

Saving data

Chapter 23

This chapter covers

- Storing data in relational databases
- Using the Python DB-API
- Accessing databases through an Object
- Relational Mapper (ORM)
- Understanding NoSQL databases and how they differ from relational databases

Relational databases

- Relational databases have long been a standard for storing and manipulating data.
- They're a mature technology and a ubiquitous one.
- Python can connect with a number relational databases, but we don't have the time or the inclination to go through the specifics of each one in this course.
- Instead, because Python handles databases in a mostly consistent way, we are going to illustrate the basics with one of them - sqlite3 - and then discuss some differences and considerations in choosing and using a relational database for data storages.

The Python Database API

- Python handles SQL database access very similarly across several database implementations because of PEP-249 (www.python.org/dev/peps/pep-0249/), which specifies some common practices for connecting to SQL databases.
- Commonly called the Database API or DB-API, it was created to encourage “code that is generally more portable across databases, and a broader reach of database connectivity.”
- Thanks to the DB-API, the examples of SQLite that you see in this chapter are quite similar to what you’d use for PostgreSQL, MySQL, or several other databases.

SQLite: Using the sqlite3 database

- Although it's not suited for large, high-traffic applications, sqlite3 has two advantages:
 - Because it's part of the standard library, it can be used anywhere you need a database without worrying about adding dependencies.
 - sqlite3 stores all of its records in a local file, so it doesn't need both a client and server, which would be the case for PostgreSQL, MySQL, and other larger databases.
- To use a sqlite3 database, the first thing you need is a Connection object.
- Getting a Connection object requires only calling the connect function with the name of file that will be used to store the data.

```
import sqlite3  
  
conn = sqlite3.connect("datafile.db")
```

```
1  import sqlite3
2
3
4  def connect_db():
5      conn = sqlite3.connect("datafile.db")
6      return conn
7
8
9  def get_cursor(conn):
10     cursor = conn.cursor()
11     print(cursor)
12     return cursor
13
14
15  def create_table(conn, cursor):
16     # cursor.execute(
17     #     "create table people (id integer primary key, name text, count integer)")
18     cursor.execute("insert into people (name, count) values ('Bob', 1)")
19     cursor.execute(
20         "insert into people (name, count) values (?, ?)", ("Jill", 15))
21     cursor.execute("insert into people (name, count) values (:username, :usercount)", {
22         "username": "Joe", "usercount": 10})
23     conn.commit()
```

```
26 def get_data(conn, cursor):
27     result = cursor.execute("select * from people")
28     print(result.fetchall())
29     result = cursor.execute(
30         "select * from people where name like :name", {"name": "bob"})
31     print(result.fetchall())
32
33
34 def update_data(conn, cursor):
35     cursor.execute("update people set count=? where name=?", (20, "Jill"))
36     result = cursor.execute("select * from people")
37     print(result.fetchall())
38
39
40 def get_all(conn, cursor):
41     result = cursor.execute("select * from people")
42     for row in result:
43         print(row)
```

```
46  ✓ if __name__ == "__main__":  
47      conn = connect_db()  
48      cursor = get_cursor(conn)  
49      # create_table(conn, cursor)  
50      get_data(conn, cursor)  
51      update_data(conn, cursor)  
52      get_data(conn, cursor)  
53      get_all(conn, cursor)
```


Making database handling easier with an ORM

- There are a few problems with the DB-API database client libraries mentioned earlier in this chapter and their requirement to write raw SQL.
 - Different SQL databases have implemented SQL in subtly different ways.
 - The second drawback is the need to use raw SQL statements.
 - The need to write SQL means that you need to think in at least two languages: Python and a specific SQL variant.
- Given those issues, people wanted a way to handle databases in Python that was easier to manage and didn't require anything more than writing regular Python code.
- The solution is an Object Relational Mapper (ORM), which converts, or maps, relational database types and structures to objects in Python.
- Two of the most common ORMs in the Python world are the Django ORM and SQLAlchemy, although of course there are many others.

SQLAlchemy

- SQLAlchemy is the other big-name ORM in the Python space.
- SQLAlchemy's goal is to automate redundant database tasks and provide Python object-based interfaces to the data while still allowing the developer control of the database and access to the underlying SQL.
- You can install SQLAlchemy in your environment with pip:

```
pip install sqlalchemy
```

NoSQL databases

- Although relational databases are all about normalizing data within related tables, other approaches look at data differently.
- Quite commonly, these types of databases are referred to as NoSQL databases, because they usually don't adhere to the row/column/table structure that SQL was created to describe.
- Rather than handle data as collections of rows, columns, and tables, NoSQL databases can look at the data they store as key-value pairs, as indexed documents, and even as graphs.
- Many NoSQL databases are available, all with somewhat different ways of handling data.
- In general, they're less likely to be strictly normalized, which can make retrieving information faster and easier.

Lab

Create a Database

Summary

- Python has a Database API (DB-API) that provides a generally consistent interface for clients of several relational databases.
- Using an Object Relational Mapper (ORM) can make database code even more standard across databases.
- Using an ORM also lets you access relational databases through Python code and objects rather than SQL queries.
- Tools such as Alembic work with ORMs to use code to make reversible changes to a relational database schema.
- Key:value stores such as Redis provide quick in-memory data access.
- MongoDB provides scalability without the strict structure of relational databases.