# SQLAlchemy

SQLAlchemy is a Python SQL toolkit and Object-Relational Mapping (ORM) library that provides a set of high-level API for interacting with databases in Python. SQLAlchemy allows developers to write Python code that interacts with databases using SQL expressions, and also provides an ORM that allows users to interact with the database using Python classes and objects.

## Installing SQLAlchemy

You can install SQLAlchemy using pip:

```
pip install sqlalchemy
```

## Connecting to a Database

To connect to a database using SQLAlchemy, you need to create an instance of the `create_engine` class. The `create_engine` class takes a connection string as an argument. The connection string is a URL that specifies the database driver, username, password, hostname, and database name. The following example shows how to connect to a SQLite database:

```
from sqlalchemy import create_engine, text

engine = create_engine("sqlite:///example.db")

# Test the connection
with engine.connect() as conn:
    result = conn.execute(text("SELECT 1"))
    print(result.fetchone())
```

For the sake of simplicity, the examples in this tutorial will use a SQLite database. You can use any other database that SQLAlchemy supports. You can find the list of supported databases here. For example, to connect to a Microsoft SQL Server database using SQLAlchemy, you first need to install the necessary drivers. You can use `pyodbc` or `pymssql` for this purpose. Here's how to install `pyodbc`:

```
pip install pyodbc
```

After installing the necessary packages, you can connect to the Microsoft SQL Server database using SQLAlchemy. Here's an example of how to do it using `pyodbc`:

```python
from sqlalchemy import create_engine

server = 'server_name'
database = 'database_name'
username = 'username'
password = 'password'
driver= '{ODBC Driver 17 for SQL Server}'
# or '{SQL Server Native Client 11.0}' depending on your configuration

engine = create_engine(
    f'mssql+pyodbc://{username}:{password}@{server}/{database}?driver={driver}'
)
```

Before you start, make sure that the SQL Server allows remote connections. If you're having trouble connecting, it might be due to network permissions or firewall settings. Also, keep in mind that the driver name in the connection string depends on the version of the ODBC driver installed on your system, so adjust it accordingly.

The examples below should work with any database that SQLAlchemy supports. You just need to change the connection string to match your database configuration.

## Defining a Table

Next, we need to define a table in our database. Tables are represented in SQLAlchemy as Python classes that inherit from the `Table` class. Each table class defines the columns of the table, as well as any constraints or indexes.

Here's an example table definition for a simple users table:

```python
from sqlalchemy import Column, Integer, String, DateTime
from sqlalchemy.orm import declarative_base

Base = declarative_base()


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    age = Column(Integer)
    name = Column(String)
    email = Column(String)
    created_at = Column(DateTime)
```

This defines a table named users with four columns: `id`, `name`, `email`, and `created_at`. The `id` column is the primary key, which means that it uniquely identifies each row in the table.

## Creating a Table

Once we've defined our table, we can create it in the database using the `create_all` method of our `Base` object:

```
# Create the table
Base.metadata.create_all(engine)
```

This creates the `users` table in our database.

## Inserting Data

Now that we've created our table, we can insert data into it using the `Session` object. The `Session` object represents a transactional scope of the database, and is used to insert, update, and delete data.

Here's an example of how to insert a new user into the `users` table:

```
from sqlalchemy.orm import sessionmaker
from datetime import datetime

# Create a session
Session = sessionmaker(bind=engine)
session = Session()

# Create a new user
new_user = User(
    name="John Doe",
    age=34,
    email="john@example.com",
    created_at=datetime.now(),
)
new_user_2 = User(
    name="Anthony Joe",
    age=54,
    email="anthony@example.com",
    created_at=datetime.now(),
)

# Add the users to the session
session.add(new_user)
session.add(new_user_2)

# Commit the session
session.commit()
```

This creates a new user object, adds it to the session, and then commits the transaction to the database.

### Querying Data

We can retrieve data from our database using the `query` method of our `Session` object. Here's an example of how to retrieve all users from the `users` table:

```
def query_all_users(session):
    # Query all users
    users = session.query(User).all()

    # Print the details for each user
    for user in users:
        print(user.id, user.name, user.age, user.email, user.created_at)


query_all_users(session)
```

### Updating Data

To update data in a table, you can use the `update` method of the table object. The `update` method takes a dictionary of values that map to the columns of the table. The `update` method also takes a `where` clause that specifies the rows to update.

```
# Retrieve the user we want to update
user = session.query(User).filter_by(id=1).first()

# Update the user's email address
user.email = "newemail@example.com"
```

```
# Commit the changes
session.commit()
```

```
query_all_users(session)
```

### Deleting Data

To delete data from a table, you need to retrieve the data you want to delete using a query, delete the data, and then commit the changes. Here's an example of how to delete a user from the `users` table:

```
# Retrieve the user we want to delete
user = session.query(User).filter_by(id=1).first()

# Delete the user
session.delete(user)

# Commit the changes
session.commit()
```

```
query_all_users(session)
```

### Relationships

Relationships are a key feature of relational databases. They allow you to define relationships between tables, and then use those relationships to retrieve data from multiple tables at once. SQLAlchemy provides a number of different ways to define relationships between tables. In this section, we will explore three types of relationships: one-to-many, many-to-one, and many-to-many.

#### One-to-Many

In a one-to-many relationship, one record in a table can have multiple related records in another table. For example, let's create a `Post` table with a one-to-many relationship to the `User` table:

```
from sqlalchemy import ForeignKey, Sequence
from sqlalchemy.orm import relationship


class Post(Base):
    __tablename__ = "posts"
    id = Column(Integer, Sequence("post_id_seq"), primary_key=True)
    title = Column(String(100))
    content = Column(String(500))
    user_id = Column(Integer, ForeignKey("users.id"))

    # Define the relationship between User and Post
    author = relationship("User", back_populates="posts")


# Add the relationship to the User class
User.posts = relationship("Post", back_populates="author")

# Create the posts table
Base.metadata.create_all(engine)
```

Now, you can create a new post and associate it with a user:

```
user = User(name="Jane Doe", age=28)
post = Post(title="My First Post", content="Hello, world!", author=user)

session.add(user)
session.add(post)
session.commit()
```

To retrieve a user's posts:

```
user_posts = session.query(Post).filter(Post.author == user).all()

for post in user_posts:
    print(f"{post.title}: {post.content}")
```

A many-to-one relationship is the inverse of a one-to-many relationship. In this case, we have already defined a many-to-one relationship between Post and User.

To query the author of a post:

```
post_author = session.query(User).filter(User.posts.contains(post)).first()
print(f"Author: {post_author.name}")
```

### Many-to-Many

In a many-to-many relationship, records in one table can be related to multiple records in another table and vice versa. Let's create a Tag table and establish a many-to-many relationship with the Post table:

```
from sqlalchemy import Table

# Define the association table for the many-to-many relationship
post_tags = Table(
    "post_tags",
    Base.metadata,
    Column("post_id", Integer, ForeignKey("posts.id"), primary_key=True),
    Column("tag_id", Integer, ForeignKey("tags.id"), primary_key=True),
)


class Tag(Base):
    __tablename__ = "tags"
    id = Column(Integer, Sequence("tag_id_seq"), primary_key=True)
    name = Column(String(20), unique=True)

    # Define the relationship between Tag and Post
    posts = relationship("Post", secondary=post_tags, back_populates="tags")


# Add the relationship to the Post class
Post.tags = relationship("Tag", secondary=post_tags, back_populates="posts")

# Create the tags table
Base.metadata.create_all(engine)
```

Now, you can create new tags and associate them with a post:

```
tag1 = Tag(name="Python")
tag2 = Tag(name="Tutorial")
```

```
post.tags = [tag1, tag2]

session.add(tag1)
session.add(tag2)
session.add(post)
session.commit()
```

To retrieve a post's tags:

```
post_tags = session.query(Tag).filter(Tag.posts.contains(post)).all()

for tag in post_tags:
    print(f"Tag: {tag.name}")
```

To retrieve all posts with a specific tag:

```
tag_posts = session.query(Post).filter(Post.tags.contains(tag1)).all()

for post in tag_posts:
    print(f"{post.title}: {post.content}")
```

### Complex Queries

You can write custom queries using SQLAlchemy's expression language, which allows you to create complex queries with a Pythonic syntax:

```
from sqlalchemy import and_, or_, not_

# Find all users with age between 20 and 40 and a name containing "Doe"
users = (
    session.query(User)
    .filter(and_(User.age.between(20, 40), User.name.like("%Doe%")))
    .all()
)

for user in users:
    print(f"User {user.id}: {user.name}, {user.age} years old")
```

### Aggregations

SQLAlchemy allows you to perform aggregation operations like COUNT, SUM, MIN, MAX, and AVG:

```python
from sqlalchemy import func

# Count the number of users
user_count = session.query(func.count(User.id)).scalar()
print(f"Total users: {user_count}")

# Calculate the average age of users
average_age = session.query(func.avg(User.age)).scalar()
print(f"Average age: {average_age}")
```

### Joining Tables

You can perform joins in SQLAlchemy to combine data from multiple tables:

```python
# Fetch all users along with their posts
users_with_posts = (
    session.query(User, Post).join(Post, User.id == Post.user_id).all()
)

for user, post in users_with_posts:
    print(f"User {user.id}: {user.name} - Post {post.id}: {post.title}")
```

### Closing the Session

When you're done interacting with the database, you need to close the session to release the database connection. You can do this using the `close` method of the `Session` object:

```python
session.close()
```

### Transactions

Transactions ensure that a group of SQL statements are executed completely, or not at all. If an error occurs during execution, the transaction can be rolled back to revert any changes.

In SQLAlchemy, a transaction is automatically started when you first use the session object. You can also manage transactions explicitly:

```python
# Start a new transaction
session.begin()

try:
    new_user = User(name="Bob Smith", age=40)
    session.add(new_user)

    new_post = Post(
        title="My Second Post", content="Another post!", author=new_user
    )
    session.add(new_post)

    # Commit the transaction
    session.commit()
    query_all_users(session)
except Exception as e:
    # Roll back the transaction in case of an error
    session.rollback()
    print(f"Error: {e}")
finally:
    # Clean up the session
    session.close()
```

You could also create a SQLAlchemy session using a context manager. This will automatically handle the session lifecycle (opening, committing, and closing) within the context of the `with` statement (like we did earlier when testing the connection to our database). This is the preferred way to manage transactions in SQLAlchemy, as it ensures that the session is closed when you're done using it. Here's an example of how to do it:

```python
from contextlib import contextmanager

# Set up the sessionmaker
Session = sessionmaker(bind=engine)


@contextmanager
def session_scope():
    """Provide a transactional scope around a series of operations."""
    session = Session()
    try:
        # This yields the session to the context manager to perform operations.
        yield session
        session.commit()
    except Exception:
        session.rollback()
        raise
    finally:
        session.close()


# Then, when you want to use a session:
with session_scope() as session:
    # Perform operations on the database
    my_data = session.query(User).filter(User.name == "Jane Doe").first()

    print(my_data.name, my_data.age)
```

However, these are very advanced concepts and it is not expected for you to understand them right away. For more information on transactions, see the SQLAlchemy documentation.

## Project: Library Management System

### Objective

Develop a library management system using SQLAlchemy for a small community library to manage books, members, and borrowing.

### Requirements

1. Python 3.7 or higher
2. SQLAlchemy 1.4.29 or higher
3. SQLite 3

### Project Structure

```
library_management_system/
  models/
      __init__.py
      base.py
```

```
        book.py
        member.py
        loan.py
    operations/
        __init__.py
        book_operations.py
        member_operations.py
        loan_operations.py
    database.py
    main.py
```

### Instructions

1. Create the following folder structure for the project:

    - Create a project folder with the name `library_management_system`.
    - Create a `models` subfolder inside the project folder. Create an empty `__init__.py` file inside the `models` folder.
    - Create a `operations` subfolder inside the project folder. Create an empty `__init__.py` file inside the `operations` folder.
    - Alternatively, you could create a single `models.py` file and a single `operations.py` file instead of creating separate subfolders/files for each model and operation.

2. Install the dependencies.

    ```
    pip install sqlalchemy
    ```

    Should you wish to expand on this project later and add more dependencies, I recommend using a virtual environment. You can create a virtual environment using the following command in your terminal (make sure the project folder is your current working directory):

    ```
    python -m venv .venv
    ```

    You can activate the virtual environment using the following command (Linux and macOS):

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

    If you're using Windows, you can activate the virtual environment using the following commands:

    PowerShell:

```
.venv\Scripts\Activate.ps1
```

CMD:

```
.venv\Scripts\activate.bat
```

See the Python documentation for more information on virtual environments.

3. Create the declarative base object `Base` in the `models/base.py` file (or inside the `models.py` file). This object will be used as the base class for all the models in our project. It will also be used to create the database tables.

4. Create models for the `Book`, `Member`, and `Loan` tables in separate files (`book.py`, `member.py`, `loan.py`) inside the `models` folder (or together inside `models.py`). Each model should inherit from the `Base` object, which can be imported using the following code (assuming the `Base` object is defined in `models/base.py`, otherwise no import is needed):

   ```
   from .base import Base
   ```

   The `Book` model should have the following columns:

   - `id`: Integer, primary key
   - `title`: String, not null
   - `author`: String, not null
   - `isbn`: String, not null
   - `available`: Integer, not null

   The `Member` model should have the following columns:

   - `id`: Integer, primary key
   - `name`: String, not null
   - `email`: String, not null
   - `phone`: String, not null

   The `Loan` model should have the following columns:

   - `id`: Integer, primary key
   - `book_id`: Integer, foreign key
   - `member_id`: Integer, foreign key
   - `start_date`: DateTime, not null
   - `due_date`: DateTime, not null
   - `return_date`: DateTime, nullable

The `Loan` model should have a many-to-one relationship with the `Book` and `Member` models.

5. Create a `database.py` file to initialize the database and create the tables. The `database.py` file should contain two functions to be imported in the `main.py` file, or any other file where you want to interact with the database (e.g. a Jupyter notebook):

   - `init_db`: This function should be called to initialize the database and create the tables.
   - `get_session`: This function should be called to create a session object to interact with the database.

   The `init_db` function should do the following:

   - Create an `engine` object.
   - Create all the tables in the database with the `engine` object as an argument.

   The `get_session` function should do the following:

   - Create a `Session` object.
   - Return the `Session` object.

   You can import the `Base` object from the `models/base.py` file using the following code:

   ```
   from models.base import Base
   ```

   or if you're using a single `models.py` file:

   ```
   from models import Base
   ```

6. (Optional) Create operations for the `Book`, `Member`, and `Loan` tables using separate files (`book_operations.py`, `member_operations.py`, `loan_operations.py`) inside the `operations` folder (or together inside the `operations.py` file). These operations should include methods to create, retrieve, update, and delete records.

7. Create a `main.py` file or Jupyter notebook to interact with the database. For example you could do the following:

   - Create a new book.
   - Create a new member.
   - Create a new loan.
   - Retrieve all books.
   - Retrieve all members.
   - Retrieve all loans.
   - Retrieve a book by ID.
   - Retrieve a member by ID.

- Retrieve a loan by ID.
- Update a book.
- Update a member.
- Update a loan.
- Delete a book.
- Delete a member.
- Delete a loan.

## Further Reading

- SQLAlchemy 2.0 Documentation
- DataCamp: SQLAlchemy Tutorial with Examples
- Real Python: Data Management With Python, SQLite, and SQLAlchemy
- Tutorialspoint: SQLAlchemy Tutorial
- Towards Data Science: SQLAlchemy — Python Tutorial