Topic 01: Database API (DB-API) with SQLite

Overview

This topic introduces the foundational concepts of database programming using Python's **DB-API 2.0** specification with **SQLite**. Working through an interactive Jupyter notebook, you'll learn how to interact with relational databases using SQL commands from Python code. This represents the most fundamental level of database programming - direct SQL execution without abstraction layers or ORMs (Object-Relational Mappers).

What is DB-API?

DB-API 2.0 (PEP 249) is Python's standard specification for database interface modules. It defines a common API that database libraries implement, allowing developers to write database code that can potentially work with different database systems with minimal changes.

Key Components:

1. **Connection Object** - Represents a connection to a database
2. **Cursor Object** - Executes SQL statements and retrieves results
3. **Transaction Management** - Commit or rollback changes
4. **Parameterized Queries** - Safe SQL with placeholders for values

Why SQLite?

SQLite is an excellent starting point for learning database concepts:

- **Serverless**: No separate database server required
- **File-based**: Entire database stored in a single file
- **Zero configuration**: No setup or administration needed
- **Full-featured**: Supports most SQL operations
- **Built into Python**: No additional installation required
- **Portable**: Database file can be easily shared or moved

SQLite is ideal for development, testing, embedded applications, and learning.

Core Database Concepts

1. Connections and Cursors

Connection: Establishes a link to the database file.

```
connection = sqlite3.connect("pets.db")
```

- Creates the database file if it doesn't exist
- Opens the file if it already exists

- Returns a connection object for further operations

Cursor: Executes SQL commands and retrieves results.

```
cursor = connection.cursor()
```

- A cursor is your "pointer" into the database
- Executes SQL statements
- Fetches query results
- Multiple cursors can share one connection

Why separate Connection and Cursor?

- Connection manages the database file and transactions
- Cursor manages individual SQL operations
- This separation allows multiple concurrent operations on one connection

2. Creating Tables (DDL - Data Definition Language)

Tables define the structure of your data with:

- **Column names**: Identify each piece of data
- **Data types**: Specify what kind of data each column holds
- **Constraints**: Rules that data must follow

Our pets table structure:

```
cursor.execute("""
    create table if not exists pets (
        id integer primary key autoincrement,
        name text not null,
        type text not null,
        age integer,
        owner text
    )
""")
```

Breaking down the schema:

- `id integer primary key autoincrement`
    - **Primary key**: Uniquely identifies each row
    - **Autoincrement**: SQLite automatically generates sequential IDs
    - Starting at 1, incrementing for each new row
- `name text not null`
    - **text**: String data type
    - **not null**: This field is required, cannot be empty

- `type text not null`
    - Stores the pet type (e.g., "dog", "cat")
    - Required field
- `age integer`
    - Whole number for the pet's age
    - Optional (can be NULL if unknown)
- `owner text`
    - Owner's name
    - Optional field

Best practice: Use `create table if not exists` to avoid errors when running the code multiple times.

3. Inserting Data (DML - Data Manipulation Language)

Single insert with parameterized query:

```
name = "dorothy"
cursor.execute("""
    insert
        into pets(name, type, age, owner)
        values (?,?,?,?)
    """, (name, "dog", 9, "greg"))
connection.commit()
```

Key principles:

- **Parameterized queries** use `?` placeholders instead of string concatenation
- Values are passed as a tuple in the second argument
- This approach **prevents SQL injection attacks**
- The `id` field is omitted - SQLite auto-generates it

Why parameterize?

[!] **UNSAFE (SQL Injection vulnerable):**

```
name = "dorothy"
cursor.execute(f"insert into pets(name, type, age, owner) values
('{name}', 'dog', 9, 'greg')")
```

✅ **SAFE (Parameterized):**

```
name = "dorothy"
cursor.execute("insert into pets(name, type, age, owner) values
(?,?,?,?)", (name, "dog", 9, "greg"))
```

If `name` contains malicious SQL, parameterization treats it as plain data, not code.

Batch inserts with loops:

```
for name in ["suzy", "casey", "heidi"]:
    cursor.execute("""
        insert
            into pets(name, type, age, owner)
            values (?,?,?,?)
        """, (name, "dog", 9, "greg"))
    connection.commit()
```

This demonstrates how to efficiently insert multiple records with varying data.

4. Querying Data (SELECT)

Retrieve data with conditions:

```
cursor.execute("""select * from pets where type=?""", ("dog",))
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Query execution flow:

1. `execute()` runs the SQL query
2. `fetchall()` retrieves all matching rows as a list of tuples
3. Each tuple represents one row with values in column order

Output format:

```
(1, 'dorothy', 'dog', 9, 'greg')
(2, 'suzy', 'dog', 9, 'greg')
(3, 'casey', 'dog', 9, 'greg')
(4, 'heidi', 'dog', 9, 'greg')
```

Each tuple contains: `(id, name, type, age, owner)`

Other fetch methods:

- `fetchone()` - Returns a single row (or None)

- `fetchmany(n)` - Returns n rows
- `fetchall()` - Returns all remaining rows

5. Updating Data (UPDATE)

Modify existing records:

```
cursor.execute("""
    update pets
    set age = ?
    where name = ?
    """, (11, "sandy"))
connection.commit()
```

Update structure:

- `update [table]` - Specifies which table to modify
- `set [column] = ?` - What to change
- `where [condition]` - Which rows to modify

[!] Critical: Always include a `WHERE` clause unless you intend to update **all** rows. Without it:

```
# This updates EVERY row in the table!
cursor.execute("update pets set age = 11")
```

6. Deleting Data (DELETE)

Remove records:

```
cursor.execute("""
    delete from pets
    where type = ?
    """, ("cat",))
connection.commit()
```

[!] Critical: Like UPDATE, always include a `WHERE` clause unless you want to delete **all** rows:

```
# This deletes EVERYTHING!
cursor.execute("delete from pets")
```

7. Transaction Management with Commit

```
connection.commit()
```

What is a transaction?

A transaction is a unit of database work that either:

- **Completes entirely** (all changes saved), or
- **Fails entirely** (no changes saved)

Why commit?

- SQLite operates in **transaction mode** by default
- INSERT, UPDATE, DELETE changes are temporary until committed
- `commit()` makes changes permanent to the database file
- Without commit, changes are lost when the connection closes

Transaction benefits:

- **Atomicity**: All operations succeed or none do
- **Consistency**: Database moves from one valid state to another
- **Isolation**: Transactions don't interfere with each other
- **Durability**: Committed changes survive crashes

Rollback (not shown in this notebook):

```
connection.rollback()  # Undo all uncommitted changes
```

CRUD Operations Summary

The notebook demonstrates the four fundamental database operations:

Operation	SQL Command	Purpose
Create	INSERT	Add new records
Read	SELECT	Query existing records
Update	UPDATE	Modify existing records
Delete	DELETE	Remove records

These form the foundation of all database applications.

SQL Injection Prevention

One of the most important lessons in this topic is **parameterized queries**.

The problem:

```
# DANGEROUS!
pet_type = request.get('type')  # User input
```

```python
cursor.execute(f"select * from pets where type='{pet_type}'")
```

A malicious user could input: `dog' OR '1'='1`

Resulting SQL: `select * from pets where type='dog' OR '1'='1'`

This returns ALL pets, bypassing your intended filter.

The solution:

```python
# SAFE!
pet_type = request.get('type')
cursor.execute("select * from pets where type=?", (pet_type,))
```

The database treats the entire input as a single string value, not executable SQL.

Rule of thumb: NEVER concatenate user input into SQL strings. Always use parameterization.

Working with the Notebook

Cell Execution Order Matters

Jupyter notebooks allow executing cells in any order, but for this topic:

1. **Import sqlite3** (Cell 1)
2. **Create connection and cursor** (Cell 2)
3. **Create table** (Cell 3)
4. **Insert data** (Cells 4-6)
5. **Query/Update/Delete** (Cells 7-9)

Running cells out of order may cause errors (e.g., querying before inserting data).

Modifying and Experimenting

The notebook format encourages experimentation:

- Change pet names, types, ages
- Add different WHERE conditions
- Try different SELECT queries
- Observe how data changes persist

Example modifications to try:

```python
# Query by owner
cursor.execute("select * from pets where owner=?", ("greg",))

# Query with multiple conditions
cursor.execute("select * from pets where type=? and age > ?", ("dog", 5))
```

```
# Count records
cursor.execute("select count(*) from pets")
count = cursor.fetchone()[0]
print(f"Total pets: {count}")

# Select specific columns
cursor.execute("select name, age from pets where type=?", ("dog",))
```

Database Persistence

The `pets.db` file contains all your data:

- Persists between notebook runs
- Can be opened by other SQLite tools
- Can be deleted to start fresh
- Can be copied or shared

View with command line:

```
sqlite3 pets.db
sqlite> .tables
sqlite> select * from pets;
sqlite> .quit
```

Key Concepts and Terminology

- **DB-API**: Python's standard database interface specification
- **SQLite**: Serverless, file-based relational database
- **Connection**: Database session object
- **Cursor**: Executes SQL and retrieves results
- **Parameterized Query**: Safe SQL with placeholders (?)
- **Commit**: Save changes permanently
- **Rollback**: Undo uncommitted changes
- **CRUD**: Create, Read, Update, Delete operations
- **SQL Injection**: Security vulnerability from unsafe query construction
- **Transaction**: Atomic unit of database work
- **Primary Key**: Unique identifier for each row
- **Autoincrement**: Automatic ID generation

Best Practices Demonstrated

1. ☑️ Use `create table if not exists` for idempotent scripts
2. ☑️ Always use parameterized queries (never string concatenation)
3. ☑️ Include WHERE clauses in UPDATE and DELETE to avoid accidents
4. ☑️ Commit after data modifications (INSERT, UPDATE, DELETE)
5. ☑️ Use meaningful column names and appropriate data types
6. ☑️ Define primary keys for record identification

7. ☑ Use NOT NULL constraint for required fields

Limitations and Considerations

What this approach doesn't handle:

- **Connection management**: No proper closing or error handling
- **Complex queries**: No joins, subqueries, or aggregations (yet)
- **Data validation**: No checks before insertion
- **Error handling**: No try/except blocks for database errors
- **Connection pooling**: Each operation uses the same connection
- **Object mapping**: Data comes back as tuples, not objects

These concerns are addressed in later topics with abstractions and ORMs.

Getting Started

1. Open the Notebook

In VS Code with Jupyter extension:

```
# The notebook is ready to use
open topic-01-db-api/db-api.ipynb
```

2. Run Cells Sequentially

Execute each cell in order from top to bottom using Shift+Enter.

3. Examine the Database

After running the notebook:

```
ls -lh topic-01-db-api/pets.db
```

You'll see the SQLite database file has been created.

4. Experiment

Modify queries, add new inserts, try different conditions, and observe results.

Connection to Future Topics

This foundational topic establishes concepts that later topics build upon:

- **Topic 02**: Integrates DB-API with Flask web applications
- **Topic 03**: Introduces database abstraction layers to reduce code duplication
- **Topic 04**: Explores primary keys, foreign keys, and JOIN operations
- **Topic 05**: Uses an ORM (Peewee) to abstract away raw SQL

- **Topic 06**: Introduces the dataset library for even simpler database operations

Key Takeaways

1. **DB-API provides a standard way** to interact with databases in Python
2. **SQLite is perfect for learning** - no server, no configuration
3. **Connections and cursors** are the fundamental objects for database work
4. **Parameterized queries are essential** for security (prevent SQL injection)
5. **Commit is required** to make changes permanent
6. **CRUD operations** are the building blocks of database applications
7. **Raw SQL gives complete control** but requires careful coding
8. **Transaction management** ensures data consistency and integrity

Understanding these fundamentals prepares you for more advanced database programming patterns and tools introduced in subsequent topics.