Databases with Python

SQLite

SQLite is a C library that provides a lightweight, disk-based database. It doesn't require a separate server process and allows access to the database using a nonstandard variant of the SQL query language. Python comes with a built-in SQLite database library, `sqlite3`, which lets you interact with SQLite databases.

Here's a comprehensive guide on how to perform CRUD (Create, Read, Update, Delete) operations in SQLite using Python:

1. Importing the sqlite3 Module

```
import sqlite3
```

2. Creating a Connection

You need to create a connection object that represents the database. SQLite databases are stored in files, and a new database gets created if it doesn't exist.

```
conn = sqlite3.connect('example.db')
```

3. Creating a Cursor Object

The cursor object is used to execute SQL commands.

```
cursor = conn.cursor()
```

4. Creating a Table (Create in CRUD)

Let's create a simple table named `employees` with a few columns.

```
cursor.execute('''CREATE TABLE employees
                  (id INTEGER PRIMARY KEY, name TEXT, position TEXT, salary REAL)''')
```

5. Inserting Data (Create in CRUD)

Insert data into the `employees` table.

```
cursor.execute("INSERT INTO employees (name, position, salary) VALUES ('Rajath', 'Thought Leaader', 5000)")
```

6. Committing Changes

After executing SQL commands, commit the changes to the database.

```
conn.commit()
```

7. Querying Data (Read in CRUD)

Read data from the table.

```
cursor.execute("SELECT * FROM employees")
print(cursor.fetchall())
```

8. Updating Data (Update in CRUD)

Update data in the table.

```
cursor.execute("UPDATE employees SET salary = 5500 WHERE name = 'Rajath'")
conn.commit()
```

9. Deleting Data (Delete in CRUD)

Delete data from the table.

```
cursor.execute("DELETE FROM employees WHERE name = 'Rajath'")
conn.commit()
```

10. Using Placeholders

To avoid SQL injection, use placeholders (`?`) instead of directly using string formatting.

```
employee_data = ('Elon Musk', 'Developer', 6000)
cursor.execute("INSERT INTO employees (name, position, salary) VALUES (?, ?, ?)", employee_data)
conn.commit()
```

11. Closing the Connection

Finally, close the connection if you are done with your operations.

```
conn.close()
```

CRUD Operations using SQLite Database

Below are separate examples for each of the CRUD operations—Create, Read, Read All, Update, and Delete—using SQLite in Python. These examples assume you're working with a table named `employees` in a database called `example.db`

1. Create (Inserting Data)

```python
import sqlite3

def create_employee(name, position, salary):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    cursor.execute("INSERT INTO employees (name, position, salary) VALUES (?, ?, ?)", (name, position, salary))

    conn.commit()
    conn.close()

# Example Usage
create_employee('Rajath', 'Manager', 5000)
create_employee('Elon', 'Engineer', 6000)
```

2. Read (Fetching a SIngle Record)

```python
def read_employee(employee_id):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM employees WHERE id = ?", (employee_id,))
    employee = cursor.fetchone()

    conn.close()
    return employee

# Example Usage
employee = read_employee(1)
print(employee)
```

3. Read All (Fetching All Records)

```python
def read_all_employees():
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM employees")
    employees = cursor.fetchall()

    conn.close()
    return employees

# Example Usage
employees = read_all_employees()
for employee in employees:
    print(employee)

```

4. Update (Updating a Record)

```python
def update_employee(employee_id, name, position, salary):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    cursor.execute("UPDATE employees SET name = ?, position = ?, salary = ? WHERE id = ?", (name, position, salary, employee_id))

    conn.commit()
    conn.close()

# Example Usage
update_employee(1, 'BillGates', 'CTO', 7000)

```

5. Delete (Deleting a Record)

```python
def delete_employee(employee_id):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    cursor.execute("DELETE FROM employees WHERE id = ?", (employee_id,))

    conn.commit()
    conn.close()

# Example Usage
delete_employee(1)

```

Notes on Usage

- Each function establishes its own connection to the SQLite database and closes it afterward. This is a simple approach and works well for small applications. For larger applications, you might want to manage your connections and cursors more efficiently.
- Error handling (like dealing with incorrect inputs or database issues) is not included in these examples but is essential for robust applications.
- Always be cautious to prevent SQL injection by using placeholders (`?`) rather than formatting strings directly. This is demonstrated in the examples above.

MySQL

Working with MySQL in Python requires a good understanding of both the MySQL database system and the Python programming language. MySQL is an open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) for managing data. Python, with its libraries for database interaction, makes it easy to connect to and manipulate MySQL databases.

Here's a comprehensive step-by-step guide to using MySQL with Python:

Introduction to MySQL

MySQL is a popular choice for many web applications, including content management systems and web applications. It is renowned for its performance, reliability, and ease of use. MySQL works on various platforms and supports a wide range of applications, making it a versatile choice for both developers and database administrators.

Step 1: Install MySQL Server and MySQL Connector

Before you start, ensure you have MySQL Server installed on your system. You can download it from the official MySQL website.

Next, you will need a MySQL driver to connect to the MySQL database from Python. Install the `mysql-connector-python` package:

```
pip install mysql-connector-python
```

Step 2: Import MySQL Connector

In your Python script, import the installed connector:

```python
import mysql.connector
from mysql.connector import Error
```

Step 3: Create a Connection to MySQL

Establish a connection to the MySQL database. Replace the placeholders with your MySQL instance's host, username, password, and database name.

```python
def create_connection(host_name, user_name, user_password, db_name):
    connection = None
    try:
        connection = mysql.connector.connect(
            host=host_name,
            user=user_name,
            password=user_password,
            database=db_name
        )
        print("Connection to MySQL DB successful")
    except Error as e:
        print(f"The error '{e}' occurred")

    return connection

# Replace with your MySQL server details
connection = create_connection("192.168.0.116", "rajath", "rajathkumar", "sampledb")
```

Step 4: Create a Table

Define a function to execute queries, including one for creating tables.

```python
def execute_query(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        connection.commit()
        print("Query executed successfully")
    except Error as e:
        print(f"The error '{e}' occurred")

# Example query to create a table
create_table_query = """
CREATE TABLE IF NOT EXISTS employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    position VARCHAR(100) NOT NULL,
    salary DECIMAL(10 , 2)
);"""

execute_query(connection, create_table_query)
```

Step 5: Insert Data (Create Operation)

Create a function to insert data into the table.

```python
def insert_employee(connection, name, position, salary):
    query = """
    INSERT INTO employees (name, position, salary)
    VALUES (%s, %s, %s);
    """
    args = (name, position, salary)
    execute_query(connection, query, args)

# Example usage
insert_employee(connection, "Rajath", "Manager", 5000)
```

Step 6: Query Data (Read Operation)

Create a function to query data from the table.

```python
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")

# Example query
select_employees = "SELECT * from employees"
employees = execute_read_query(connection, select_employees)

for employee in employees:
    print(employee)
```

Step 7: Update Data

Create a function to update existing data.

```python
def update_employee_salary(connection, id, new_salary):
    query = """
    UPDATE employees
    SET salary = %s
    WHERE id = %s;
    """
    args = (new_salary, id)
    execute_query(connection, query, args)

# Example usage
update_employee_salary(connection, 1, 60000)
```

Step 8: Delete Data

Create a function to delete data.

```python
def delete_employee(connection, id):
    query = "DELETE FROM employees WHERE id = %s"
    args = (id,)
    execute_query(connection, query, args)

# Example usage
delete_employee(connection, 1)
```

Step 9: Close the Connection

After all your operations are complete, always close the connection.

```python
connection.close()
```

Final Notes

- Ensure you have error handling in place to deal with any issues that arise during database operations.
- It's good practice to use parameterized queries (as shown in the insert and update examples) to prevent SQL injection attacks.
- Always securely manage your database credentials and access.
- For complex applications, consider using an ORM (Object-Relational Mapping) library like SQLAlchemy or SQL Model for more robust database handling.

SQLModel (ORM for Python)

`SQLModel` is a Python library that simplifies interactions with SQL databases. It combines the capabilities of `SQLAlchemy` for interacting with relational databases and `Pydantic` for data validation and schema definition. `SQLModel` is particularly useful when you are working with FastAPI, as it is designed to integrate seamlessly with it, although it can be used in any Python project.

Step 1: Install SQLModel

First, install SQLModel if you haven't already:

```
pip install sqlmodel
```

Step 2: Define the Model

In `SQLModel`, you define your data structure as classes. Here, we will define an `Employee` model.

```python
from sqlmodel import SQLModel, Field

class Employee(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    name: str
    position: str
    salary: float
```

Step 3: Create the Database and Tables

`SQLModel` can create the database and tables based on your models. Here's an example of how to create the SQLite database and the employees table.

```python
from sqlmodel import create_engine

# SQLite Database
sqlite_file_name = "database.db"
sqlite_url = f"sqlite:///{sqlite_file_name}"

engine = create_engine(sqlite_url)

# Create the tables
SQLModel.metadata.create_all(engine)
```

Step 4: CRUD Operations with SQLModel

Create (Insert Data)

To create (or insert) data, you can use a session from `SQLModel` to add objects to your database.

```python
from sqlmodel import Session

def create_employee(employee: Employee):
    with Session(engine) as session:
        session.add(employee)
        session.commit()
        session.refresh(employee)
        return employee

# Usage
new_employee = Employee(name="Rajath", position="Manager", salary=5000)
created_employee = create_employee(new_employee)
```

Read (Fetch Data)

You can read data by querying the database for your model objects.

```python
def get_employee(employee_id: int):
    with Session(engine) as session:
        employee = session.get(Employee, employee_id)
        return employee

# Usage
employee = get_employee(1)
```

Read All (Fetch All Data)

Fetching all entries from the table is similar, but uses `session.exec()`.

```python
from sqlmodel import select

def get_all_employees():
    with Session(engine) as session:
        return list(session.exec(select(Employee)))

# Usage
employees = get_all_employees()
# print(employees)
```

Update (Modify Data)

To update data, you fetch the object, modify its properties, and then commit the changes.

```python
def update_employee_salary(employee_id: int, new_salary: float):
    with Session(engine) as session:
        employee = session.get(Employee, employee_id)
        if employee:
            employee.salary = new_salary
            session.add(employee)
            session.commit()
            session.refresh(employee)
        return employee

# Usage
update_employee_salary(1, 55000)
```

Delete (Remove Data)

Deleting an object is straightforward - fetch it and then remove it from the session.

```python
def delete_employee(employee_id: int):
    with Session(engine) as session:
        employee = session.get(Employee, employee_id)
        if employee:
            session.delete(employee)
            session.commit()

# Usage
delete_employee(1)
```

Final Notes

- `SQLModel` is a powerful tool that leverages the strengths of both `SQLAlchemy` and `Pydantic`. It provides an elegant way to define your database schema and interact with the database.
- It is particularly beneficial in projects where you need quick development and less boilerplate code for database operations.
- Make sure to handle exceptions and errors that can occur during database operations.
- For more complex database interactions or advanced features, refer to the `SQLModel` documentation for additional capabilities and options.

MongoDB

Working with MongoDB in Python typically involves using the `pymongo` library, which is a popular MongoDB driver for Python. MongoDB is a NoSQL database that stores data in JSON-like documents (BSON), making it quite different from SQL databases like MySQL. Here's a comprehensive guide to performing CRUD operations in MongoDB using `pymongo`.

Step 1: Install PyMongo

First, install the `pymongo` package. You might also want to install `dnspython` for DNS support (useful if you are connecting to MongoDB Atlas).

```
pip install pymongo dnspython
```

Step 2: Connect to MongoDB

Create a connection to your MongoDB instance. If you're using MongoDB Atlas, you'll have a connection URI. If you're using a local instance, the connection URI will typically be `'mongodb://localhost:27017/'`.

```
from pymongo import MongoClient

# MongoDB connection string
connection_string = "your_connection_string_here"

# Create a MongoClient
client = MongoClient(connection_string)

# Connect to the database
db = client['example_db']
```

Step 3: Define the Collection

In MongoDB, data is stored in collections, which are similar to tables in SQL databases.

```
# Define the collection
employees = db.employees
```

CRUD Operations in MongoDB with PyMongo

Create (Insert Data)

To insert a document into a collection, you use the `insert_one()` or `insert_many()` method.

```
def insert_employee(employee_data):
    return employees.insert_one(employee_data).inserted_id

# Usage
new_employee = {
    "name": "Rajath",
    "position": "Manager",
    "salary": 5000
}
insert_employee(new_employee)
```

Read (Fetch Data)

To read documents, you can use `find_one()` to get a single document or `find()` to get multiple documents.

```python
def find_employee(name):
    return employees.find_one({"name": name})

# Usage
employee = find_employee("Rajath")
print(employee)
```

For fetching all documents:

```python
def find_all_employees():
    return list(employees.find())

# Usage
all_employees = find_all_employees()
print(all_employees)
```

Update (Modify Data)

To update documents, you can use `update_one()`, `update_many()`, or their variants.

```python
def update_employee_salary(name, new_salary):
    return employees.update_one({"name": name}, {"$set": {"salary": new_salary}})

# Usage
update_employee_salary("Rajath", 5500)
```

Delete (Remove Data)

To delete documents, you use `delete_one()` or `delete_many()`.

```python
def delete_employee(name):
    return employees.delete_one({"name": name})

# Usage
delete_employee("Rajath")
```

Closing the Connection

Close the connection if it's no longer needed.

```python
client.close()
```

Final Notes

- MongoDB's document model is quite different from SQL's table model. The flexibility of MongoDB documents can be very powerful.
- Always ensure the security of your MongoDB instance, especially if it's exposed over the internet.
- MongoDB is well-suited for use cases that require flexible schemas and scalability.
- `pymongo` offers much more functionality, including indexing, aggregation, map-reduce, and more advanced querying capabilities. For complex applications, you should explore these advanced features.
- Error handling and validation are crucial, especially given the schema-less nature of MongoDB. While `pymongo` does some basic checks, the responsibility largely falls on the application to ensure data integrity.
- You can also use the other ODM - `Beanie for its Async Operation`