Lesson 9: Python Database Integration

Duration: 20 minutes
Deliverable: `lesson9_database.py`

🎯 Learning Objectives

By the end of this lesson, you will be able to:

- Connect to SQLite databases from Python
- Execute SQL queries programmatically
- Fetch and process query results
- Use parameterised queries to prevent SQL injection
- Handle database errors gracefully
- Close connections properly

📚 Why Use Python with Databases?

Writing SQL queries in `.sql` files is great for learning, but real applications need to:

- Accept user input
- Process data dynamically
- Display results in custom formats
- Integrate databases with other systems
- Automate database operations

Python's `sqlite3` module lets you do all of this!

🛠️ Part 1: Connecting to a Database (8 minutes)

Step 1: Create Your Python File

1. Navigate to `lessons/` folder
2. Create: `lesson9_database.py`
3. Add header:

```
"""
Lesson 9: Python Database Integration
Student Name: [Your Name]
Date: [Today's Date]

This script demonstrates SQLite database operations in Python
"""

import sqlite3
import sys
from pathlib import Path
```

Step 2: Connect to the Database

```
# Best Practice: Use context managers for database connections
def query_characters():
    """
    Query characters using context manager
    Context managers automatically handle connection closing and commits
    """
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            print(f"✓ Connected to database")
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM characters LIMIT 5")
            results = cursor.fetchall()
            for row in results:
                print(row)
            # Connection automatically closes here
    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
        # Rollback happens automatically with context manager
```

Explanation:

- `with sqlite3.connect()` creates a context manager
- Connection automatically closes when exiting the `with` block
- Commits happen automatically on success
- Rollbacks happen automatically on error
- No need to manually call `conn.close()` or `conn.commit()`

Step 3: Test the Connection

```
# Test the connection
def test_connection():
    """Test database connection using context manager"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            print("✓ Connection successful!")
            print(f"✓ SQLite version: {sqlite3.sqlite_version}")
        print("✓ Connection automatically closed")
    except sqlite3.Error as e:
        print(f"✗ Connection failed: {e}")

# Run the test
if __name__ == "__main__":
    test_connection()
```

Execute your script:

```
python lessons/lesson9_database.py
```

You should see: "✓ Connection successful!" and "✓ Connection automatically closed"

📊 Part 2: Executing Queries (8 minutes)

Step 4: Create a Cursor

A **cursor** is used to execute SQL commands and fetch results.

```
def get_all_characters():
    """Retrieve all characters from the database"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            # Create cursor
            cursor = conn.cursor()

            # Execute query
            cursor.execute("SELECT id, name, species, homeworld FROM characters")

            # Fetch all results
            characters = cursor.fetchall()

            # Display results
            print("\n=== All Characters ===")
            for char in characters:
                print(f"ID: {char[0]}, Name: {char[1]}, Species: {char[2]}, Homeworld

            print(f"\nTotal characters: {len(characters)}")
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error executing query: {e}")
```

Key Concepts:

- `cursor.execute()` runs SQL query
- `cursor.fetchall()` gets all rows as list of tuples
- Each tuple is one row
- Context manager ensures connection closes even if error occurs

Step 5: Different Fetch Methods

```
def demonstrate_fetch_methods():
    """Show different ways to fetch data"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            cursor = conn.cursor()

            # Method 1: fetchone() - Get one row at a time
            print("\n=== fetchone() Demo ===")
            cursor.execute("SELECT name, species FROM characters LIMIT 3")
            row = cursor.fetchone()
            while row:
                print(f"Name: {row[0]}, Species: {row[1]}")
                row = cursor.fetchone()

            # Method 2: fetchmany() - Get specified number of rows
            print("\n=== fetchmany() Demo ===")
            cursor.execute("SELECT name, species FROM characters")
            rows = cursor.fetchmany(5)  # Fetch 5 rows
            for row in rows:
                print(f"Name: {row[0]}, Species: {row[1]}")

            # Method 3: fetchall() - Get all remaining rows
            print("\n=== fetchall() Demo ===")
            cursor.execute("SELECT name, height FROM characters WHERE height IS NOT N
            all_rows = cursor.fetchall()
            print("Top 3 Tallest Characters:")
            for row in all_rows:
                print(f"{row[0]}: {row[1]}cm")
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
```

Step 6: Using Row Factory for Named Columns

```
def get_characters_as_dict():
    """Get results as dictionaries instead of tuples"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            # Set row factory to return dictionary-like objects
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            cursor.execute("SELECT name, species, height FROM characters WHERE height
            characters = cursor.fetchall()

            print("\n=== Characters as Dictionaries ===")
            for char in characters:
                # Access by column name instead of index!
                print(f"{char['name']} ({char['species']}): {char['height']}cm")
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
```

Benefit: Access columns by name instead of remembering index numbers!

🔒 Part 3: Parameterised Queries (4 minutes)

Never build SQL queries using string formatting! It's vulnerable to SQL injection attacks.

❌ DANGEROUS: String Formatting

```
# ⚠️  NEVER DO THIS!
def search_character_unsafe(name):
    """UNSAFE: Vulnerable to SQL injection"""
    conn = create_connection()
    cursor = conn.cursor()

    # This is DANGEROUS!
    query = f"SELECT * FROM characters WHERE name = '{name}'"
    cursor.execute(query)

    result = cursor.fetchall()
    conn.close()
    return result

# Malicious input could execute arbitrary SQL:
# search_character_unsafe("Luke'; DROP TABLE characters; --")
```

✅ SAFE: Parameterised Queries

```
def search_character_safe(name):
    """SAFE: Uses parameterised query"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            # Use ? placeholder and pass parameters as tuple
            cursor.execute("SELECT name, species, homeworld FROM characters WHERE nam

            results = cursor.fetchall()
            return results
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
        return []

def search_by_species(species):
    """Search characters by species using parameterised query"""
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            # Multiple parameters
            cursor.execute("""
                SELECT name, species, height
                FROM characters
                WHERE species = ? AND height IS NOT NULL
                ORDER BY height DESC
            """, (species,))

            results = cursor.fetchall()

            print(f"\n=== {species} Characters ===")
            for char in results:
                print(f"{char['name']}: {char['height']}cm")
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
```

Always use `?` placeholders and pass values as tuple!

🎓 Part 4: Complete Example Functions

Step 7: CRUD Operations

```
def add_character(name, species, homeworld_id, height=None):
    """
    Create: Add a new character to the database
    """
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            cursor = conn.cursor()
            cursor.execute("""
                INSERT INTO characters (name, species, homeworld_id, height)
                VALUES (?, ?, ?, ?)
            """, (name, species, homeworld_id, height))

            # Commit happens automatically with context manager!
            print(f"✓ Added character: {name}")
            return True

    except sqlite3.Error as e:
        print(f"✗ Error adding character: {e}")
        return False
        # Rollback happens automatically

def update_character_height(name, new_height):
    """
    Update: Change a character's height
    """
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            cursor = conn.cursor()
            cursor.execute("""
                UPDATE characters
                SET height = ?
                WHERE name = ?
            """, (new_height, name))

            # Commit happens automatically

            if cursor.rowcount > 0:
                print(f"✓ Updated {cursor.rowcount} character(s)")
                return True
            else:
                print(f"✗ No character found with name: {name}")
                return False

    except sqlite3.Error as e:
        print(f"✗ Error updating character: {e}")
        return False
        # Rollback happens automatically

def delete_character(name):
    """
    Delete: Remove a character from the database
    """
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            cursor = conn.cursor()
            cursor.execute("DELETE FROM characters WHERE name = ?", (name,))

            # Commit happens automatically

            if cursor.rowcount > 0:
                print(f"✓ Deleted character: {name}")
                return True
            else:
                print(f"✗ No character found with name: {name}")
                return False

    except sqlite3.Error as e:
        print(f"✗ Error deleting character: {e}")
        return False
        # Rollback happens automatically

def get_character_statistics():
    """
    Read: Get database statistics
    """
    try:
        with sqlite3.connect('database/starwars.db') as conn:
            cursor = conn.cursor()

            # Get counts
            cursor.execute("SELECT COUNT(*) FROM characters")
            total_chars = cursor.fetchone()[0]

            cursor.execute("SELECT COUNT(DISTINCT species) FROM characters")
            total_species = cursor.fetchone()[0]

            cursor.execute("SELECT AVG(height) FROM characters WHERE height IS NOT NU
            avg_height = cursor.fetchone()[0]

            print("\n=== Database Statistics ===")
            print(f"Total Characters: {total_chars}")
            print(f"Different Species: {total_species}")
            print(f"Average Height: {avg_height:.1f}cm")
            # Connection automatically closes here

    except sqlite3.Error as e:
        print(f"✗ Error: {e}")
```

Step 8: Main Menu

```
def main():
    """Main program menu"""
    print("=" * 50)
    print("STAR WARS DATABASE - PYTHON INTERFACE")
    print("=" * 50)

    # Run demonstrations
    print("\n--- Testing Connection ---")
    test_connection()

    print("\n--- All Characters ---")
    get_all_characters()

    print("\n--- Fetch Methods ---")
    demonstrate_fetch_methods()

    print("\n--- Dictionary Access ---")
    get_characters_as_dict()

    print("\n--- Search by Species ---")
    search_by_species("Human")

    print("\n--- Database Statistics ---")
    get_character_statistics()

    print("\n--- CRUD Operations Demo ---")
    # Add a test character
    add_character("Test Character", "Test Species", 1, 175)

    # Update the character
    update_character_height("Test Character", 180)

    # Delete the character
    delete_character("Test Character")

    print("\n" + "=" * 50)
    print("Demo complete!")
    print("=" * 50)

if __name__ == "__main__":
    main()
```

Run your complete script:

```
python lessons/lesson9_database.py
```

🐛 Common Errors & Troubleshooting

Error: "unable to open database file"

Problem: Wrong database path or file doesn't exist.

Solution:

```
# Check if database exists
from pathlib import Path
db_path = Path(__file__).parent.parent / 'starwars.db'
if not db_path.exists():
    print(f"Database not found at: {db_path}")
```

Error: "no such table"

Problem: Table hasn't been created yet.

Solution: Run your Lesson 1 SQL scripts first to create tables.

Error: "database is locked"

Problem: Another process has the database open (less common with context managers).

Solution:

- Context managers handle this better by closing connections promptly
- Check VSCode SQLite plugin isn't holding a lock
- Ensure all `with` blocks complete properly

IndexError when accessing results

Problem: Trying to access column that doesn't exist.

Solution: Use `row_factory` or check your query:

```
conn.row_factory = sqlite3.Row  # Access by name
# or
print(len(row))  # Check how many columns returned
```

Forgot to commit()

Problem: INSERT/UPDATE/DELETE doesn't save changes.

Solution: Context managers handle commits automatically!

```
# With context managers, commit is automatic on success
with sqlite3.connect('database/starwars.db') as conn:
    cursor = conn.cursor()
    cursor.execute("INSERT INTO...")
    # Commit happens automatically when exiting 'with' block successfully
```

✅ Checkpoint: What You've Learnt

Before moving on, make sure you can:

- ✅ Connect to SQLite database from Python using context managers
- ✅ Create cursors and execute queries
- ✅ Fetch results with fetchone(), fetchmany(), fetchall()
- ✅ Use row_factory for dictionary-like access
- ✅ Write parameterised queries (prevent SQL injection)
- ✅ Perform CRUD operations (Create, Read, Update, Delete)
- ✅ Handle errors with try/except
- ✅ Use context managers for automatic connection/commit/rollback handling

🎯 Challenge Problem (Optional)

Task: Create a function that:

1. Accepts a planet name as input
2. Finds all characters from that planet
3. For each character, finds what vehicles they pilot
4. Displays the results in a formatted way
5. Uses parameterised queries
6. Handles errors gracefully

💾 Save Your Work with Git

```
git status
git add lessons/lesson9_database.py database/starwars.db
git commit -m "Completed Lesson 9: Python SQLite database integration"
git push
```

📖 Key Python Concepts Learnt

Concept	Purpose	Example
`with sqlite3.connect():`	Context manager for connection	`with sqlite3.connect('db.db') as conn:`
`cursor()`	Execute SQL commands	`cursor = conn.cursor()`
`execute()`	Run SQL query	`cursor.execute("SELECT...")`
`fetchone()`	Get one row	`row = cursor.fetchone()`
`fetchall()`	Get all rows	`rows = cursor.fetchall()`
`row_factory`	Access columns by name	`conn.row_factory = sqlite3.Row`
Parameterised query	Prevent SQL injection	`execute("SELECT * WHERE id = ?", (id,))`
Context manager auto-commit	Save changes automatically	Happens when `with` block exits successfully
Context manager auto-rollback	Undo on error	Happens when exception occurs in `with` block
`rowcount`	Number of affected rows	`cursor.rowcount`

🔒 Security Best Practices

1. **Always use parameterised queries**

```python # ✅ SAFE cursor.execute("SELECT * WHERE name = ?", (name,))

# ❌ UNSAFE cursor.execute(f"SELECT * WHERE name = '{name}'") ```

1. **Always use context managers**

`python # ✅ BEST PRACTICE with sqlite3.connect('db.db') as conn: cursor = conn.cursor() # ... operations ... # Connection closes automatically, commit/rollback handled`

1. **Handle errors appropriately** `python try: with sqlite3.connect('db.db') as conn: # ... operations ... except sqlite3.Error as e: print(f"Error: {e}") # Rollback happens automatically`
```

🎉 Excellent Work!

You can now build Python applications that interact with databases! In the final lesson, you'll learn about ORMs and NoSQL databases.

Ready to continue? Move on to `lesson10_instructions.md`

Need Help?

- Print variables to debug
- Check database file exists and has data
- Always close connections
- Read error messages carefully
- Ask your instructor!