CRUD Operations from Python

AiCE Software Engineering Foundations — MySQL Programming

Python is connected to aice_books. Now let's put that connection to work — running every CRUD operation you already know from MySQL directly, but this time triggered from Python code, the same way a real application would.

1. The Pattern Behind Every Operation

Every operation in this lesson follows the same shape: write a SQL string, hand it to cursor.execute(), and — for anything that changes data — call conn.commit() to make the change permanent. Reading data skips the commit step entirely, since nothing is being changed.

2. Reading Data: The R in CRUD

Let's read data from Python. Add this to app.py:

```
cursor.execute("SELECT * FROM books")
results = cursor.fetchall()

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

Run the file, and each book prints as a tuple — id, title, author, genre, price, quantity, date_added. Notice the SQL query written inside the Python code, SELECT * FROM books, is the exact same query you used to read all the data directly in MySQL — the language doesn't change, only who's typing it.

3. Creating Data: The C in CRUD

Now let's insert data from Python:

```
sql = "INSERT INTO books (title, author, genre, price,
quantity, date_added) VALUES (%s, %s, %s, %s, %s, %s)"
```

```
values = ("Homegoing", "Yaa Gyasi", "Fiction", 16.75, 20,
"2025-03-01")

cursor.execute(sql, values)
conn.commit()

print("Book added successfully!")
```

This is the same INSERT INTO query you already know from MySQL. Notice %s is used as a placeholder for each value, with the actual values passed separately in a tuple — never built directly into the SQL string with string formatting or concatenation.

> **Why Parameterized Queries Matter: SQL Injection**
> *Passing values as a separate tuple, instead of pasting them straight into the SQL string, protects your application from SQL injection — a technique where malicious input is crafted to alter what the query actually does. %s placeholders let the MySQL connector handle values safely, no matter what a user types.*

4. Confirming Changes Directly in MySQL

Every change made through Python happens in the exact same aice_books database you were working in directly. To confirm it, open a second terminal, log in to MySQL, and check the table yourself:

```
sudo mysql -u root -p
USE aice_books;
SELECT * FROM books;
```

Before running app.py, "Homegoing" won't be there. After running it, the same SELECT * FROM books; query shows it added — confirming Python is working directly inside the same database, not some separate copy of it.

5. Updating Data: The U in CRUD

Let's change the quantity of "Homegoing" from 20 to 35:

```
sql = "UPDATE books SET quantity = %s WHERE title = %s"
values = (35, "Homegoing")

cursor.execute(sql, values)
conn.commit()

print("Book updated successfully!")
```

Run the file, then confirm in the MySQL terminal with SELECT * FROM books; — quantity is now 35. The WHERE clause matters exactly as much here as it did directly in MySQL.

6. Deleting Data: The D in CRUD

Finally, let's delete "Homegoing":

```
sql = "DELETE FROM books WHERE title = %s"
values = ("Homegoing",)

cursor.execute(sql, values)
conn.commit()

print("Book deleted successfully!")
```

Run the file, then confirm in MySQL — "Homegoing" is gone.

7. Closing Your Cursor and Connection

Always close both when you're done with them:

```
cursor.close()
conn.close()
```

8. Putting It All Together: A Complete CRUD Script

Let's rewrite app.py from scratch, performing all four CRUD operations in a single, complete script:

```
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="your_password",
    database="aice_books"
)
cursor = conn.cursor()

# READ
cursor.execute("SELECT * FROM books")
for row in cursor.fetchall():
    print(row)

# CREATE
cursor.execute(
    "INSERT INTO books (title, author, genre, price, quantity,
date_added) VALUES (%s, %s, %s, %s, %s, %s)",
    ("Homegoing", "Yaa Gyasi", "Fiction", 16.75, 20,
"2025-03-01")
)
cursor.execute(
    "INSERT INTO books (title, author, genre, price, quantity,
date_added) VALUES (%s, %s, %s, %s, %s, %s)",
    ("The Way of Zen", "Alan Watts", "Philosophy", 18.50, 25,
"2025-01-22")
)
cursor.execute(
    "INSERT INTO books (title, author, genre, price, quantity,
date_added) VALUES (%s, %s, %s, %s, %s, %s)",
    ("Why Nations Fail", "Daron Acemoglu & James A. Robinson",
"Politics", 19.95, 18, "2026-08-05")
)
conn.commit()

# UPDATE
```

```
cursor.execute(
    "UPDATE books SET quantity = %s WHERE title = %s",
    (35, "Homegoing")
)
conn.commit()

# DELETE
cursor.execute(
    "DELETE FROM books WHERE title = %s",
    ("Homegoing",)
)
conn.commit()

cursor.close()
conn.close()
```

Run the file, then check the MySQL terminal to confirm everything reflects automatically — the new books added, quantity changed, and "Homegoing" removed. Push your work to GitHub.

Key Takeaways

Every CRUD operation follows the same pattern from Python — build a SQL string with %s placeholders, pass real values in a separate tuple, call cursor.execute(), and call conn.commit() for anything that changes data. Parameterized queries protect your application from SQL injection, and reading, creating, updating, and deleting data from Python works against the exact same aice_books database you built directly in MySQL — always close your cursor and connection when you're done. You now know enough to build real applications with Python and MySQL together. There is one problem left: app.py still has a real password sitting in plain text, visible to anyone who opens the file or clones the repository. The next unit fixes that permanently, using environment variables to keep credentials out of your source code.