour obtenir les données météorologiques d'une ville. Pour ce faire, nous ferons une requête HTTP GET.
3. **Analyser la réponse de l'API :** La réponse de l'API est généralement au format JSON. Nous devons donc analyser cette réponse pour extraire les informations nécessaires.

1.0.1 Remarques

- Remplacez `'votre_cle_api'` par la clé API que vous avez obtenue.
- Vous pouvez ajuster les paramètres et le traitement de la réponse en fonction de vos besoins.

1.1 Récupération des données météorologiques

Obtention et affichage des données météorologiques actuelles pour la ville de Paris à partir de l'API OpenWeatherMap et les stocker dans un dictionnaire.

1. **Importation du module requests** :
   - `import requests` : Importation de la bibliothèque `requests` pour permettre l'envoi de requêtes HTTP.
   - `import datetime` : Importation de la bibliothèque `datetime` pour travailler avec des dates et des heures dans Python.
2. **Définition des paramètres de l'API** :
   - `base_url = "http://api.openweathermap.org/data/2.5/weather"` : URL de base pour l'API OpenWeatherMap.
   - `api_key = 'votre_cle_api'` : Clé API pour authentification (à remplacer par la vôtre).
   - `city_name = 'Paris'` : Nom de la ville pour laquelle les données seront récupérées.
3. **Paramètres de la requête** :
   - `params = { 'q': city_name, 'appid': api_key, 'units': 'metric' }` : Dictionnaire contenant les paramètres de la requête : nom de la ville, clé API, et unités métriques pour les données (température en degrés Celsius).
4. **Envoi de la requête GET** :
   - `response = requests.get(base_url, params=params)` : Envoi de la requête GET à l'API OpenWeatherMap avec les paramètres spécifiés.
5. **Vérification du statut de la réponse** :
   - `if response.status_code == 200` : Vérifie si la requête a réussi (statut 200).
6. **Traitement de la réponse JSON** :

- data = response.json() : Convertit la réponse JSON en dictionnaire Python.
- main = data['main'] : Extrait les informations principales des données (température, pression, humidité).
- weather = data['weather'][0] : Extrait la description du temps.
- current_weather = { ... } : Crée un dictionnaire contenant les données météorologiques actuelles pour la ville.

7. **Gestion des erreurs** :

- else: current_weather = {"error": "Unable to fetch data"} : Si la requête échoue, stocke un message d'erreur dans le dictionnaire.

1.2 Création d'une table current dans une base de données SQLite nommée weather.db :

Configuration de la structure de la base de données et préparation de la table weather pour stocker des données météorologiques.

1. **Importation du module sqlite3** :
   - import sqlite3 : Importation du module nécessaire pour travailler avec des bases de données SQLite.
2. **Définition du nom de la table** :
   - table_name = 'current' : Attribution d'un nom à la table.
3. **Vérification si les données météorologiques valides ont été obtenues** :
   - Vérifier si des données météorologiques valides ont été obtenues. Si c'est le cas, il procède à la création de la base de données et de la table, puis à l'insertion des données dans cette table. Sinon, il affiche un message d'erreur.
   - if 'error' not in current_weather_data:
4. **Connexion à la base de données** :
   - conn = sqlite3.connect('weather.db') : Établit une connexion à la base de données SQLite nommée weather.db. Si cette base de données n'existe pas, elle sera créée automatiquement dans le répertoire courant.
5. **Création d'un curseur** :

- cursor = conn.cursor() : Crée un curseur qui permet d'exécuter des commandes SQL sur la base de données.

6. **Création de la table current** :

- cursor.execute(''' ... ''') : Exécute une commande SQL pour créer une table current si elle n'existe pas déjà. La table contient les colonnes suivantes :
  - id : Clé primaire, de type entier, avec incrémentation automatique.
  - city : Nom de la ville, de type texte.
  - temperature : Température, de type réel (float).
  - pressure : Pression, de type entier.
  - humidity : Humidité, de type entier.
  - description : Description du temps, de type texte.

7. **Validation de la transaction** :

- conn.commit() : Valide la transaction et applique les modifications apportées (création de la table).

8. **Fermeture de la connexion** :

- conn.close() : Ferme la connexion à la base de données pour libérer les ressources et s'assurer que toutes les opérations sont terminées correctement.

La table current a été créée avec succès.

1.3Insertion des données météorologiques dans la table current

Connection à la base de données SQLite, puis insertion des données météorologiques dans la table current, puis fermeture de la connexion.

1. **Importation du module sqlite3** :
   - import sqlite3 : Importation de la bibliothèque sqlite3 pour permettre l'interaction avec les bases de données SQLite.
2. **Connexion à la base de données** :
   - conn = sqlite3.connect(db_name) : Connexion à la base de données nommée weather.db. Si cette base de données n'existe pas, elle sera créée automatiquement.
3. **Création d'un curseur** :
   - cursor = conn.cursor() : Création d'un curseur qui permet d'exécuter des commandes SQL sur la base de données.
4. **Insertion des données dans la table weather** :
   - cursor.execute(''' INSERT INTO weather (city, temperature, pressure, humidity, description, local_time) VALUES (?, ?, ?, ?, ?, ?) ''', (current_weather_data['city'], current_weather_data['temperature'], current_weather_data['pressure'], current_weather_data['humidity'], current_weather_data['weather'])) :
     - INSERT INTO table_name (city, temperature, pressure, humidity, description, local_time) VALUES (?, ?, ?, ?, ?, ?) : Commande SQL pour insérer des données dans la table current.
     - (current_weather_data['city'], current_weather_data['temperature'], current_weather_data['pressure'], current_weather_data['humidity'], current_weather_data['weather']) : Les valeurs à insérer dans la table sont fournies sous forme de tuple. L'utilisation de ? comme espace réservé et la fourniture des valeurs sous forme de tuple permet de se protéger contre les injections SQL.
5. **Validation des modifications** :
   - conn.commit() : Validation des modifications apportées à la base de données. Sans cette commande, les modifications ne seraient pas enregistrées.
6. **Fermeture de la connexion** :
   - conn.close() : Fermeture de la connexion à la base de données pour libérer les ressources.

Pour insérer les données météorologiques actuelles récupérées dans la base de données SQLite, assurez-vous que la variable db_name contient les informations récupérées précédemment (comme dans votre code de récupération des données météorologiques).

Données insérées avec succès dans la table current de la base de données SQLite weather.db.

1.4 SQLiteStudio pour la gestion des bases de données SQLite

SQLiteStudio est un outil graphique open-source conçu pour faciliter la gestion des bases de données SQLite. Voici quelques-uns de ses principaux usages et fonctionnalités :

1. **Interface utilisateur conviviale** :

Offre une interface graphique intuitive qui rend l'interaction avec les bases de données SQLite plus simple que l'utilisation de la ligne de commande.

2. **Exploration et gestion des bases de données** :
   - Permet de naviguer facilement dans les tables, vues, index et autres objets de la base de données.
   - Offre des options pour créer, modifier et supprimer des tables, ainsi que pour gérer les données qu'elles contiennent.
3. **Exécution de requêtes SQL** :
   - Permet de rédiger et d'exécuter des requêtes SQL, avec des fonctionnalités comme la coloration syntaxique, l'auto-complétion et la mise en forme du code.
   - Affiche les résultats des requêtes sous forme de tableaux pour une lecture et une analyse plus facile.
4. **Manipulation des données** :
   - Offre des outils pour insérer, mettre à jour et supprimer des enregistrements de manière interactive.
   - Permet d'importer et d'exporter des données vers et depuis divers formats (CSV, SQL, etc.).
5. **Sauvegarde et restauration** :
   - Facilite la sauvegarde de la base de données dans des fichiers pour la restauration ou le partage.
6. **Outils de gestion avancés** :
   - Inclut des fonctionnalités avancées comme la gestion des déclencheurs (triggers), des contraintes, et des transactions.
   - Offre des options pour l'analyse des performances et l'optimisation des requêtes.
7. **Visualisation des schémas de base de données** :
- Permet de visualiser les relations entre les tables et les contraintes sous forme de diagrammes.

1.4.1 Avantages de SQLiteStudio

- **Accessibilité** : Utilisable sur diverses plateformes (Windows, macOS, Linux).
- **Gratuit et Open-source** : Aucun coût d'utilisation et possibilité de contribuer au développement.
- **Portable** : Ne nécessite pas d'installation complexe, peut être utilisé directement après le téléchargement.

1.4.2 Exemple d'utilisation

Vous avez créé une base de données SQLite pour stocker des données météorologiques. Avec SQLiteStudio, vous pouvez :

- **Ouvrir la base de données weather.db** : Visualisez et explorez les tables existantes.
- **Inspecter la table current** : Voir les données insérées, vérifier les structures des tables et les contraintes.

- **Exécuter des requêtes SQL** : Effectuer des sélections, mises à jour ou autres opérations sur les données météorologiques stockées.
- **Gérer la structure de la base de données** : Ajouter ou modifier des colonnes, ajuster les contraintes d'unicité, ou optimiser les index.

1.5 Interaction avec la base des données SQLite

Pour récupérer et afficher toutes les données météorologiques précédemment enregistrées. Voici une explication détaillée de chaque étape envisageable :

1. **Connexion à la base de données SQLite** :
   - `conn = sqlite3.connect(db_name)` : Cette ligne établit une connexion à une base de données SQLite. `db_name` est une variable qui doit contenir le chemin ou le nom de la base de données SQLite à laquelle on souhaite se connecter. Si la base de données n'existe pas, elle sera créée à cet instant.
2. **Création d'un curseur** :
   - `cursor = conn.cursor()` : Un curseur est créé à partir de la connexion établie. Le curseur est utilisé pour exécuter des commandes SQL sur la base de données et récupérer les résultats.
3. **Exécution d'une requête SQL** :
   - `cursor.execute(f"SELECT * FROM {table_name}")` : Cette ligne exécute une requête SQL pour sélectionner toutes les colonnes (*) de toutes les lignes de la table spécifiée par `table_name`. Le contenu de `table_name` doit être remplacé par le nom de la table exacte dans la base de données à interroger.
4. **Récupération des résultats** :
   - `rows = cursor.fetchall()` : Cette ligne récupère toutes les lignes de résultats de la requête SQL précédemment exécutée. Les résultats sont stockés dans la variable `rows` sous forme d'une liste de tuples, où chaque tuple représente une ligne de la table.
5. **Affichage des résultats** :
   - `for row in rows:` : Une boucle `for` est utilisée pour parcourir chaque tuple dans la liste `rows`.
   - `print(row)` : À chaque itération de la boucle, la ligne actuelle (un tuple) est imprimée sur la console. Cela affiche chaque enregistrement de la table, un par ligne, sur la sortie standard.
6. **Fermeture de la connexion à la base de données** :
   - `conn.close()` : Une fois que toutes les opérations sur la base de données sont terminées, la connexion est fermée. Cela libère les ressources utilisées par la connexion et empêche toute fuite de mémoire potentielle.