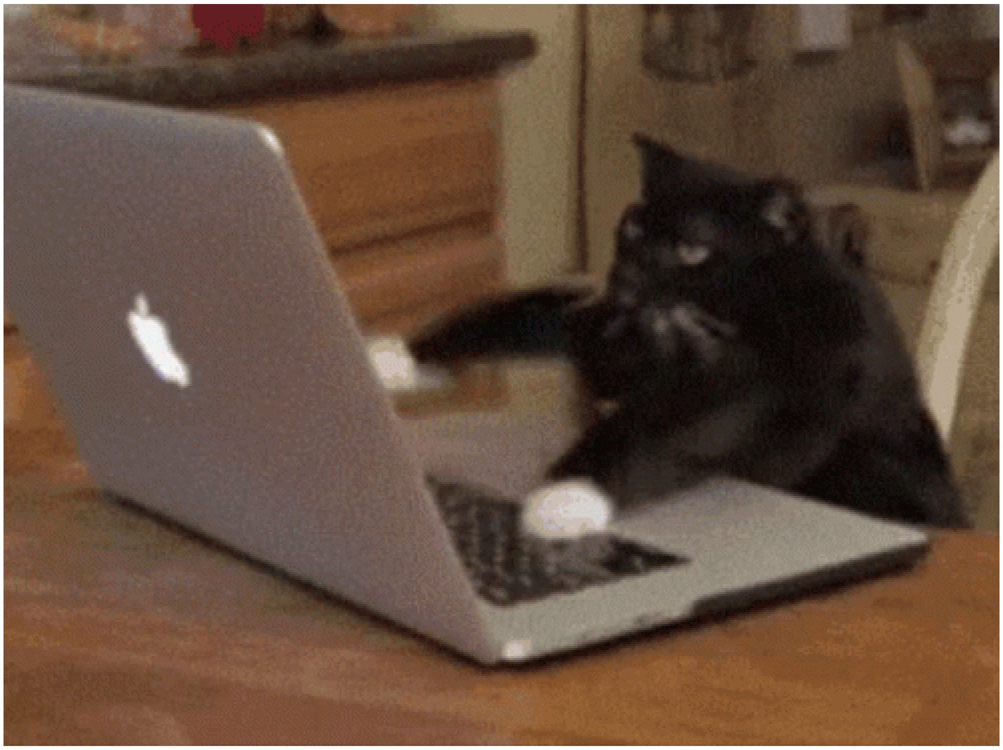
Creating a database using Python and SQLAlchemy

S

Sandyjtech · Follow

5 min read · Aug 3, 2023



> *SQLAlchemy is an awesome Object-Relational Mapping (ORM) library that allows us to interact with databases using Python. I will provide you with my step-by-step*

> *notes on how to create a database using Python and Alchemy.*

Prerequisites:

Before you begin you have to make sure you have the necessary tools installed.

1. Python
2. SQLAlchemy (install via pip: `pip install sqlalchemy`)

> *Awesome. Now let's begin!*

Step 1: Import necessary modules

Let's start by creating a new Python script and importing the necessary modules.

```
from sqlalchemy import create_engine, Column, Integer, String, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

Step 2: Establish a database connection

Create a database connection using SQLAlchemy's `create_engine` function. Replace `'your_database_url'` with the URL of your database.

- For PostgreSQL, the URL format is similar to MySQL:

```
# PostgreSQL database URL
```

```
database_url = 'postgresql://username:password@host/database_name'
```

- For MySQL, you need to provide the necessary credentials (username, password, host, and database name) in the URL format:

```
# MySQL database URL
database_url = 'mysql+pymysql://username:password@host/database_name'
```

For MySQLite, he URL would be:

```
engine = create_engine('sqlite:///your_database_name.db')
```

Note: Replace `'username'` , `'password'` , `'host'` , and `'database_name'` with your MySQL or PostgreSQL credentials and database information. It's good to remember the database needs to be installed and running before you are connecting to it.

Step 3: Define your data mode

To create your data model, make a Python class that inherits from the Base. Each attribute of the class represents a column in the relevant database table. You can include additional columns as you see fit.

```
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    username = Column(String(50), unique=True, nullable=False)
    email = Column(String(100), unique=True, nullable=False
    password = Column(String(100), unique=True, nullable=False)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)
```

Step 4: Create the database tables

With the data model defined, create the database tables using the `create_all` method of the `Base` class:

```
Base.metadata.create_all(engine)
```

Step 5: Insert data into the database

To insert data into your database, first, create a session using SQLAlchemy's `sessionmaker`:

```
Session = sessionmaker(bind=engine)
session = Session()
```

Step 6: Adding and creating objects

Hurray! Now, you can create objects and add them to your database.

```
# Example: Inserting a new user into the database
new_user = User(username='Sandy', email='sandy@gmail.com', password='cool-passwo
session.add(new_user)
session.commit()
```

Step 7: Query data from the database

To retrieve data from the database, use the session’s `query` method:

```
# Example: Querying all users from the database
all_users = session.query(User).all()

# Example: Querying a specific user by their username
user = session.query(User).filter_by(username='Sandy').first()
```

Step 8: Close the session

Remember to close the session once you’re done working with the database:

```
session.close()
```

SQL vs SQLAlchemy

> *If we were to compare SQLAlchemy and plain SQL, we could notice the difference in how they interact with data and the advantages they offer.*

1. Abstraction level:

- *SQLAlchemy*: Provides a high-level abstraction with Object-Relational Mapping (ORM). It eliminates the need to write raw SQL queries.

- *Plain SQL*: Requires us to write raw SQL queries and manually handle the mapping between Python data and the database tables.

2. Portability:

- *SQLAlchemy*: Offers database abstraction allowing us to easily switch between different database URLs.
- *Plain SQL*: Often writes database-specific queries, requiring us to rewrite the queries to match a new database's syntax when switching systems.

3. Safety and security:

- *SQLAlchemy*: Provides protection against SQL injection attacks. SQL injection is a type of code injection that targets data-driven applications by inserting malicious SQL statements into an entry field. SQLAlchemy handles parameter binding and query construction to prevent these attacks.
- *Plain SQL*: Vulnerable to SQL injection if user-provided data is not properly sanitized. To sanitize is to program our system to detect malicious bugs that can harm our database.

4. Code maintainability and readability:

- *SQLAlchemy*: Enhances the maintainability and readability of code, just like regular Python code is used for interacting with databases.
- *Plain SQL*: Queries can become long and complex, and potentially impact our code's readability and maintainability.

5. Database schema management:

- *SQLAlchemy:* Simplifies schema management and version control by providing tools for managing database schemas using Python classes.
- *Plain SQL*: Requires manual handling of schema changes using SQL migration scripts or external tools.

> *Migration scripts are SQL scripts made by users in ApexSQL Source Control. They help configure changes, handle overrides and more.*

After everything has been said and done, our SQLAlchemy code should resemble the following example:

```
# Step 1: Import the necessary modules

from sqlalchemy import create_engine, Column, Integer, String, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
import datetime


# Step 2: Establish a database connection

database_url = 'sqlite:///your_database_name.db'

# Create an engine to connect to a SQLite database
engine = create_engine(database_url)

#will return engine instance
Base = declarative_base()

# Step 3: Define your data model
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    username = Column(String(50), unique=True, nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

# Step 4: Create the database tables
Base.metadata.create_all(engine)

# Step 5: Insert data into the database
Session = sessionmaker(bind=engine)
session = Session()

# Example: Inserting a new user into the database
new_user = User(username='Sandy', email='sandy@gmail.com', password='cool-passwo
session.add(new_user)
session.commit()

# Step 6: Query data from the database
# Example: Querying all users from the database
all_users = session.query(User).all()

# Example: Querying a specific user by their username
user = session.query(User).filter_by(username='Sandy').first()

# Step 7: Close the session
session.close()
```

Conclusion:

Great job! You have successfully mastered the art of creating a database with Python and SQLAlchemy.

Things to remember:

- *Engine*: It establishes a connection to the database and handles the communication behind the scenes for us.
- *Declarative Base*: We utilize SQLAlchemy's declarative base to establish a link between our Python classes and the corresponding database tables.
- *Sessions*: We use this tool to manage our database interactions and it offers us methods to effectively manage CRUD operations.

- *Querying*: This tool allows us to create intricate database queries through Python methods.
- *Relationships*: With SQLAlchemy, we can establish connections between various classes that represent database relationships such as one-to-many, many-to-one, and many-to-many.
- *Mapping*: Using SQLAlchemy, we also have the ability to map attributes of our Python classes to columns in our corresponding database tables.

If you are looking for a flexible library to manage databases in your Python projects, SQLAlchemy is a great choice. It provides a user-friendly and robust interface to handle all your database-related tasks with ease.

Sql Sqlalchemy Python Python Programming Database

S

Written by Sandyjtech

19 Followers

Follow

More from Sandyjtech

S Sandyjtech

Building a Full Stack App with Flask, React, MySQL

Creating a full-stack app is a daunting task and I remember feeling overwhelmed and lo...

6 min read · Sep 1, 2023

82

S Sandyjtech

Mastering Flask: From Fundamentals to Advanced...

Embarking on Flask web development unlocks versatile web application...

5 min read · Aug 18, 2023

57

Sandyjtech

What is Big O Notation?

At the start of my Full-stack Software Engineering Bootcamp, I was intimidated by…

5 min read · Jul 17, 2023

4

Sandyjtech

6 Steps to bettering your life.

I know better said than done.

2 min read · Apr 17, 2023

2

See all from Sandyjtech

Recommended from Medium

Jone4307

SQLAlchemy Intro Guide

SQLAlchemy is a toolkit library that allows users to interact with databases through...

11 min read · 3 days ago

Andrii Hrimov

Flask project structure template

Flask is a popular micro web framework that is used very often, even with the advent of...

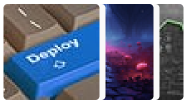
10 min read · Nov 26, 2023

87 1

Lists

Coding & Development

11 stories · 541 saves

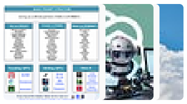


Predictive Modeling w/ Python

20 stories · 1062 saves

Practical Guides to Machine Learning

10 stories · 1273 saves



ChatGPT

21 stories · 551 saves

Pooya Oladazimi

Dockerizing Flask App with Postgres: A Step-by-Step Guide

In the previous episodes about Flask, I explained how to initiate your Flask...

6 min read · Nov 25, 2023

1

Thomas Aitken

Setting up a FastAPI App with Async SQLALchemy 2.0 & Pydant...

Early this year, a major update was made to SQLAlchemy with the release of SQLAlchem...

11 min read · Oct 20, 2023

299 3

Alexander obidiegwu

50 Coding Laws That Would Make You A Decent Programmer.

Follow these laws or get fired.

23 min read · Mar 2, 2024

218 1

Nithin Bharathwaj

SQLAlchemy With Python Flask

Integrating Flask with SQLAlchemy is a common approach for adding a database...

2 min read · Dec 2, 2023

3

See more recommendations