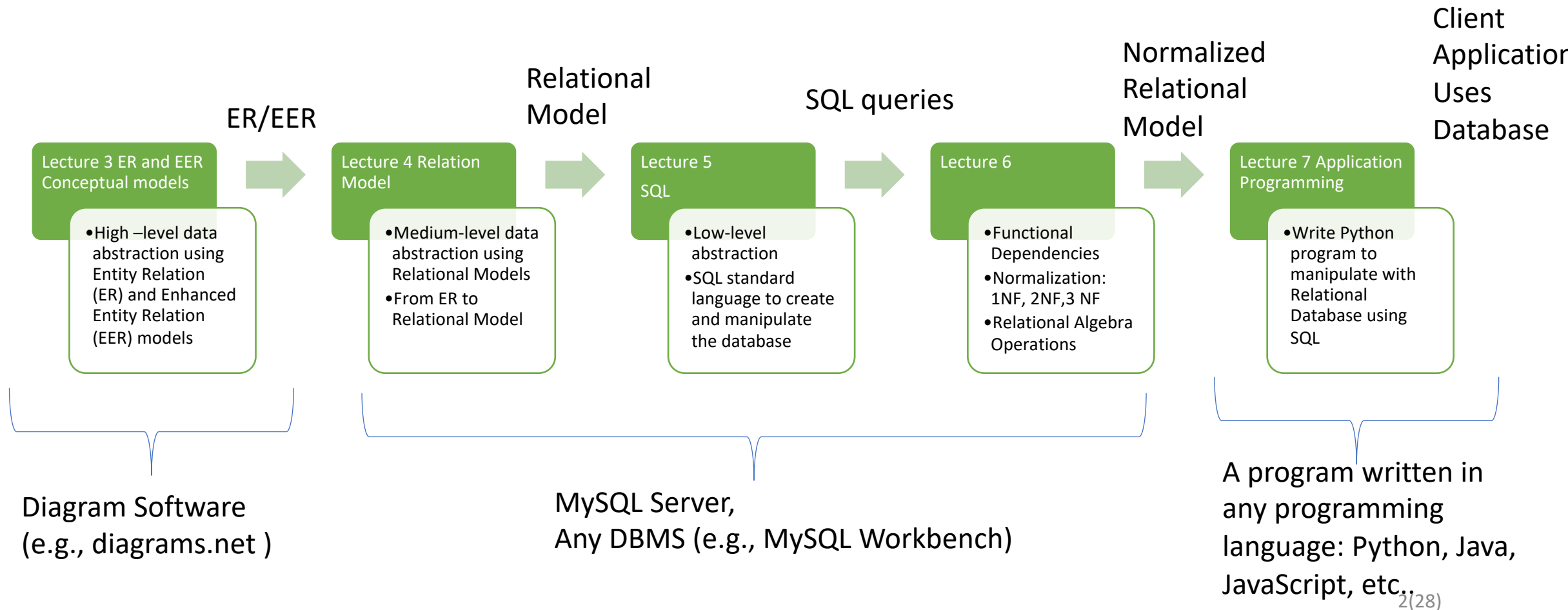


# Lecture 7. Database Application programming with SQL, Python

[alisa.lincke@lnu.se](mailto:alisa.lincke@lnu.se)

# Connection to previous lectures



# Outline

---

- Database Programming Techniques and Issues (Chapter 10)
- Python Database Application Programming (see References)

# Introduction to SQL Programming Techniques

- Most database access is accomplished through software programs that implement database applications.
- These programs are usually developed in different programming languages: Java, C#, Python, C/C++, JavaScript, PHP
- There are whole books devoted to each database programming technique. New techniques are developed all the time, and updates for existing techniques are also evolving.
- Although there are SQL standards which exist which are continually evolving and each DBMS may have some variations from the standard.

# Overview of main approaches for accessing a database from programs

- **Approach 1. Embedding database commands in a programming language.** The database statements are marked with special syntax which is recognized by the precompiler or preprocessor.
- **Approach 2. Using a library of database functions.** This is the most common approach to access the database through library written for specific programming language (Python, Java, PHP, etc.). It provides ready functions to connect to a database, prepare and execute queries. Nowadays this approach is known as *application programming interface (API)*.
- **Approach 3. Designing a new database programming language.**

# Approach 2 Issues

- **Data type mismatch:** *data types* of the programming language (e.g., Python, Java, etc.) differ from the *attribute types* available in SQL DBMS.
  - *Consequences:* program crashed, or SQL transaction (query) is failed.
  - *Solution:* a *binding* between attribute types (SQL) and compatible data type of *programming language* have to be performed. The values sets should be also check in the program side before performing the parsing data or data convertation, especially for dates data type.
- **Data structure:** the data structure which comes from SQL is set of tuples, like this [(‘att1’, ‘att2’, ‘att3’,...),(),(),()...]. In order to access the individual tuple’s elements, you need a binding to map query result data structure to an appropriate data structure in the programming language.
  - *Solution:* A general function is needed to loop over the tuples in a query result and re-structure the data in more convenient data structure (e.g., dictionaries in Python). Also you need to have another function which maps the programming language structure to set of tuples (SQL) for updating/inserting operations to database.

# General Programming Workflow

## Setting up database:

- Run MySQL Server
- Use any DBMS to connect to MySQL server
- Create a database using DBMS interactive interface
- Define database schema using DBMS interactive interface

## Writing Database Application/Program:

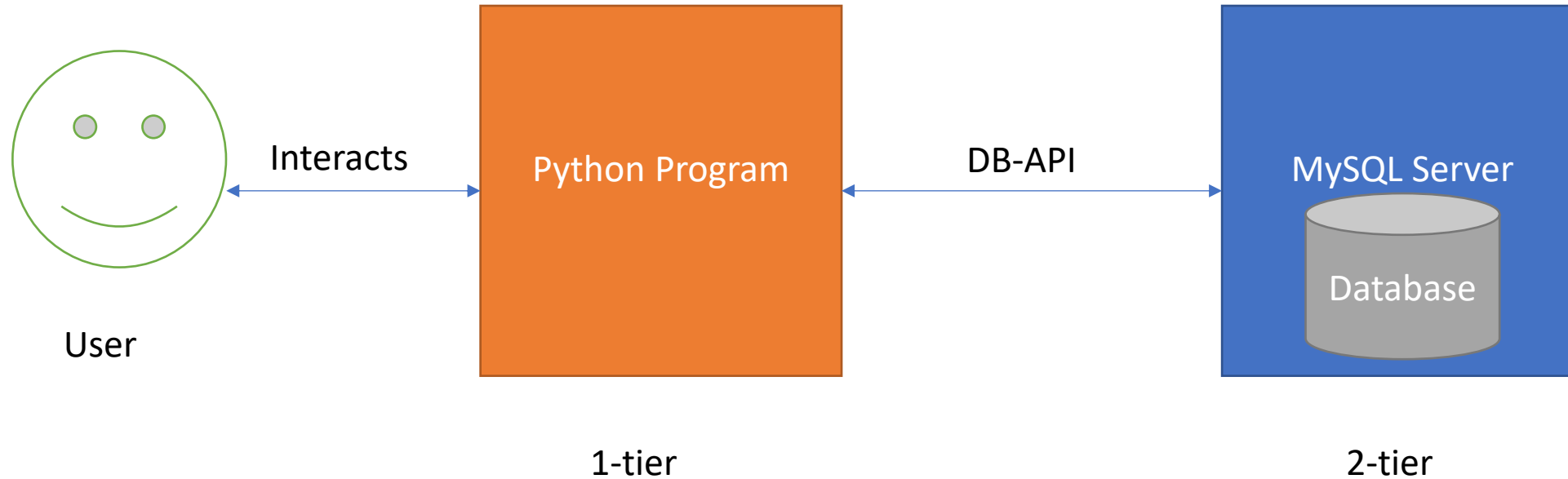
- Establish the connection to MySQL Server and database. In this step, we specify the URL address (e.g., localhost) of the machine where the database server is located, and providing a login account name and password for database access. **Note!** In real life application we do not use an admin (root) user to access the database. The database admin have to create a new user with read and write access for each database application/program. This user must not have access to modify the database schemes (such as delete tables, adding new tables)
- Once the database connection is established, the program can interact with the database by submitting queries commands.
- When the program no need anymore access to the database, it should *terminate* or *close* the connection to the database.

# Overview of Python Database Programming

---



# Python Database 2-tier Architecture



# DB-API for Python

- The Python DB API is a widely used module that provides a database application programming interface
- Python DB-API is independent of any database engine, which enables you to write Python scripts to access any database engine
- The Python DB API implementation for MySQL is [MySQLdb](#)
- MySQLdb version 1 is not supported anymore
- MySQLdb version 2 is outdated and migrated to so called mysqlclient library
- There are other DB-API exists: [Benchmarking MySQL drivers \(Python 3.4\)](#) among them is **MySQL Connector API** which we are going to use in this course

# MySQL Connector API

- MySQL provides MySQL adapter called MySQL connector API (written in Python and does not require any third-party library) which enables Python programs to access MySQL databases.
- This adapter allows the conversion between Python and MySQL data types
- It includes:
  - Connection
  - Cursor for querying the data
  - Handling exceptions

# Database Application workflow of Python program

- 1 Connect to the MySQL server
- 2 Connect to an existing database or create a new database
- 3 Write a Python interactive application with some user input
- 4 Execute a SQL query and fetch results
- 5 Print the results to end-user
- 6 Close the connection to the MySQL server

# Connection to Database Server (MySQL Server)

- The `getpass` module is used to hide the password
- The `connect` object from **`mysql.connector`** module is used to establish the database connection
- `try` and `catch` statement is used to catch and print exceptions/errors to the database
- The `with statement` is used to taking care of closing the database connection. Leaving unused open connections can lead to several unexpected errors and performance issues
- The `input` is used for providing the user and password in order to **avoid** the *hardcoded login credentials*. More secure ways to store sensitive information is using *environment variables*

Python

```
from getpass import getpass
from mysql.connector import connect, Error

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
    ) as connection:
        print(connection)
except Error as e:
    print(e)
```

# Create a new database (1)

Use a cursor to execute a SQL query in Python.

A [cursor](#) is a mechanism that enables traversal over the records in a database (similar to Iterator concept)

MySQL Connector/Python provides the [MySQLCursor](#) class, which instantiates objects that can execute MySQL queries in Python.

An instance of the MySQLCursor class is also called a cursor.

SQL	Python
CREATE DATABASE books_db;	db_query = "CREATE DATABASE books_db" with <a href="#">connection.cursor()</a> as cursor: cursor.execute(db_query )

**Note!** In MySQL, it's mandatory to put a semicolon (;) at the end of a statement, which denotes the termination of a query. However, MySQL Connector/Python automatically appends a semicolon at the end of your queries, so there's no need to use it in your Python code.

# Create a new database (2)

- You might receive an error here if a database with the same name already exists in your server.
- To check if the database is already exists use the following code:

```
db_exist = False
show_db_query = "SHOW DATABASES"
with connection.cursor() as cursor:
    cursor.execute(show_db_query)
    for database in cursor:
        if database == (db_Name,):
            db_exist = True
            cursor.execute("USE %s" % db_Name)
            break
```

```
print(database)
```

```
('information_schema',)
('mysql',)
('online_movie_rating',)
('performance_schema',)
('sys',)
```

# Connecting to an Existing Databases

- In most of the cases we already have created database using the DBMS and we only want to connect to it from our Python application
- We can do this the same connect() function by adding additional parameter called *database*, which is the name of database.

Python

```
from getpass import getpass
from mysql.connector import connect, Error

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
        database="online_movie_rating",
    ) as connection:
        print(connection)
except Error as e:
    print(e)
```



# Inserting Records in Tables (1)

- Using `cursor.execute()` method for inserting small number of records and the records can be hard-coded
- Use `connection.commit()` method to perform changes in the actual table.
- You can use many times the `cursor.execute()` method and in the end use **once** `connection.commit()` method to perform atomic transaction.
- **Note!** You don't need to add data for IDs as the AUTO\_INCREMENT build in function automatically creates id values in database.

Python

```
insert_movies_query = """
INSERT INTO movies (title, release_year, genre, collection_in_mil)
VALUES
    ("Forrest Gump", 1994, "Drama", 330.2),
    ("3 Idiots", 2009, "Drama", 2.4),
    ("Eternal Sunshine of the Spotless Mind", 2004, "Drama", 34.5),
    ("Good Will Hunting", 1997, "Drama", 138.1),
    ("Skyfall", 2012, "Action", 304.6),
    ("Gladiator", 2000, "Action", 188.7),
    ("Black", 2005, "Drama", 3.0),
    ("Titanic", 1997, "Romance", 659.2),
    ("The Shawshank Redemption", 1994, "Drama", 28.4),
    ("Udaan", 2010, "Drama", 1.5),
    ("Home Alone", 1990, "Comedy", 286.9),
    ("Casablanca", 1942, "Romance", 1.0),
    ("Avengers: Endgame", 2019, "Action", 858.8),
    ("Night of the Living Dead", 1968, "Horror", 2.5),
    ("The Godfather", 1972, "Crime", 135.6),
    ("Haider", 2014, "Action", 4.2),
    ("Inception", 2010, "Adventure", 293.7),
    ("Evil", 2003, "Horror", 1.3),
    ("Toy Story 4", 2019, "Animation", 434.9),
    ("Air Force One", 1997, "Drama", 138.1),
    ("The Dark Knight", 2008, "Action", 535.4),
    ("Bhaag Milkha Bhaag", 2013, "Sport", 4.1),
    ("The Lion King", 1994, "Animation", 423.6),
    ("Pulp Fiction", 1994, "Crime", 108.8),
    ("Kai Po Che", 2013, "Sport", 6.0),
    ("Beasts of No Nation", 2015, "War", 1.4),
    ("Andadhun", 2018, "Thriller", 2.9),
    ("The Silence of the Lambs", 1991, "Crime", 68.2),
    ("Deadpool", 2016, "Action", 363.6),
    ("Drishyam", 2015, "Mystery", 3.0)
"""

with connection.cursor() as cursor:
    cursor.execute(insert_movies_query)
    connection.commit()
```

# Inserting Records in Tables (2)

- Using `cursor.executemany()` method when the data are stored in file or generated by a different script.
- It takes two parameters:
  - A **query** string
  - A **list** with records to be inserted
- The code uses `%s` as a placeholder for the two strings that had to be inserted in the query string
- Placeholders act as **format specifiers** and help reserve a spot for a variable inside a string. The specified variable is then added to this spot during execution.

```
Python

insert_reviewers_query = """
INSERT INTO reviewers
(first_name, last_name)
VALUES ( %s, %s )
"""

reviewers_records = [
    ("Chaitanya", "Baweja"),
    ("Mary", "Cooper"),
    ("John", "Wayne"),
    ("Thomas", "Stoneman"),
    ("Penny", "Hofstadter"),
    ("Mitchell", "Marsh"),
    ("Wyatt", "Skaggs"),
    ("Andre", "Veiga"),
    ("Sheldon", "Cooper"),
    ("Kimbra", "Masters"),
    ("Kat", "Dennings"),
    ("Bruce", "Wayne"),
    ("Domingo", "Cortes"),
    ("Rajesh", "Koothrappali"),
    ("Ben", "Glocker"),
    ("Mahinder", "Dhoni"),
    ("Akbar", "Khan"),
    ("Howard", "Wolowitz"),
    ("Pinkie", "Petit"),
    ("Gurkaran", "Singh"),
    ("Amy", "Farah Fowler"),
    ("Marlon", "Crafford"),
]

with connection.cursor() as cursor:
    cursor.executemany(insert_reviewers_query, reviewers_records)
    connection.commit()
```

# Inserting Records in Tables (3)

**Note!** We use sting placeholder %s for any data type

**Note!** The Python program should check if the data type of the values are correct for the database

Python

```
insert_ratings_query = """
INSERT INTO ratings
(rating, movie_id, reviewer_id)
VALUES ( %s, %s, %s)
"""

ratings_records = [
    (6.4, 17, 5), (5.6, 19, 1), (6.3, 22, 14), (5.1, 21, 17),
    (5.0, 5, 5), (6.5, 21, 5), (8.5, 30, 13), (9.7, 6, 4),
    (8.5, 24, 12), (9.9, 14, 9), (8.7, 26, 14), (9.9, 6, 10),
    (5.1, 30, 6), (5.4, 18, 16), (6.2, 6, 20), (7.3, 21, 19),
    (8.1, 17, 18), (5.0, 7, 2), (9.8, 23, 3), (8.0, 22, 9),
    (8.5, 11, 13), (5.0, 5, 11), (5.7, 8, 2), (7.6, 25, 19),
    (5.2, 18, 15), (9.7, 13, 3), (5.8, 18, 8), (5.8, 30, 15),
    (8.4, 21, 18), (6.2, 23, 16), (7.0, 10, 18), (9.5, 30, 20),
    (8.9, 3, 19), (6.4, 12, 2), (7.8, 12, 22), (9.9, 15, 13),
    (7.5, 20, 17), (9.0, 25, 6), (8.5, 23, 2), (5.3, 30, 17),
    (6.4, 5, 10), (8.1, 5, 21), (5.7, 22, 1), (6.3, 28, 4),
    (9.8, 13, 1)
]

with connection.cursor() as cursor:
    cursor.executemany(insert_ratings_query, ratings_records)
    connection.commit()
```

# Reading Records From the Database (1)

- Create SELECT query string
- Use `cursor.fetchall()` method to extract the records
- It returns a list of tuples representing individual records from the table
- [Tuple](#) is ordered collection of objects, they are defined by enclosing the elements in parentheses `()` instead of square brackets. And they are immutable (read-only).
- Use the [LIMIT clause with \(offset,max\)](#) to constrain the number of rows that are received from the SELECT statement. And to provide the [pagination](#) when handling large volumes of data.

```
Python >>> select_movies_query = "SELECT * FROM movies LIMIT 5"
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     result = cursor.fetchall()
...     for row in result:
...         print(row)
...
(1, 'Forrest Gump', 1994, 'Drama', Decimal('330.2'))
(2, '3 Idiots', 2009, 'Drama', Decimal('2.4'))
(3, 'Eternal Sunshine of the Spotless Mind', 2004, 'Drama', Decimal('34.5'))
(4, 'Good Will Hunting', 1997, 'Drama', Decimal('138.1'))
(5, 'Skyfall', 2012, 'Action', Decimal('304.6'))
```

LIMIT 5 means only the first 5 records are fetched

limit = 5

“””.... LIMIT {0},{1};”””.format(offset,limit)

.....

If button\_next\_click:

offset += 1 //increase the offset

# Reading Records From the Database (2)

```
planetName = input("Provide a planet name: ")
cursor.execute("SELECT * FROM planets WHERE name = '%s'" % planetName)
planet_details = cursor.fetchall()
print(planet_details)
```

# Printing Tuples list as Table in Python

Functions to map data structure from database structure to Python data structure and vice versa

## Printing tuples as table

```
list_el = [('element1', 'element2', 'element3'),
            ('elementelel4', 'element5', 'elementelement'),
            ('el7', 'el8', 'elel9')]

row = "{name1:^20}|{name2:^20}|{name3:^20}".format
for tup in list_el:
    print(row(name1=tup[0], name2=tup[1], name3=tup[2]))
```

Output:

element1		element2		element3
elementelel4		element5		elementelement
el7		el8		elel9

```
def tuples_to_dict(list_tuples, column_names):
    list_dict = []
    for tuple in list_tuples:
        dict = {}
        dict[column_names[0]] = tuple[0]
        dict[column_names[1]] = tuple[1]
        dict[column_names[2]] = tuple[2]
        list_dict.append(dict)
    return list_dict
```

```
def dict_to_tuples(list_dict):
    list_tuples = []
    for item in list_dict:
        list_tuples.append(tuple(item.values()))
    return list_tuples
```

```
list_tuples = [('element1', 'element2', 'element3'), ('element5', 'element6', 'element7'),
column_names = ["first_name", "last_name", "birthday"]
```

```
my_list = tuples_to_dict(list_tuples, column_names)
print(my_list)
```

```
my_tuples = dict_to_tuples(my_list)
print(my_tuples)
```

# Filtering Results Using the WHERE Clause

```
Python >>>
>>> select_movies_query = """
... SELECT title, collection_in_mil
... FROM movies
... WHERE collection_in_mil > 300
... ORDER BY collection_in_mil DESC
... """
>>> with connection.cursor() as cursor:
...     cursor.execute(select_movies_query)
...     for movie in cursor.fetchall():
...         print(movie)
...
('Avengers: Endgame', Decimal('858.8'))
('Titanic', Decimal('659.2'))
('The Dark Knight', Decimal('535.4'))
('Toy Story 4', Decimal('434.9'))
('The Lion King', Decimal('423.6'))
('Deadpool', Decimal('363.6'))
('Forrest Gump', Decimal('330.2'))
('Skyfall', Decimal('304.6'))
```

```
height = int(input("Provide a height: "))
cursor.execute("SELECT name FROM species WHERE average_height>=%s", (height,))
planets_with_height = cursor.fetchall()
print(planets_with_height)
```

# Updating Records From the Database

Python

```
update_query = """
UPDATE
    reviewers
SET
    last_name = "Cooper"
WHERE
    first_name = "Amy"
"""
with connection.cursor() as cursor:
    cursor.execute(update_query)
    connection.commit()
```

**Note:** In the UPDATE query, the WHERE clause helps specify the records that need to be updated. If you don't use WHERE, then all records will be updated!

Python

```
from getpass import getpass
from mysql.connector import connect, Error

movie_id = input("Enter movie id: ")
reviewer_id = input("Enter reviewer id: ")
new_rating = input("Enter new rating: ")
update_query = """
UPDATE
    ratings
SET
    rating = "%s"
WHERE
    movie_id = "%s" AND reviewer_id = "%s";

SELECT *
FROM ratings
WHERE
    movie_id = "%s" AND reviewer_id = "%s"
""" % (
    new_rating,
    movie_id,
    reviewer_id,
    movie_id,
    reviewer_id,
)

try:
    with connect(
        host="localhost",
        user=input("Enter username: "),
        password=getpass("Enter password: "),
        database="online_movie_rating",
    ) as connection:
        with connection.cursor() as cursor:
            for result in cursor.execute(update_query, multi=True):
                if result.with_rows:
                    print(result.fetchall())
            connection.commit()
except Error as e:
    print(e)
```



# Deleting Records From the Database

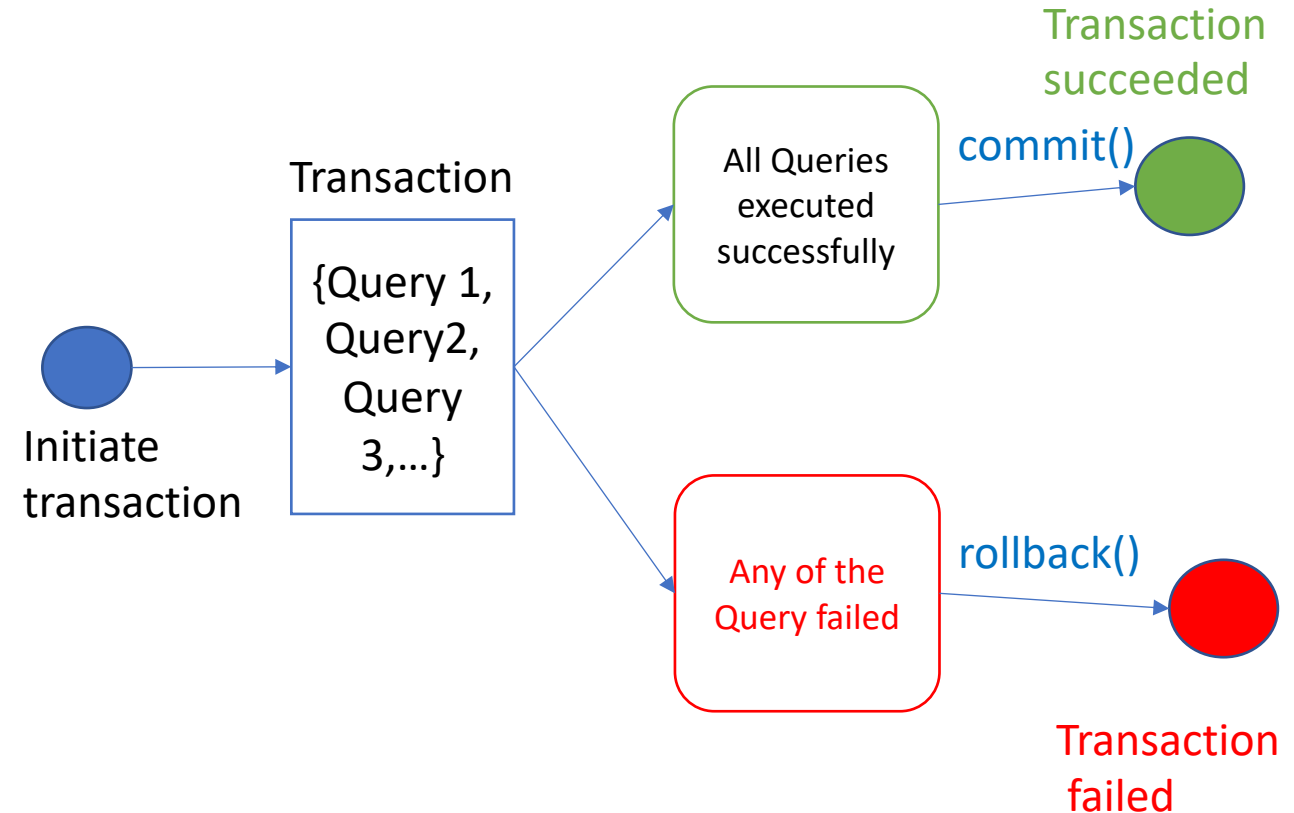
Python

```
delete_query = "DELETE FROM ratings WHERE reviewer_id = 2"
with connection.cursor() as cursor:
    cursor.execute(delete_query)
    connection.commit()
```

**Note:** Deleting is an *irreversible* process. If you don't use the **WHERE** clause, then all records from the specified table will be deleted. You'll need to run the **INSERT INTO** query again to get back the deleted records.

# Commit & Rollback Operation to manage transactions in Python

- The `commit()` method is used to make sure the changes made to the database are consistent
  - Syntax: `connection.commit()`
- The `rollback()` method is used to revert the last changes made to the database when one of the query failed.
  - Syntax: `connection.rollback()`



# Example

```
import mysql.connector

try:
    conn = mysql.connector.connect(host='localhost',
                                   database='python_db',
                                   user='pynative',
                                   password='pynative@#29')

    conn.autocommit = False
    cursor = conn.cursor()
    # withdraw from account A
    sql_update_query = """Update account_A set balance = 1000 where id = 1"""
    cursor.execute(sql_update_query)

    # Deposit to account B
    sql_update_query = """Update account_B set balance = 1500 where id = 2"""
    cursor.execute(sql_update_query)
    print("Record Updated successfully ")

    # Commit your changes
    conn.commit()

except mysql.connector.Error as error:
    print("Failed to update record to database rollback: {}".format(error))
    # reverting changes because of exception
    conn.rollback()
finally:
    # closing database connection.
    if conn.is_connected():
        cursor.close()
        conn.close()
        print("connection is closed")
```

Do not hard code the user credentials!  
You can do it only in development mode

# References

Python and MySQL Database: A Practical Introduction: [Web URL](#)

Use Commit and Rollback to Manage MySQL Transactions in Python:  
<https://pynative.com/python-mysql-transaction-management-using-commit-rollback/>