13. Working with databases

September 19, 2025

1 Working with databases

1.1 DB-API 2.0

The Python Database API Specification, often referred to as DB-API, is a standard interface for connecting to relational databases from Python programs. It provides a consistent and uniform way to interact with databases, regardless of the underlying database management system (DBMS) being used. DB-API 2.0 was introduced to improve and standardize database connectivity in Python, making it easier for developers to work with databases in a consistent manner.

1.2 sqlite3

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The `sqlite3` module in Python Standard Library implements DB-API 2.0 for SQlite databases.

To work with a database we have to either create a database or use an existing one. Call `sqlite3.connect()` to create a connection to the database `tutorial.db` in the current working directory, implicitly creating it if it does not exist:

[17]:
```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned `Connection` object `con` represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call `con.cursor()` to create the `Cursor`:

[18]:
```
cur = con.cursor()
```

Now that we've got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score.

[19]:
```
cur.execute("CREATE TABLE IF NOT EXISTS movie(title TEXT, year INTEGER, score␣
↪INTEGER)")
```

[19]:
```
<sqlite3.Cursor at 0x104aa3840>
```

Now, add two rows of data supplied as SQL literals by executing an INSERT statement, once again by calling `cur.execute()`:

[20]:
```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

[20]:
```
<sqlite3.Cursor at 0x104aa3840>
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database. Call `con.commit()` on the connection object to commit the transaction:

[21]:
```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query.

[22]:
```
res = cur.execute("SELECT * FROM movie")
res.fetchall()
```

[22]:
```
[('Monty Python and the Holy Grail', 1975, 8.2),
 ('And Now for Something Completely Different', 1971, 7.5)]
```

Now, insert three more rows by calling `cur.executemany()`:

[23]:
```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind data to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks.

We can verify that the new rows were inserted by executing a SELECT query, this time iterating over the results of the query:

[24]:
```
for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
    print(row)
```

```
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Finally, close the cursor and the database connection:

[25]:
```
cur.close()
con.close()
```

1.3 Oracle

Oracle databases are widely used for managing large-scale enterprise data. Whether you're building a data pipeline, automating tasks, or developing applications, integrating Oracle with Python makes these tasks more efficient and accessible.

The official Python library for connecting to Oracle databases is **python-oracledb**. It replaces the older `cx_Oracle` driver, simplifying installation and offering more flexibility.

Official documentation: python-oracledb

1.3.1 1. Setting Up the Environment

Install the `oracledb` library:

```
pip install oracledb
```

No additional drivers or Oracle Clients are needed in most cases.

1.3.2 2. Thin vs. Thick Modes

`python-oracledb` supports two connection modes:

- **Thin Mode (Default):**
  - No need to install Oracle Instant Client.
  - Uses pure Python for the connection.
  - Ideal for simple applications and cloud environments.
- **Thick Mode:**
  - Requires Oracle Instant Client or full Oracle Client installation.
  - Offers more features like connection pooling and advanced data types.
  - Activate it with:
    ```
    import oracledb
    oracledb.init_oracle_client(lib_dir="/path/to/instant/client")
    ```
    In most cases, Thin Mode works perfectly with Autonomous Databases.

1.3.3 3. Connecting to Oracle

Basic Connection (Thin Mode)

```
import oracledb

connection = oracledb.connect(
    user="your_username",
    password="your_password",
    dsn="hostname:1521/service_name"
)

print("Connected to Oracle DB:", connection.version)
connection.close()
```

Autonomous DB with Wallet If you're using Oracle Autonomous Database, download the wallet and extract its contents.

```
connection = oracledb.connect(
    user="admin",
    password="your_password",
    dsn="mydb_high",
    config_dir="/path/to/wallet"
)
```

This securely connects to the cloud database using SSL.

1.3.4 4. Running Queries

Once connected, you can run SQL queries using cursors.

Creating Tables

```
cursor = connection.cursor()
cursor.execute("""
CREATE TABLE universities (
    id NUMBER GENERATED BY DEFAULT AS IDENTITY,
    name VARCHAR2(255) NOT NULL,
    PRIMARY KEY (id)
)
""")
```

Inserting Data You can use **positional placeholders** or **named placeholders** when inserting data.

- **Positional Placeholders**: Use `:1`, `:2`, etc., to represent values in the query.

```
cursor.execute("INSERT INTO universities (name) VALUES (:1)", ["Harvard"])
connection.commit()
```

- **Named Placeholders**: Use `:name`, `:value`, etc., for more clarity.

```
cursor.execute("INSERT INTO universities (name) VALUES (:name)", {"name": "MIT"})
connection.commit()
```

1.3.5 Inserting Multiple Rows with execute_many

When inserting multiple records, use `executemany()` to improve performance:

```
data = [
    ("Stanford"),
    ("Oxford"),
    ("Cambridge")
]
cursor.executemany("INSERT INTO universities (name) VALUES (:1)", data)
connection.commit()
```

This reduces the number of requests to the database.

Fetching Data

```
cursor.execute("SELECT * FROM universities")
for row in cursor:
    print(row)
```

1.3.6 5. Handling Errors

Always handle database operations with proper error handling.

```
try:
    connection = oracledb.connect(...)
except oracledb.DatabaseError as e:
    print("Error:", e)
```

1.3.7 6. Closing Connections

Closing the cursor and connection is crucial to avoid resource leaks.

```
cursor.close()
connection.close()
```

1.4 Other DBMS

DB-API 2.0 has been widely adopted by Python database libraries and modules, making it possible to work with various databases using a consistent programming interface. Developers can choose from a variety of database drivers and modules that implement this standard to work with their preferred DBMS.

- MySQL
- PostgrsSQL
- SQL Server: pyodbc / cTDS
- MariaDB
- GaussDB with PyGreSQL

1.5 Exercises

1. Create a new table `books` with the following fields: `id`, `title`, `author`, `year`, `genre`, `pages`, `publisher`.
2. Load data from `books.csv` and insert it into the table.
3. Run a `SELECT` query to check that the data was inserted.
4. Update all rows in the table with randomly generated years (use `random.randint` and an interval like 1900-2024).
5. Export data from the `books` table to a JSON file.