Connecting Python to Databases

1. What Does It Mean to Connect Python to a Database?

A Python program can store data in variables, lists, dictionaries, and files. But these are not always enough for real applications.

Imagine a haas.dev application storing thousands of resources:

```
Resource
  ↓
Title
Description
Category
PDF link
Difficulty
Tags
```

If this information is stored only in Python variables, it disappears when the program stops.

A database gives the application **persistent storage**.

The basic idea is:

```
Python Application
       ↓
Database Driver / Library
       ↓
Database
       ↓
Tables
       ↓
Rows and Columns
```

Python sends commands to the database, and the database returns results.

---

2. Why Applications Need Databases

Databases are useful when an application needs to:

- store information permanently
- retrieve specific records
- update existing information
- delete records
- search large amounts of data

- handle multiple users
- maintain relationships between data
- preserve data between program runs

For example, an online learning platform might store:

```
Users
Resources
Categories
Quizzes
Comments
Progress
Bookmarks
```

Instead of keeping everything inside Python code, the application stores this information in a database.

---

3. Python and Databases

Python does not itself act as a database.

Instead, Python communicates with a database through a **database driver or library**.

For example:

```
Python
   ↓
sqlite3
   ↓
SQLite
```

Or:

```
Python
   ↓
PostgreSQL driver
   ↓
PostgreSQL
```

Or:

```
Python
   ↓
SQLAlchemy
   ↓
```

```
Database driver
   ↓
PostgreSQL / MySQL / SQLite
```

This distinction is important.

Python is the application language.

The database stores the data.

The driver or library allows them to communicate.

---

4. Common Databases Used With Python

Python can work with many database systems.

SQLite

SQLite is lightweight and does not require a separate database server.

It stores the database in a file.

```
app/
    main.py
    database.db
```

Python includes the `sqlite3` module in its standard library.

That makes SQLite a useful database for learning, prototypes, small applications, and local storage.

PostgreSQL

PostgreSQL is a powerful relational database commonly used for production applications.

A typical architecture is:

```
Python Application
       ↓
PostgreSQL Driver
       ↓
PostgreSQL Server
```

MySQL

MySQL is another widely used relational database.

Python applications can communicate with it through an appropriate driver or database library.

---

5. SQLite Is a Good Starting Point

For learning database connections, SQLite is convenient because Python provides the `sqlite3` module directly.

You do not need to install a separate database server.

```
import sqlite3
```

Then connect:

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

If `app.db` does not exist, SQLite can create the database file.

---

6. The Database Connection

The connection represents communication between Python and the database.

```
import sqlite3

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

Think of it as opening a communication channel:

```
Python
  |
  | connection
  ↓
SQLite database
```

When you finish working with the database, close the connection:

```
connection.close()
```

A connection can also be managed using a context manager when appropriate.

---

7. Cursor

A cursor can be used to execute SQL statements and retrieve results.

```
import sqlite3

connection = sqlite3.connect("app.db")

cursor = connection.cursor()

cursor.execute("SELECT * FROM resources")
```

```
connection.close()
```

The cursor acts as an interface through which SQL statements can be executed.

Conceptually:

```
Python
   ↓
Connection
   ↓
Cursor
   ↓
SQL
   ↓
Database
```

The `sqlite3` connection also provides shortcut methods such as `execute()` when an explicit cursor is unnecessary.

---

8. Creating a Table

Before storing data, we need a table.

```
import sqlite3

connection = sqlite3.connect("app.db")

cursor = connection.cursor()

cursor.execute("""
    CREATE TABLE IF NOT EXISTS resources (
        id INTEGER PRIMARY KEY,
        title TEXT NOT NULL,
        category TEXT,
        difficulty TEXT
    )
""")

connection.commit()
connection.close()
```

The table contains:

```
resources
--------------------------------
id
title
category
difficulty
```

---

9. Understanding SQL in Python

Python is not replacing SQL.

Python is sending SQL instructions to the database.

For example:

```
cursor.execute(
    "SELECT * FROM resources"
)
```

The SQL part is:

```
SELECT * FROM resources;
```

Python controls the application.

SQL communicates what should happen inside the relational database.

---

10. Inserting Data

We can insert a resource:

```
cursor.execute("""
    INSERT INTO resources
    (title, category, difficulty)
    VALUES (?, ?, ?)
""", (
    "Python Functions",
    "Python",
    "Beginner"
))
```

Then save the transaction:

```
connection.commit()
```

Complete example:

```
import sqlite3

connection = sqlite3.connect("app.db")

cursor = connection.cursor()

cursor.execute("""
    INSERT INTO resources
    (title, category, difficulty)
    VALUES (?, ?, ?)
""", (
    "Python Functions",
    "Python",
    "Beginner"
))

connection.commit()
connection.close()
```

---

11. Why `commit()` Matters

A database operation that changes data may be part of a transaction.

For example:

```
cursor.execute("""
    INSERT INTO resources
    (title, category, difficulty)
    VALUES (?, ?, ?)
""", (
    "Python Functions",
    "Python",
    "Beginner"
))
```

The change should then be committed:

```
connection.commit()
```

Without committing when required, changes may not be persisted as expected.

A useful mental model is:

```
execute()
   ↓
```

```
change requested
   ↓
commit()
   ↓
change saved
```

If something goes wrong, a transaction can also be rolled back:

```
connection.rollback()
```

---

12. Reading Data

Use `SELECT` to retrieve records.

```
cursor.execute(
    "SELECT * FROM resources"
)
```

Retrieve one row:

```
resource = cursor.fetchone()

print(resource)
```

Retrieve multiple rows:

```
resources = cursor.fetchall()

for resource in resources:
    print(resource)
```

A returned row might look like:

```
(1, 'Python Functions', 'Python', 'Beginner')
```

---

13. Selecting Specific Columns

Instead of:

```
SELECT *
```

you can request only the information you need.

```
cursor.execute("""
    SELECT title, difficulty
```

```
    FROM resources
""")
```

Then:

```
for title, difficulty in cursor.fetchall():
    print(title, difficulty)
```

This is generally clearer than retrieving unnecessary columns.

---

14. Filtering Data

SQL can filter records.

```
cursor.execute("""
    SELECT title
    FROM resources
    WHERE category = ?
""", ("Python",))
```

Then:

```
for row in cursor.fetchall():
    print(row[0])
```

The Python value is passed separately from the SQL statement.

---

15. Parameterized Queries

This is one of the most important database concepts.

Do **not** construct SQL using string concatenation.

Bad:

```
username = input("Username: ")

query = (
    "SELECT * FROM users "
    + "WHERE username = '" + username + "'"
)

cursor.execute(query)
```

This can create SQL injection vulnerabilities.

Instead, use placeholders:

```
username = input("Username: ")

cursor.execute(
    "SELECT * FROM users WHERE username = ?",
    (username,)
)
```

The Python `sqlite3` documentation specifically recommends parameter substitution instead of assembling SQL with Python string operations.
Mental model:

```
SQL structure
      +
Python values
      ↓
Parameterized query
```

Never treat user input as SQL code.

---

16. Named Parameters

SQLite also supports named placeholders.

```
cursor.execute("""
    SELECT *
    FROM resources
    WHERE category = :category
""", {
    "category": "Python"
})
```

This can be useful when queries have several parameters.

---

17. Updating Data

Use `UPDATE`.

```
cursor.execute("""
    UPDATE resources
    SET difficulty = ?
    WHERE id = ?
""", (
    "Intermediate",
```

```
    1
))

connection.commit()
```

The query means:

```
Find resource with id 1
        ↓
Change difficulty
        ↓
Intermediate
```

Always be careful with the `WHERE` clause.

This is dangerous:

```
UPDATE resources
SET difficulty = 'Intermediate';
```

It updates every row.

---

18. Deleting Data

Use `DELETE`.

```
cursor.execute("""
    DELETE FROM resources
    WHERE id = ?
""", (1,))

connection.commit()
```

Again, the `WHERE` clause matters.

This:

```
DELETE FROM resources;
```

can remove every record from the table.

---

19. CRUD

These four operations form the foundation of many database applications.

```
C → Create
R → Read
```

```
U → Update
D → Delete
```

Operation	SQL
Create	`INSERT`
Read	`SELECT`
Update	`UPDATE`
Delete	`DELETE`

For a resource system:

```
Create → add resource
Read   → show resources
Update → edit resource
Delete → remove resource
```

---

20. Inserting Multiple Records

If many records need to be inserted, `executemany()` can be useful.

```
resources = [
    ("Python Functions", "Python", "Beginner"),
    ("Python Classes", "Python", "Intermediate"),
    ("REST APIs", "Backend", "Intermediate")
]

cursor.executemany("""
    INSERT INTO resources
    (title, category, difficulty)
    VALUES (?, ?, ?)
""", resources)

connection.commit()
```

Python's `sqlite3` documentation provides `executemany()` specifically for repeatedly executing parameterized DML statements.

---

21. A Complete SQLite Example

Here is a small database application:

```
import sqlite3

connection = sqlite3.connect("resources.db")

cursor = connection.cursor()

cursor.execute("""
    CREATE TABLE IF NOT EXISTS resources (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        category TEXT NOT NULL,
        difficulty TEXT NOT NULL
    )
""")

cursor.execute("""
    INSERT INTO resources
    (title, category, difficulty)
    VALUES (?, ?, ?)
""", (
    "Connecting Python to Databases",
    "Python",
    "Intermediate"
))

connection.commit()

cursor.execute("""
    SELECT id, title, category, difficulty
    FROM resources
""")

resources = cursor.fetchall()

for resource in resources:
    print(resource)

connection.close()
```

The flow is:

```
Connect
   ↓
Create table
   ↓
Insert data
   ↓
Commit
   ↓
Read data
   ↓
Close
```

---

22. Using Context Managers

Python's database code can be made safer and cleaner with `with`.

For example:

```
import sqlite3

with sqlite3.connect("resources.db") as connection:
    connection.execute("""
        INSERT INTO resources
        (title, category, difficulty)
        VALUES (?, ?, ?)
    """, (
        "SQL Basics",
        "Databases",
        "Beginner"
    ))
```

The context manager helps manage transaction behavior.

You should still understand what is happening rather than treating `with` as magic.

---

23. Handling Database Errors

Database operations can fail.

For example:

- invalid SQL
- missing table

- constraint violation
- invalid database path
- locked database
- connection problems

SQLite provides database-related exceptions.

```
import sqlite3

try:
    connection = sqlite3.connect("resources.db")

    connection.execute("""
        INSERT INTO resources
        (title, category, difficulty)
        VALUES (?, ?, ?)
    """, (
        "Python",
        "Programming",
        "Beginner"
    ))

    connection.commit()

except sqlite3.Error as error:
    print("Database error:", error)

finally:
    connection.close()
```

Error handling should be specific enough to understand what failed.

---

24. Transactions

A transaction groups database operations into a logical unit.

Imagine transferring money:

```
Remove money from Account A
        ↓
Add money to Account B
```

Both operations should succeed together.

If the second operation fails, you may need to undo the first.

That is where transactions become important.

Conceptually:

```
BEGIN
 ↓
Operation 1
 ↓
Operation 2
 ↓
COMMIT
```

If something fails:

```
BEGIN
 ↓
Operation 1
 ↓
Operation 2 ❌
 ↓
ROLLBACK
```

Transactions become especially important in applications where several database changes must remain consistent.

---

25. Primary Keys

A database table commonly has a primary key.

Example:

```
CREATE TABLE resources (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL
);
```

The `id` identifies a particular record.

For example:

```
id    title
----------------------
1     Python Functions
2     Python Classes
3     REST APIs
```

The primary key lets Python identify a specific resource.

---

26. Constraints

Databases can enforce rules.

For example:

```
title TEXT NOT NULL
```

means the title cannot be `NULL`.

Other common constraints include:

```
PRIMARY KEY
NOT NULL
UNIQUE
FOREIGN KEY
CHECK
```

Example:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    email TEXT UNIQUE NOT NULL
);
```

The database now helps enforce the rule that two users should not have the same email.

---

27. Relationships Between Tables

Real applications usually have multiple tables.

For example:

```
users
resources
categories
```

A resource might belong to a category.

```
Category
   │
   └── Resources
```

This leads to relational database concepts such as:

- primary keys

- foreign keys
- one-to-one relationships
- one-to-many relationships
- many-to-many relationships
- joins

Example:

```
CREATE TABLE categories (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE resources (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL,
    category_id INTEGER,
    FOREIGN KEY (category_id)
        REFERENCES categories(id)
);
```

Now `category_id` connects the resource to a category.

---

28. SQL JOINs

Suppose we have:

```
categories
----------------
1   Python
2   Backend
```

and:

```
resources
------------------------------
1   Functions       1
2   REST API        2
```

A JOIN can combine information from both tables.

```
SELECT
    resources.title,
    categories.name
```

```
FROM resources
JOIN categories
    ON resources.category_id = categories.id;
```

Python can execute this query normally:

```
cursor.execute("""
    SELECT
        resources.title,
        categories.name
    FROM resources
    JOIN categories
        ON resources.category_id = categories.id
""")
```

---

29. Python Objects vs Database Rows

Python may represent a resource as an object:

```
class Resource:
    def __init__(self, title, category):
        self.title = title
        self.category = category
```

The database might represent the same resource as:

```
id | title             | category
---------------------------------
1  | Python Functions  | Python
```

There are therefore two worlds:

```
Python world
     ↓
Objects

Database world
     ↓
Rows and tables
```

A database integration layer connects these worlds.

---

30. Database Drivers

For larger databases, Python generally uses a database driver.

Conceptually:

```
Python Application
       ↓
Database Driver
       ↓
Database Server
```

Examples include drivers for:

```
PostgreSQL
MySQL
SQLite
```

The exact package depends on the database.

This is why database connection code can differ between database systems.

---

31. SQLAlchemy

For larger Python applications, developers often use a library such as SQLAlchemy.

SQLAlchemy provides database connectivity and higher-level abstractions.

Its architecture can look like:

```
Python Application
       ↓
SQLAlchemy
       ↓
Database Driver
       ↓
Database
```

SQLAlchemy applications commonly create an `Engine`, which acts as a central source for database connections and connection pooling.

---

32. Creating a SQLAlchemy Engine

Install SQLAlchemy:

```
pip install sqlalchemy
```

Then:

```
from sqlalchemy import create_engine
```

```
engine = create_engine(
    "sqlite:///resources.db"
)
```

The engine represents the application's configured connection to the database.

A typical application creates an engine once rather than creating a new engine for every function call.

---

33. Executing SQL With SQLAlchemy

```
from sqlalchemy import create_engine, text

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

with engine.connect() as connection:
    result = connection.execute(
        text("SELECT * FROM resources")
    )

    for row in result:
        print(row)
```

The `with` statement gives the connection a clear lifecycle.

SQLAlchemy recommends scoped connection usage rather than leaving connection resources open indefinitely.

---

34. SQLAlchemy ORM

SQLAlchemy also provides an ORM.

ORM means:

Object Relational Mapper

Instead of thinking only in terms of:

```
tables
rows
SQL
```

you can work with Python classes representing database records.

Conceptually:

```
Python Class
     ↕
Database Table
```

For example:

```
class Resource:
    ...
```

could represent:

```
resources table
```

The ORM can then help map Python objects to database data.

---

35. Engine vs Connection vs Session

When learning SQLAlchemy, these terms can be confusing.

Engine

The engine manages access to the database and provides connections.

```
Engine
  ↓
Connections
```

Connection

A connection represents an active database interaction.

```
Engine
  ↓
Connection
  ↓
SQL
```

Session

The ORM `Session` is the main interface for working with ORM-mapped objects and manages database conversations and transactions.

Conceptually:

```
Application
     ↓
Session
     ↓
```

```
Engine
     ↓
Connection
     ↓
Database
```

---

36. When Should You Use SQLite?

SQLite is useful for:

- learning
- prototypes
- local tools
- small applications
- testing
- embedded applications

Example:

```
Personal CLI tool
       ↓
SQLite
```

You don't necessarily need PostgreSQL for every project.

---

37. When Should You Use PostgreSQL or MySQL?

A server-based relational database becomes more appropriate when an application needs things such as:

- multiple concurrent users
- centralized database access
- larger production workloads
- advanced database features
- production deployment
- stronger operational tooling

A common production architecture is:

```
Frontend
   ↓
Python API
```

```
   ↓
PostgreSQL
```

For example:

```
React
  ↓
FastAPI
  ↓
PostgreSQL
```

---

38. Database Connection Strings

Libraries often use a database URL.

For example:

```
sqlite:///resources.db
```

A PostgreSQL connection URL can contain:

```
postgresql://username:password@host/database
```

Conceptually:

```
database type
      ↓
credentials
      ↓
host
      ↓
database name
```

Connection URLs should be treated as configuration rather than hard-coded secrets.

---

39. Environment Variables

Never hard-code production database passwords in source code.

Bad:

```
DATABASE_URL = (
    "postgresql://admin:secret123@localhost/app"
)
```

Better:

```
import os

DATABASE_URL = os.environ["DATABASE_URL"]
```

For local development, a `.env` file may be used with an appropriate environment-variable library.

This keeps sensitive configuration outside the source code.

---

40. Python + FastAPI + Database

A typical backend architecture might look like:

```
Client
  ↓
FastAPI
  ↓
Service Layer
  ↓
Database Access Layer
  ↓
PostgreSQL
```

For example:

```
GET /resources
       ↓
FastAPI route
       ↓
Resource service
       ↓
Database query
       ↓
PostgreSQL
       ↓
Rows
       ↓
Python objects/data
       ↓
JSON response
```

This is how database knowledge connects with the REST API concepts covered earlier.

---

41. Database Layer Design

Avoid putting every database operation directly inside API routes.

Instead of:

```
@app.get("/resources")
def get_resources():
    # huge amount of SQL here
    ...
```

a larger application can separate responsibilities:

```
routes/
services/
repositories/
models/
database/
```

For example:

```
Route
 ↓
Service
 ↓
Repository
 ↓
Database
```

Each layer has a clearer responsibility.

---

42. Repository Pattern

A repository can centralize database operations.

```
class ResourceRepository:

    def get_all(self):
        ...

    def get_by_id(self, resource_id):
        ...

    def create(self, resource):
        ...
```

```
    def delete(self, resource_id):
        ...
```

The service can then use it:

```
class ResourceService:

    def __init__(self, repository):
        self.repository = repository

    def get_resources(self):
        return self.repository.get_all()
```

This can make the application easier to test and maintain.

---

43. Common Database Mistakes

Mistake 1: Building SQL with user input

Bad:

```
query = (
    "SELECT * FROM users WHERE name = '"
    + username
    + "'"
)
```

Use parameters.

Mistake 2: Forgetting `commit()`

```
cursor.execute(...)
```

does not automatically mean your application has completed the intended transaction.

Understand when changes need to be committed.

Mistake 3: Leaving connections open

Bad:

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

# application continues forever
```

Manage connection lifecycles.

Mistake 4: Updating every row accidentally

Bad:

```
UPDATE users
SET role = 'admin';
```

Always verify your `WHERE` condition.

Mistake 5: Deleting everything accidentally

Bad:

```
DELETE FROM users;
```

Check the query before executing destructive operations.

Mistake 6: Hard-coding credentials

Never put production passwords directly into Git-tracked source code.

Mistake 7: Using raw SQL everywhere

Raw SQL is useful and important to understand.

But large applications may benefit from a structured database layer or ORM.

---

44. Database Security

Database security begins with basic practices:

```
Validate input
      ↓
Parameterized queries
      ↓
Secure credentials
      ↓
Least privilege
      ↓
Encrypted connections where appropriate
      ↓
Backups
      ↓
Monitoring
```

The most important rule for SQL injection prevention is:

> Treat user input as data, not SQL code.

---

45. A Practical Database Architecture

For a Python backend:

```
               Client
                 ↓
              FastAPI
                 ↓
          Route / Endpoint
                 ↓
             Service
                 ↓
           Repository
                 ↓
        SQLAlchemy / Driver
                 ↓
            Database
```

This gives each component a clear job.

---

46. haas.dev Example

Imagine haas.dev stores resources in a database.

A resource might contain:

```
id
title
description
category
difficulty
file_url
```

The database might look like:

```
resources
-----------------------------------------------------
id | title | description | category | difficulty
```

A user opens:

```
GET /resources
```

The request travels through:

```
Browser
   ↓
FastAPI
   ↓
Resource Service
   ↓
Resource Repository
   ↓
Database
   ↓
Resources
   ↓
JSON
   ↓
Browser
```

The frontend does not need to know how the database works.

It only communicates with the API.

---

47. Mini Project: Resource Database

Build a small resource manager.

Requirements

Create a SQLite database containing:

```
resources
```

Columns:

```
id
title
category
difficulty
```

Implement:

```
add_resource()
get_resources()
get_resource()
update_resource()
delete_resource()
```

Example:

```
def add_resource(title, category, difficulty):
    ...
```

Then:

```
add_resource(
    "Python Databases",
    "Python",
    "Intermediate"
)
```

And:

```
resources = get_resources()

for resource in resources:
    print(resource)
```

This gives you a complete CRUD database application.

---

48. Five Minute Challenge

Create:

```
students.db
```

Create a table:

```
students
```

with:

```
id
name
age
grade
```

Then:

1. Insert three students.
2. Retrieve all students.
3. Retrieve only students from grade 9.
4. Update one student's age.
5. Delete one student.

6. Print the remaining records.

Use parameterized queries for every value.

---

49. Exercises

Exercise 1

Create a `books` table:

```
id
title
author
year
```

Insert five books.

Retrieve all books published after 2020.

Exercise 2

Create a `tasks` table:

```
id
title
completed
```

Implement:

```
add_task()
get_tasks()
complete_task()
delete_task()
```

Exercise 3

Create two tables:

```
categories
resources
```

Connect them with a foreign key.

Write a JOIN query that displays:

```
resource title
category name
```

Exercise 4

Build a CLI resource manager:

```
1. Add resource
2. View resources
3. Search resources
4. Update resource
5. Delete resource
6. Exit
```

Store all data in SQLite.

---

50. Knowledge Check

1. Why do applications use databases?

A. To make Python syntax shorter

B. To persist and organize data

C. To replace Python

D. To create HTML

Answer: B

---

2. What does CRUD stand for?

Answer:

```
Create
Read
Update
Delete
```

---

3. Why should parameterized queries be used?

Answer: To safely pass values into SQL and help prevent SQL injection.

---

4. What does `commit()` do?

Answer: It commits the current transaction's changes to the database.

---

5. What is a primary key?

Answer: A column or set of columns used to uniquely identify a record.

---

6. What is SQLAlchemy?

Answer: A Python SQL/database toolkit and ORM that provides abstractions for working with relational databases.

---

51. Interview Questions

Beginner

1. What is a database?
2. Why do applications need databases?
3. What is SQLite?
4. How do you connect Python to SQLite?
5. What is a cursor?
6. What does `commit()` do?
7. What is CRUD?
8. What is a primary key?
9. What is a foreign key?
10. What is SQL injection?

Intermediate

11. What is a transaction?
12. What is rollback?
13. Why use parameterized queries?
14. What is a database driver?
15. What is an ORM?
16. What is SQLAlchemy?
17. What is the difference between an engine and a connection in SQLAlchemy?
18. What is a SQLAlchemy Session?
19. When might SQLite be appropriate?
20. Why should database credentials be stored outside source code?

Practical

21. How would you design a database layer for a FastAPI application?
22. How would you safely handle user-provided search input?
23. How would you prevent accidental deletion of all records?
24. How would you structure database code in a larger Python project?
25. How would you handle a failed multi-step database operation?

---

52. Database Mental Model

Remember this:

```
Python
  ↓
Driver / Database Library
  ↓
Connection
  ↓
SQL
  ↓
Database
  ↓
Tables
  ↓
Rows
```

For larger applications:

```
Python
  ↓
SQLAlchemy
  ↓
Driver
  ↓
Database
```

And for an API:

```
Client
  ↓
FastAPI
  ↓
Service
  ↓
Repository
  ↓
SQLAlchemy / Driver
  ↓
Database
```

---

53. Cheat Sheet

Connect to SQLite

```
import sqlite3

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

Create cursor

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

Execute SQL

```
cursor.execute("SELECT * FROM users")
```

Parameterized query

```
cursor.execute(
    "SELECT * FROM users WHERE id = ?",
    (user_id,)
)
```

Insert

```
cursor.execute(
    "INSERT INTO users (name) VALUES (?)",
    (name,)
)
```

Commit

```
connection.commit()
```

Fetch one

```
row = cursor.fetchone()
```

Fetch many

```
rows = cursor.fetchall()
```

Update

```
cursor.execute(
    "UPDATE users SET name = ? WHERE id = ?",
    (name, user_id)
)
```

Delete

```
cursor.execute(
    "DELETE FROM users WHERE id = ?",
    (user_id,)
)
```

Rollback

```
connection.rollback()
```

Close

```
connection.close()
```

SQLAlchemy

```
from sqlalchemy import create_engine

engine = create_engine(
    "sqlite:///app.db"
)
```

---

54. Key Takeaways

- Python applications often need persistent data storage.
- A database stores application data beyond the lifetime of a Python process.
- Python communicates with databases through drivers and database libraries.
- SQLite is a simple database to start with because Python provides `sqlite3`.
- A connection represents communication with the database.
- Cursors can execute SQL and retrieve results.
- CRUD means Create, Read, Update, Delete.
- `commit()` saves transactional changes.
- `rollback()` can undo pending transactional changes.
- Primary keys identify records.
- Foreign keys connect related tables.

- Parameterized queries should be used instead of building SQL with user input.
- SQL injection is a major database security concern.
- SQLite is useful for many small and local applications.
- PostgreSQL and other server databases are commonly used for production systems.
- SQLAlchemy provides higher-level database abstractions and ORM capabilities.
- In larger applications, database access should usually have a clear architectural boundary.
- FastAPI and a database commonly work together through routes, services, repositories, and a database layer.
- Understanding SQL remains important even when using an ORM.

The core idea:

```
Python controls the application.
SQL describes database operations.
The database stores the data.
The database library connects the two.
```

haas.dev

https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.