Learning Objectives

Learners will be able to...

- Execute select, insert, delete, update request to SQL database from Python
- Show SELECT response
- Describe the concept of SQL injection and implement preventive measures

info

Make Sure You Know

It is essential for the learner to have a basic understanding of SQL (Structured Query Language). They should be familiar with writing SQL queries using SELECT, INSERT, DELETE, and UPDATE statements to interact with a database.

Select Interface

We will use PyGreSQL to interact with the database. During intro, we ran a request for fetch cities, now we will fetch actors.

The library has 2 ways to execute request: own custom Db.query wrapper and Python Database API Specification v2.0 Interface.

Db.query wrapper

`.query()` request will return class of Query.

Useful methods here are the following.

Method	Description
Query.getresult()	Get query values as list of tuples
Query.dictresult()	Get query values as list of dictionaries
Query.one()	Get one row from the result of a query as a tuple
Query.onedict()	Get one row from the result of a query as a dictionary

```
q = db.query("SELECT actor_id, first_name, last_name FROM actor
LIMIT 10")


# print result as tuple
print(q.getresult())
# print result as dict
print(q.dictresult())
```

```
# results are sorted by last_name
q = db.query("SELECT actor_id, first_name, last_name FROM actor
ORDER BY last_name LIMIT 10")

# print first row as tuple
print(q.one())
# print next one row as dict
print(q.onedict())
```

Python Database API Specification v2.0

This API has been defined to encourage similarity between the Python modules that are used to access databases.

DB API provides a constructor connect, which returns a Connection object:

```
con = connect(dbname='codio', port=5432, user='codio')
```

Connection could support 4 methods: `.close()`, `.commit()`, `.rollback()` and `.cursor()`. We will use cursor method to interact with database and cursor method - `.execute()`.

```
q = cursor.execute("SELECT actor_id, first_name, last_name FROM
actor LIMIT 10")
```

The return value `.execute` or `.executemany` is undefined and should not be used. These methods are the way to get results after an operation:

Method	Description
.fetchone()	Returns the next row from a result set, and None when none remain.
.fetchmany([size=cursor.arraysize])	Returns a sequence of size rows (or fewer) from a result set. An empty sequence is returned when no rows remain. Defaults to arraysize.
.fetchall()	Returns all (remaining) rows from a result set. This behavior may be affected by arraysize.

```
q = cursor.execute("SELECT actor_id, first_name, last_name FROM
actor LIMIT 10")
print(q.fetchall())
```

Insert New Records

You can insert data using the execute method of the cursor and adding an INSERT SQL statement.

```
cursor.execute(
  "INSERT INTO actor(first_name, last_name) VALUES (%s, %s)",
  (first_name, last_name,)
)
```

The basis of this code is the same instruction that we used to insert Jackie Chan into the database.

```
INSERT INTO actor (first_name, last_name)
VALUES ('JACKIE', 'CHAN');
```

Only in this case, we request the user's first and last name and pass it to the request as parameters.

Check if you have successfully inserted the data.

---

important

Commit changes

Notice the `con.commit()` statement after the insert instruction. By default, changes are not applied to the database immediately, they are in a pending state until a command is issued to commit changes (by `.commit()`) or discard all changes(by `.rollback()`).

SQL Injections

Notice how the insert statement was formed in the previous example.

```
cursor.execute(
  "INSERT INTO actor(first_name, last_name) VALUES (%s, %s)",
  (first_name, last_name,)
)
```

It consists of two parts: the parameterized query (`INSERT INTO actor(first_name, last_name) VALUES (%s, %s)`) and its actual arguments (`(first_name, last_name,)`).

It would seem much easier to make a query like this.

```
cursor.execute(
  f"INSERT INTO actor(first_name, last_name) VALUES
('{first_name}', '{last_name}')"
)
```

But such code is potentially dangerous and can lead to SQL injection.

info

SQL injection

SQL injection is a security vulnerability that allows an attacker to interfere with the queries that an application makes to its database.

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.

For example, if there was vulnerable code in the file, then such an expression would result in the insertion of two new records.

```
 Here'); INSERT INTO actor(first_name, last_name) VALUES
('You','hacked
```

And this is the most harmless thing that can happen, because an attacker can either steal or simply delete data from the database if there is a vulnerability.

important

SQL injection prevention

Always use parameterized database queries when executing SQL code and validate data from the user.

Update existing record

Executing the update command will be much like the paste command, we will just use different SQL commands, but the general approach will remain the same.

```
cursor.execute(
  "UPDATE actor SET first_name = %s, last_name = %s WHERE
actor_id = %d",
  (first_name, last_name, actor_id,)
)
```

Note that a different string format character was used in the case of the `actor_id` - `%d`. This is because the data type for this field is an integer and we need to use the appropriate formatting.

At the end of the script, we checked the `rowcount` value to make sure the value we chose was updated. If the value of this field were 0, then no record was updated.

info

rowcount

This read-only attribute specifies the number of rows that the `last .execute*()` produced (for DQL statements like `SELECT`) or affected (for DML statements like `UPDATE` or `INSERT`)

Press button below to see how this works in a terminal.

Delete existing record

Deleting an entry looks almost the same, only we will use the DELETE statement.

```
cursor.execute(
  "DELETE FROM actor WHERE actor_id = %d",
  (actor_id,)
)
```

We pass the ID of the actor we want to remove as a parameter to the request.

important

When using the RUN DELETE ACTION button in the terminal type in “no” to see the correct action

info

Delete warning

Be careful when deleting records if an invalid identifier is passed - you may lose data.
That is why often, instead of a physical deletion, a record is marked as deletion, and the record itself remains in the database.

Complex select example

And of course we can execute and display a query of any complexity through the database driver.

Show top customers by payment example

```
q = cursor.execute("""
  SELECT MIN(amount), MAX(amount), SUM(amount) AS sum,
customer.first_name || ' ' || customer.last_name AS customer
  FROM payment
  JOIN customer
  ON payment.customer_id = customer.customer_id
  GROUP BY customer.customer_id
  ORDER BY sum DESC
  LIMIT %d;
""", (limit,))
```

Find the full name and email address of all customers that have rented an specified category movie.

```
q = cursor.execute("""
  SELECT first_name || ' ' || last_name AS name, email
  FROM customer WHERE customer_id IN (
    SELECT customer_id FROM rental WHERE inventory_id IN (
      SELECT inventory_id FROM inventory WHERE film_id IN (
        SELECT film_id FROM film_category JOIN category ON
film_category.category_id = category.category_id WHERE
category.name=%s
      )
    )
  )
  ORDER BY 1
  LIMIT %d;
""", (catogory, limit,))
```