



# **MASTER OF COMPUTER APPLICATIONS**

## **SEMESTER 1**

### **PYTHON PROGRAMMING**

# Unit 10

## Database Interaction

### Table of Contents

SL.No	Topic	Fig No / Table / Graph	SAQ / Activity	Page No
1	<a href="#">Introduction</a>	-	-	4 - 5
1.1	<a href="#">Learning Objectives</a>	-	-	
2	<a href="#">SQL Basics</a>	-	-	6 - 9
2.1	<a href="#">Understanding Relational Databases</a>	-	-	
2.2	<a href="#">Primary Keys and Foreign Keys</a>	-	-	
2.3	<a href="#">SQL Syntax</a>	-	-	
2.4	<a href="#">Data Definition Language (DDL)</a>	-	-	
2.5	<a href="#">Data Manipulation Language (DML)</a>	-	-	
2.6	<a href="#">Transactions in SQL</a>	-	-	
2.7	<a href="#">Data Control Language (DCL)</a>	-	-	
2.8	<a href="#">SQL Joins</a>	-	-	
3	<a href="#">System Requirements</a>	-	-	10 - 14
3.1	<a href="#">Advantages of MySQL Connector Python</a>	-	-	
3.2	<a href="#">Environmental Setup</a>	-	-	
3.3	<a href="#">Install Mysql.Connector</a>	-	-	
4	<a href="#">Creating The Connection</a>	-	-	15 - 16
4.1	<a href="#">Creating Cursor Object</a>	-	-	
5	<a href="#">Fetching The Data</a>	-	-	17 - 20
5.1	<a href="#">Creating A Table</a>	-	-	
5.2	<a href="#">Alter Table</a>	-	-	
6	<a href="#">Insert Operation</a>	-	-	21 - 23
6.1	<a href="#">Read Operation</a>	-	-	
6.2	<a href="#">Update Operation</a>	-	-	
6.3	<a href="#">Join Operation</a>	-	-	

7	<a href="#">SQLite Module</a>		-	-	24 – 26
	7.1	<a href="#">Installing SQLite</a>	-	-	
	7.2	<a href="#">Basic Example</a>	-	-	
8	<a href="#">CRUD operations</a>		-	-	27 – 30
	8.1	<a href="#">Creating Data</a>	-	-	
	8.2	<a href="#">Reading Data</a>	-	-	
	8.3	<a href="#">Updating Data</a>	-	-	
	8.4	<a href="#">Deleting Data</a>	-	-	
	8.5	<a href="#">Real-Life Use Case: Inventory Management System</a>	-	-	
9	<a href="#">Transactions</a>		-	-	31 – 33
	9.1	<a href="#">Understanding Transactions</a>	-	-	
	9.2	<a href="#">Implementing Transactions in SQLite with Python</a>	-	-	
	9.3	<a href="#">Working with Savepoints</a>	-	-	
10	<a href="#">Summary</a>		-	-	32 – 35
11	<a href="#">Glossary</a>		-	-	36 – 37
12	<a href="#">Self-Assessment Questions</a>		-	<a href="#">1</a>	38 – 40
13	<a href="#">Terminal Questions</a>		-	-	41
14	<a href="#">Answers</a>		-	-	42 – 44
	14.1	<a href="#">Self-Assessment Questions</a>	-	-	
	14.2	<a href="#">Terminal Questions</a>	-	-	

## 1. INTRODUCTION

Unit 10 dives into the heart of database interaction, a pivotal skill in the realm of software development. After mastering regular expressions and text processing in Unit 9, where you learned to sift through and manipulate textual data efficiently, you're now stepping into the structured world of databases. Regular expressions provided you with a powerful tool for parsing and extracting information from unstructured data. In contrast, SQL and database interactions will empower you to handle, manipulate, and analyze structured data in relational databases.

SQL Basics form the cornerstone of this unit, providing you with the foundational knowledge to interact with relational databases. You'll explore the creation, manipulation, and querying of databases, learning how to use SQL to communicate with databases effectively. From understanding the intricacies of tables and relationships to mastering the art of writing efficient queries, this unit is designed to equip you with the skills needed to manage data in a structured environment. Whether it's retrieving specific information, updating records, or ensuring data integrity through transactions, SQL Basics covers the essential aspects of database management.

To excel in this unit, immerse yourself in the practical application of the concepts. Begin by experimenting with simple SQL commands, gradually moving to more complex queries and database operations. Engage with real-life scenarios or projects that require database interaction, applying the CRUD operations and transactional control you learn. Utilize online resources, practice exercises, and, if possible, real-world datasets to challenge your understanding and enhance your skills. Remember, the key to mastering SQL and database interaction lies in practice, persistence, and continuous exploration of new challenges.

### 1.1 Learning Objectives

*After studying this unit, you will be able to:*

- ❖ *Recall the basic syntax and commands of SQL.*
- ❖ *Explain the concepts of tables, rows, columns, and data types in relational databases.*
- ❖ *Execute SQL queries to insert, retrieve, update, and delete data in a database.*
- ❖ *Differentiate between various types of SQL commands and their purposes, such as DDL, DML, DCL, and TCL.*

- ❖ *Assess the effects of different SQL queries on the database and data integrity.*
- ❖ *Construct complex SQL queries involving multiple tables, using joins, subqueries, and aggregate functions to solve specific data retrieval challenges.*



## 2. SQL BASICS

SQL, or Structured Query Language, is a powerful tool for managing and manipulating relational databases. It's used for everything from querying data to updating and managing databases and their structures. This comprehensive overview will cover key concepts, commands, and practices in SQL, guiding you through the basics and beyond.

SQL is the standard language for relational database management systems. It's used to communicate with databases and perform various operations like creating, modifying, and querying databases. SQL commands can be categorized into several types, including Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), and Transaction Control Language (TCL).

### 2.1. Understanding Relational Databases

Before diving into SQL, it's crucial to understand the structure of relational databases. They are organized into tables, which consist of rows and columns. Each table represents a specific type of entity, and each row within a table represents a single instance of that entity. Columns, on the other hand, represent attributes of the entity.

### 2.2. Primary Keys and Foreign Keys

**Primary Key:** A column or a set of columns that uniquely identifies each row in a table. Every table should have a primary key to ensure that each record can be uniquely identified.

**Foreign Key:** A column or a set of columns in one table that refers to the primary key of another table. Foreign keys establish a relationship between two tables.

### 2.3. SQL Syntax

SQL commands are made up of statements that follow a specific syntax. Each statement consists of clauses, expressions, predicates, and queries. The syntax is designed to be somewhat intuitive, resembling natural language to a degree.

### 2.4. Data Definition Language (DDL)

DDL commands are used to define, modify, and delete database structures but not the data within them. Common DDL commands include:

- **CREATE:** Used to create a new table or database.

- **ALTER:** Used to modify an existing database object, like adding or deleting a column in a table.
- **DROP:** Used to delete an entire table or database.

### Creating Tables

Creating a table involves specifying its name and defining its columns and their data types.

For example:

```
CREATE TABLE Students (  
    StudentID int,  
    FirstName varchar(255),  
    LastName varchar(255),  
    Primary Key (StudentID)  
);
```

### Altering Tables

You might need to modify a table after it's created, such as adding a new column:

```
ALTER TABLE Students  
ADD Email varchar(255);
```

## 2.5. Data Manipulation Language (DML)

DML commands are used for managing data within tables. These include:

- **SELECT:** Retrieves data from one or more tables.
- **INSERT:** Adds new rows to a table.
- **UPDATE:** Modifies existing data in a table.
- **DELETE:** Removes rows from a table.

### Selecting Data

The SELECT statement is used to query the database and retrieve data that matches specified criteria.

```
SELECT FirstName, LastName FROM Students WHERE StudentID = 1;
```

### Inserting Data

The INSERT INTO statement is used to add new rows to a table.

```
INSERT INTO Students (StudentID, FirstName, LastName)
```

```
VALUES (1, 'John', 'Doe');
```

### Updating Data

The UPDATE statement is used to modify existing records in a table.

```
UPDATE Students  
SET Email = 'john.doe@example.com'  
WHERE StudentID = 1;
```

### Deleting Data

The DELETE statement is used to remove rows from a table.

```
DELETE FROM Students WHERE StudentID = 1;
```

## 2.6. Transactions in SQL

A transaction is a sequence of operations performed as a single logical unit of work. SQL transactions ensure data integrity and follow the ACID properties (Atomicity, Consistency, Isolation, Durability).

To manage transactions, SQL provides the following commands:

- **BEGIN TRANSACTION:** Starts a new transaction.
- **COMMIT:** Saves all changes made during the current transaction.
- **ROLLBACK:** Undoes all changes made during the current transaction.

### Implementing Transactions

```
BEGIN TRANSACTION;  
INSERT INTO Students (StudentID, FirstName, LastName) VALUES (2, 'Jane', 'Doe');  
COMMIT;
```

## 2.7. Data Control Language (DCL)

DCL commands are used to grant and revoke user permissions. The most common DCL commands are GRANT and REVOKE.

```
GRANT SELECT ON Students TO User1;  
REVOKE SELECT ON Students FROM User1;
```



## 2.8. SQL Joins

Joins are used to combine rows from two or more tables based on a related column. The most common types of joins are:

- **INNER JOIN:** Returns rows when there is at least one match in both tables.
- **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table, and the matched rows from the right table.
- **RIGHT JOIN (or RIGHT OUTER JOIN):** Returns all rows from the right table, and the matched rows from the left table.
- **FULL JOIN (or FULL OUTER JOIN):** Returns rows when there is a match in one of the tables.

Example of an INNER JOIN

```
SELECT Students.FirstName, Courses.CourseName  
FROM Students  
INNER JOIN Enrollments ON Students.StudentID = Enrollments.StudentID  
INNER JOIN Courses ON Enrollments.CourseID = Courses.CourseID;
```

### Advanced SQL Concepts

As you become more familiar with SQL, you'll encounter more advanced concepts, such as:

- **Indexes:** Structures that improve the speed of data retrieval operations on a table.
- **Views:** Virtual tables based on the result set of an SQL statement.
- **Stored Procedures:** SQL code that can be saved and reused.
- **Triggers:** Automated procedures that are triggered by specific actions on a table.

SQL is a versatile and powerful language that forms the backbone of database management and manipulation. Understanding its syntax, commands, and best practices is essential for anyone working with databases. As you continue to explore SQL, experiment with different queries and consider the impact of database design on query performance. Remember, the key to mastering SQL is practice and continuous learning.

We'll be looking at 2 libraries to use SQL with Python. One is using MySQL Connector, and the other is using SQLite3.

### 3. SYSTEM REQUIREMENTS

What is MySQL?

It is an RDBMS with a multitude of features. You can create multiple related tables in a database using MySQL. Being open-source software, it can be downloaded freely and customised as per your needs. It has built-in support for languages such as Python and PHP. It offers an efficient security system and consists of a thread-based memory allocation system. There is no risk of memory leakage with MySQL. It can also hold large databases with efficiency.

You can integrate MySQL database in Python using the module MySQL Connector Python. Using the MySQL database server, you can develop and integrate Python applications. There are various other modules that one can choose as per their requirements such as PyMySQL, MySQL DB, OurSQL, and more. Among the various available options, we will choose to continue with MySQL Connector. Note that the syntax, method, and ways of accessing the database are some of the factors that are the same for all the modules. This is because they are designed to adhere to the regulations of Python Database AP V2.0.

#### 3.1. Advantages of MySQL Connector Python

There are various benefits of choosing the module MySQL Connector Python over others for its better support and efficiency. These benefits are listed below.

- MySQL Connector Python is coded in Python and can execute queries through Python seamlessly.
- Supported by Oracle, it is the official link between MySQL and Python.
- MySQL Connector Python is frequently updated and maintained for the best experience and result.

#### 3.2. Environmental Setup

The system requirements for Python MySQL are similar to that of Python. By now, you must have set up Python on your system. To access the MySQL database, the first requirement for Python is to download a MySQL driver. Here, we are using MySQL Connector as the driver. To download it, you will need PIP.

PIP is pre-installed with the Python package and will be readily available for use. You will need root or administrative access to begin the installation process. MySQL Connector needs to be in the system's path. Unless it can find Python on your system, it will not install.

In Unix, Python is located in a directory included in the default path. However in Windows, Python may not exist in the system's path. It can be added to the directory containing python.exe by the user.

The steps that are provided below to install Python MySQL Connector are applicable for the following specifications:

*Platforms: 64-bit Windows, Windows 10, Windows 7, Windows 8, Windows Vista, Windows XP, Linux, Ubuntu Linux, Debian Linux, SUSE Linux, Red Hat Linux, Fedora, MacOS.*

*Python Version: Python 2 and Python 3.*

*MySQL Version: Version 4.1 or higher.*

### **3.3. Install Mysql.Connector**

MySQL Connector can be installed in various ways and methods. These are explained in detail below.

#### **Using Pip Command to Install MySQL Connector Python**

It is a straightforward process to download MySQL Connector Python with pip in Python. The MySQL Connector Python is available in pypi.org and can be downloaded using the pip command. On the command window, type the command given below.

*pip install mysql – connector – python*

You can also mention the version of the module in the command line if you are facing any issues while downloading the module. The command with version will be as provided below.

*pip install mysql – connector – python == 8.0.11*

Once the command is run, you will need to verify that MySQL Connector Python has been successfully installed. You can do this by checking the following messages that appear on the command window.

*Connecting mysql – connector – python.*

*Downloading packages.*

*Requirement already satisfied: setup tools in D: python\python 37-32\lib\site – packages*

*Installing collected packages: mysql – connector – python.*

*Successfully installed mysql – connector – python – 8.0.13.*

### Using Source Code Distribution (On Windows)

Firstly, go to the link <https://dev.mysql.com/downloads/connector/python/> to download the MySQL Connector Python for Windows for free. It will be available in the .zip extension. The link leads you to the latest version of the module.

You can also download older versions that are compatible with your device from the website at which you land when you click on the link provided below.

Click on the download button to save the ZIP file on your system.

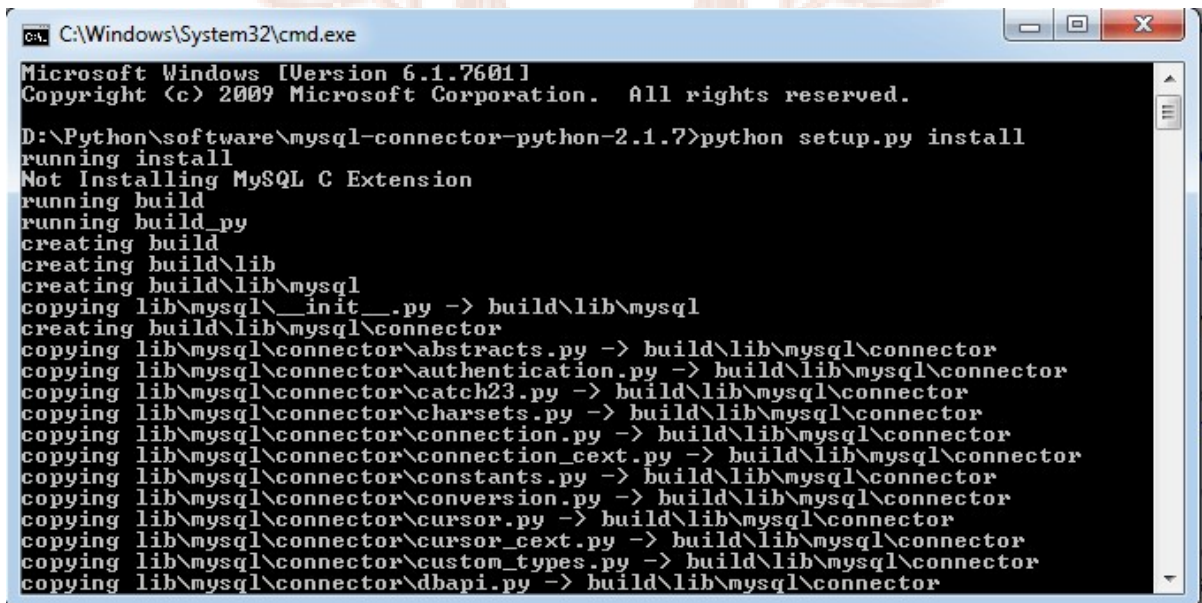
Unzip the zip archive in the directory of your choice. You can use any tool for this process.

Start a console window and change the location to the folder or directory where you unzipped the Zip archive.

Use the command provided below to begin installation inside the MySQL Connector Python folder.

*python setup.py install.*

Here, we have considered that the zip file is saved in the C drive of the system. Now, your screen should look like the one shown in the image below.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

D:\Python\software\mysql-connector-python-2.1.7>python setup.py install
running install
Not Installing MySQL C Extension
running build
running build_py
creating build
creating build\lib
creating build\lib\mysql
copying lib\mysql\__init__.py -> build\lib\mysql
creating build\lib\mysql\connector
copying lib\mysql\connector\abstracts.py -> build\lib\mysql\connector
copying lib\mysql\connector\authentication.py -> build\lib\mysql\connector
copying lib\mysql\connector\catch23.py -> build\lib\mysql\connector
copying lib\mysql\connector\charsets.py -> build\lib\mysql\connector
copying lib\mysql\connector\connection.py -> build\lib\mysql\connector
copying lib\mysql\connector\connection_cext.py -> build\lib\mysql\connector
copying lib\mysql\connector\constants.py -> build\lib\mysql\connector
copying lib\mysql\connector\conversion.py -> build\lib\mysql\connector
copying lib\mysql\connector\cursor.py -> build\lib\mysql\connector
copying lib\mysql\connector\cursor_cext.py -> build\lib\mysql\connector
copying lib\mysql\connector\custom_types.py -> build\lib\mysql\connector
copying lib\mysql\connector\dbapi.py -> build\lib\mysql\connector
```

Source: Pynative

**Fig 1:** Python Setup in Windows

## Download MySQL Connector Python on Linux

### Using Source Code Distribution

Go to the link <https://dev.mysql.com/downloads/connector/python/> and download the suitable version of MySQL Connector Python on your system.

You can choose the version of the TAR archive as per the specifications of your system.

Once the download is complete, you will need to follow the steps provided below to install the module.

To untar the downloaded file, use the command provided below.

```
shell>tar xzf mysql – connector – python – VERSION.tar.gz
```

Change the directory where you extracted the tar file.

```
shell> cd mysql – connector – python – VERSION
```

Now, execute the command provided below to install the module on Linux.

```
shell> sudo python setup.py install
```

## Download MySQL Connector Python on macOS

Firstly, download the source code for macOS from the link <https://dev.mysql.com/downloads/connector/python/>.

Now, download the mysql-connector-python-8.0.11-macos10.13.dmg file.

Install the module by opening the downloaded file and double-clicking on the .pkg file.

## Download MySQL Connector Python on Ubuntu

Use the code provided below to install MySQL Connector Python.

```
sudo apt-get install mysql – connector – python
```

```
pip install mysql – connector – python
```

If the above code does not work, then you can proceed by downloading the Source Code form <https://dev.mysql.com/downloads/connector/python/> link.

You will get two DEB packages. Download them both and unpack them. To install them, use the code provided below.

```
sudo dpkg -I /path_to_downloaded_deb_file
```

```
sudo apt-get install -f
```

When you run both these commands one after the other, the MySQL Connector Python will install on Ubuntu.

To test if the installation was successful, you can run the command provided below.

```
Import mysql.connector
```

If the code runs without any errors, then you have the module installed on your system.



## 4. CREATING THE CONNECTION

The first step after downloading the module is to establish the connection between the database. You can create the database using the create command in the command prompt. This command comes with the MySQL module and is built-in. Here is an example.

```
#mysqladmin create Employee
```

This command will create a database named employee.

You can also create a database using Python with the help of the code provided below.

```
import mysql.connector  
db1 = mysql.connector.connect (  
    host = "localhost",  
    user = "myusername",  
    password = "mypassword"  
)  
mycursor = db1.cursor ()  
mycursor.execute ("CREATE DATABASE mydatabase")
```

In this code, you will find a cursor method. This will be explained in detail later in the unit.

To create a connection with an existing database, here is the code that you will need to type on your Python window.

```
import mysql-connector  
mydatabase = mysql.connector.connect (  
    host = "localhost",  
    user = "myname",  
    password = "mypassword"  
)  
print (mydatabase)
```

Here, the name of the database is mydatabase. The username is myusername and the password is mypassword. We are using the constructor connect () that accepts username, password, host, and the name of the password. It will return an object of the MySQL Connection class.



You can disconnect the database using the exit command in the mysql prompt. This is applied as follows.

```
mysql> exit
```

Or, you can close the connection through the Python window by adding the code provided below.

```
mydatabase.close ()
```

## 4.1. Creating Cursor Object

You can create a cursor object using the cursor () method that is built-in in the MySQL Connector Python module. The cursor objects are used to process and analyse individual rows returned by database system queries. Through the cursor, you can manipulate the whole result set at once. The code to create a cursor object using Python is provided below.

```
import mysql.connector
conn = mysql.connector.connect (
    host = "localhost",
    user = "myusername"
    password = "mypassword"
    database = "db1"
)
cursor = conn.cursor ()
```

The cursor class has the following properties.

- **column\_names:** It returns the list containing the column names of a result. It is a read-only value.
- **description:** It returns a list of description of columns in a result rest. It is read-only as well.
- **lastrowid:** It returns the value generated in an auto-incremented column of the table in the last INSERT or UPDATE operation.
- **rowcount:** It returns the number of rows that undergone the SELECT or UPDATE operations.
- **statement:** It returns the last executed statement.



## 5. FETCHING THE DATA

In this section, we will learn the basics of manipulating with database in MySQL using Python. If a database already exists, using few code lines, you can fetch the data present in the database. This can be done by either retrieving rows or columns of the table. You can also specify the data that you want to retrieve using queries such as WHERE.

Go through the code provided below. This code fetches the data present in a table using the query SELECT. Note that all the queries are written in all caps. The code uses the method fetchall () to retrieve all the rows from a MySQL table.

```
import mysql.connector
conn = mysql.connector.connect (
    host = "localhost",
    user = "myusername",
    password = "mypassword",
    database = "db1"
)
my_database = conn.cursor ()
sql_table = "SELECT * FROM EMPLOYEE"
my_database.execute (sql_table)
output = my_database.fetchall ()
for x in output:
    print (x)
```

Here, the execute () method is used to execute the MySQL query. The variables retrieved by the SELECT query have been saved in the variable sql\_table. The result set is saved as a list of tuples with the help of fetchall ().

### 5.1. Creating A Table

To create a table in MySQL, you use the CREATE query. Through Python, we will use this query and create a new table in our database. The code to form a new table in the database is provided below. Ensure that the name of the database has been defined while forming the connection with it.

```
import mysql.connector
db1 = mysql.connector.connect (
    host = "localhost",
    user = "myusername",
    password = "mypassword",
    database = "my_database"
)
mycursor = db1.cursor()
mycursor.execute (" CREATE TABLE employee (name VARCHAR (200), address VARCHAR (200))")
```

The code creates a table called employee in the database named db1. The table contains two columns, one will store the names of the employees and the other will store the address. Note that the data type for these two columns is set to be VARCHAR, which is a character string with a length of 200 bits.

You should also create a primary key while creating a table. The primary key is a column with a unique key for each record. The example to create a primary key has been provided with the code below.

```
import mysql.connector
db1 = mysql.connector.connect (
    host = "localhost",
    user = "myusername",
    password = "mypassword",
    database = "my_database"
)
mycursor = db1.cursor()
mycursor.execute (" CREATE TABLE employee (id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR (200), address VARCHAR (200))")
```

Here is a list of data types available in MySQL that can help you create various tables that can store different type of data.

**Table 1:** Data Types Available in Mysql

Data Type	Description
<b>NUMERIC</b>	It represents a number.
<b>DECIMAL (precision, scale)</b>	It has two parts, precision and scale. Precision specifies the number of decimal digits and scale specifies the number of digits that are to be stored following the decimal point.
<b>INT</b>	It represents an integer ranging from -2147483648 to 2147483647.
<b>SMALLINT</b>	They also represent integers from -32768 to 32767.
<b>FLOAT</b>	It represents a number from -1.79E+308 to 1.79E+38.
<b>REAL</b>	It represents a number ranging from -3.40E+38 to 3.40E+38.
<b>DOUBLE PRECISION</b>	It represents a 64-bit value from $10^{308}$ to $10^{-323}$
<b>DATETIME</b>	It represents date in the format of yyyy-mm-dd and time in hh:mm:ss.
<b>DATE</b>	It shows date in the format of yyyy-mm-dd.
<b>TIMESTAMP</b>	It represents dates ranging from 1970 to 2037 with a resolution of one second.
<b>TIME</b>	Shows time in the format of hh:mm:ss.
<b>YEAR</b>	It represents the year in form of yyyy from 1901 to 2155.
<b>CHAR (no-of-bytes)</b>	It represents strings of the length from 0 to 255.
<b>VARCHAR (no-of-bytes)</b>	It can be used to store only that many characters as needed. Thus, it is a variable string of characters.
<b>TEXT</b>	It stores strings without the need of specification of size.

## 5.2. Alter Table

ALTER query is used to change an existing table. When working in MySQL, you can simply apply the query and add a column to the table. An example is shown below.

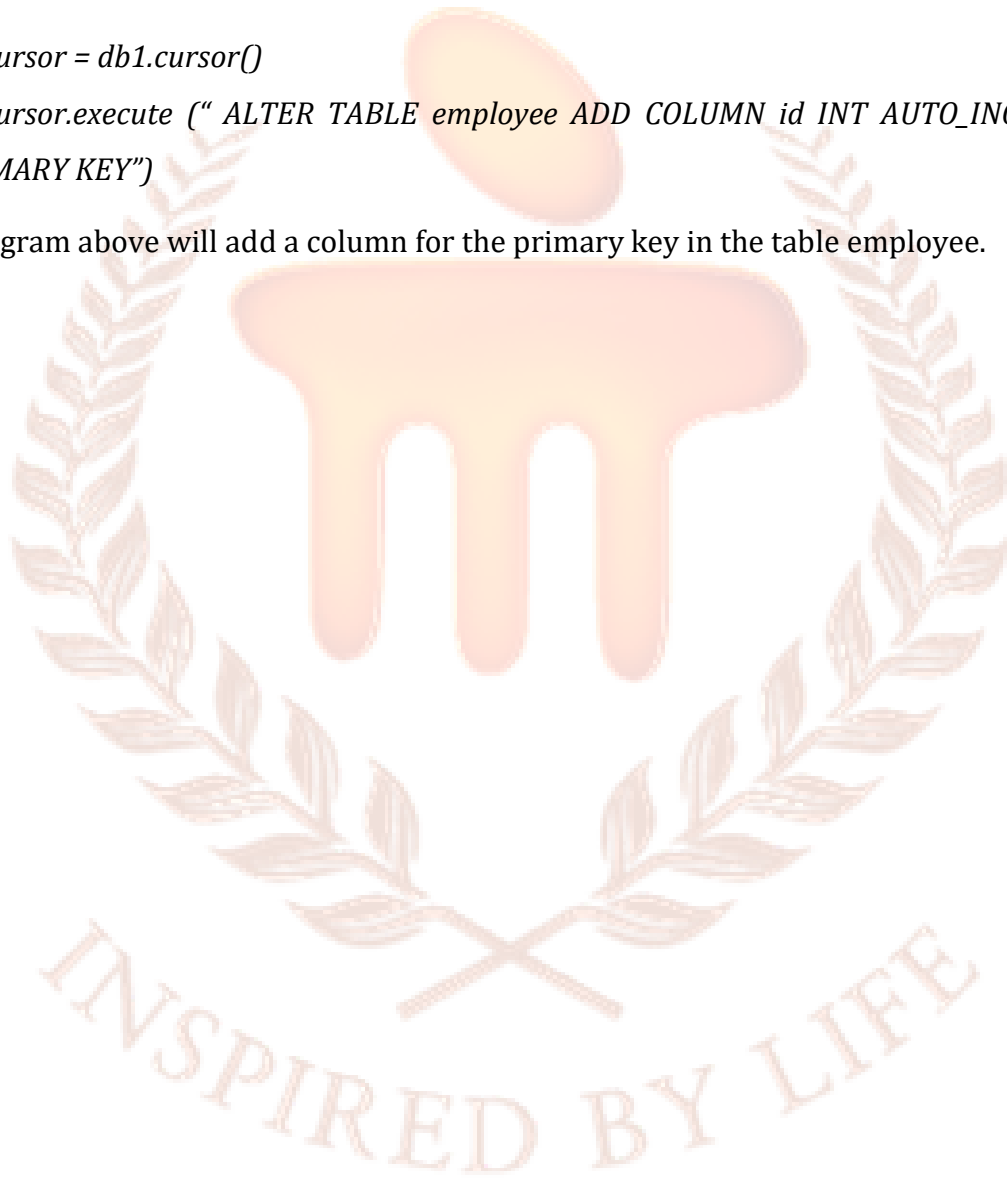
ALTER TABLE employee MODIFY COLUMN address VARCHAR (255).

Thus, the length of the string that could be stored in the address column is changed from 200 to 255. In Python, this code will be as shown below.

```
import mysql.connector
db1 = mysql.connector.connect (
```

```
host = "localhost",  
user = "myusername",  
password = "mypassword",  
database = "my_database"  
)  
mycursor = db1.cursor()  
mycursor.execute (" ALTER TABLE employee ADD COLUMN id INT AUTO_INCREMENT  
PRIMARY KEY")
```

The program above will add a column for the primary key in the table employee.



## 6. INSERT OPERATION

With INSERT Operation, you can add new rows to an existing table. You will need to specify the name of the table, column names, and values that you need to store in the added row.

The execute () method is used to accept a query as a parameter. The method executes the given query in Python. You must have noticed the function in various examples given previously. The code provided below gives an example regarding how the INSERT query works.

```
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print (mycursor.rowcount, "record inserted.")
```

You can insert multiple rows into a table using the method executemany(). You will need to provide parameters such as a list of tuples with the method. The list should contain the data that you want to insert.

### 6.1. Read Operation

To read values saved in the database, the SELECT query is used. There are various methods in Python that return the data stored in the table. These methods include fetchall () and fetchone (). These methods are already discussed above. You can use the SELECT query to select a complete table, some columns from the table, or only a row. Here is an example using the fetchone() method to return only one row.

```
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchone()
print(myresult)
```

To select data from a table, one can apply the filter using the WHERE filter. Here is what the command will be like when using the query.

```
sql = "SELECT * FROM customers WHERE address = 'Park Lane 38'"
```

## 6.2. Update Operation

Through UPDATE operation, the databases can be updated. You can update specific rows using the WHERE clause.

```
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'"
mycursor.execute(sql)
mydb.commit()
print(mycursor.rowcount, "record(s) affected")
```

### 6.3. Join Operation

To retrieve data that has been divided into two tables, the JOIN operation is used. The working of the operation has been depicted through the program provided below.

```
import mysql.connector
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT \
users.name AS user, \
products.name AS favorite \
FROM users \
INNER JOIN products ON users.fav = products.id"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

INNER JOIN only retrieves values that match in both the tables. If you need to retrieve all the values, then you can use LEFT JOIN and RIGHT JOIN statement.

## 7. SQLITE MODULE

SQLite is a C library that provides a lightweight disk-based database. It doesn't require a separate server process, making it a popular choice for embedding in applications. Python's standard library includes a module, `sqlite3`, which provides an interface to interact with SQLite databases. This integration allows Python applications to use SQLite databases for data storage and retrieval, combining Python's ease of use with SQLite's efficiency and simplicity.

### Why SQLite with Python?

SQLite with Python offers several advantages:

- **Simplicity:** SQLite databases are stored in a single file, simplifying database management and deployment. This is particularly useful for small to medium-sized applications, testing environments, and educational purposes where a full-scale database system might be overkill.
- **Portability:** SQLite databases are portable across different platforms without any modifications, making it ideal for applications that need to be distributed across various operating systems.
- **Zero Configuration:** SQLite does not require a separate server process or setup, making it incredibly easy to start with. There's no need to configure a database server, manage connections, or worry about user permissions at a basic level.
- **Integration:** Python's `sqlite3` module is part of the standard library, ensuring that any Python environment can interact with SQLite databases out of the box, without needing additional installations or configurations.
- **Concurrency and Atomicity:** SQLite supports transactions that are atomic, consistent, isolated, and durable (ACID), even after system crashes and power failures, ensuring data integrity.

### 7.1. Installing SQLite

Python's `sqlite3` module is included in the standard library, so you don't need to install anything extra to use SQLite with Python. However, to use SQLite's command-line tools or to ensure you have the latest version, you might consider installing SQLite separately.



For most operating systems, SQLite comes pre-installed. You can check your SQLite version by running `sqlite3 --version` in your terminal. If you need to install or upgrade SQLite, you can download pre-compiled binaries from the SQLite Downloads page.

## 7.2. Basic Example

Let's walk through a basic example to illustrate using SQLite with Python. This example will create a database, define a table, insert data, and query that data.

### 1. Importing the Module and Connecting to a Database

```
import sqlite3  
# Connect to a database (or create one if it doesn't exist)  
conn = sqlite3.connect('example.db')
```

### 2. Creating a Table

```
c = conn.cursor()  
# Create a table  
c.execute("""CREATE TABLE IF NOT EXISTS stocks  
(date text, trans text, symbol text, qty real, price real)""")
```

### 3. Inserting Data

```
# Insert a row of data  
c.execute("INSERT INTO stocks VALUES ('2023-03-17','BUY','RHAT',100,35.14)")  
# Save (commit) the changes  
conn.commit()
```

### 4. Querying Data

```
# Execute a query  
c.execute("SELECT * FROM stocks WHERE trans='BUY'")  
# Fetch and display the results  
for row in c.fetchall():  
    print(row)
```

### 5. Closing the Connection

```
# Close the connection when done  
conn.close()
```

This example covers the basics of connecting to an SQLite database, creating a table, inserting data, and running a query. These operations form the foundation of database interaction in any application.



## 8. CRUD OPERATIONS

In database systems, CRUD stands for Create, Read, Update, and Delete. These operations form the foundation of data manipulation and retrieval in databases. Let's dive into how these operations are implemented in SQLite using Python.

### 8.1. Creating Data

The "Create" in CRUD stands for the creation of new records in the database. In the context of SQLite and Python, this usually means inserting new rows into an existing table. You've already seen a basic example of this when we inserted a single row into the stocks table. Let's expand on this by adding multiple records and handling exceptions gracefully.

```
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Insert multiple records using the more secure "?" placeholder
stocks_data = [
    ('2023-03-17', 'BUY', 'GOOG', 150, 820.23),
    ('2023-03-18', 'SELL', 'AAPL', 100, 125.67),
    ('2023-03-19', 'BUY', 'MSFT', 200, 112.89)
]

c.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', stocks_data)

conn.commit()
conn.close()
```

This example uses the `executemany()` method, which is more efficient for inserting multiple records. It also uses placeholders (?) for parameters to prevent SQL injection, a common security vulnerability.

### 8.2. Reading Data

Reading, or querying, data from a database involves executing a `SELECT` statement and then fetching the results. SQLite and Python make this straightforward with the cursor's `execute()` and `fetchall()` methods.

```
conn = sqlite3.connect('example.db')
```

```
c = conn.cursor()

# Select all rows from the stocks table
c.execute('SELECT * FROM stocks')

# Fetch all results
rows = c.fetchall()

for row in rows:
    print(row)

conn.close()
```

This code selects all rows from the stocks table and prints each one. For large datasets, consider using `fetchone()` or `fetchmany(size)` to control memory usage.

### 8.3. Updating Data

Updating data involves modifying existing records in the database based on a specified condition. This is typically done with an UPDATE statement.

```
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Update the price of all 'AAPL' transactions
c.execute("UPDATE stocks SET price = 130.00 WHERE symbol = 'AAPL'")

conn.commit()
conn.close()
```

This code updates the price of all records where the symbol is 'AAPL' to 130.00.

### 8.4. Deleting Data

Finally, the "Delete" operation removes existing records from the database. This is done using a DELETE statement.

```
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Delete all 'SELL' transactions
```

```
c.execute("DELETE FROM stocks WHERE trans = 'SELL'")
```

```
conn.commit()
```

```
conn.close()
```

This code deletes all rows in the stocks table where the transaction type is 'SELL'.

## 8.5. Real-Life Use Case: Inventory Management System

Consider an inventory management system for a small retail business. The system needs to track products, their quantities, prices, and suppliers. CRUD operations form the backbone of this system, enabling the business to:

- **Create:** Add new products as they start selling them or add new suppliers as they partner with more vendors.
- **Read:** Generate reports on current inventory levels, products that need reordering, or sales reports showing the most popular products.
- **Update:** Adjust inventory levels as products are sold or received, update prices based on supplier changes, or update supplier information.
- **Delete:** Remove products that are no longer sold or remove suppliers they no longer do business with.

**Example Scenario:** A new shipment arrives, and the system needs to update the inventory.

**Read:** The system first checks the current inventory levels of the products in the shipment.

**Update:** For each product in the shipment, the system updates the inventory level, adding the quantity received to the existing quantity.

```
# Assuming 'shipment_details' is a dictionary with product names as keys and quantities as values
```

```
for product, quantity in shipment_details.items():
```

```
    c.execute("UPDATE products SET quantity = quantity + ? WHERE name = ?", (quantity, product))
```

```
conn.commit()
```

**Read:** After updating, the system generates a report showing the updated inventory levels for confirmation.

This simple use case illustrates how CRUD operations support the day-to-day functionality of a business through its inventory management system, enabling efficient and accurate tracking of products.



## 9. TRANSACTIONS

Transactions are fundamental to maintaining data integrity in database systems, including SQLite. A transaction is a sequence of one or more SQL operations that are executed as a single unit. Transactions ensure that a database transitions from one consistent state to another, maintaining data integrity even in cases of system failure or concurrent accesses.

### 9.1. Understanding Transactions

A transaction in a database system adheres to the ACID properties:

- **Atomicity:** Ensures that all operations within a transaction block are completed successfully; if not, the transaction is aborted, and the database state is left unchanged.
- **Consistency:** Guarantees that a transaction transforms the database from one valid state to another, maintaining all predefined rules, such as constraints, cascades, and triggers.
- **Isolation:** Ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- **Durability:** Ensures that once a transaction has been committed, it will remain so, even in the event of a system crash.

### 9.2. Implementing Transactions in SQLite with Python

In SQLite with Python, transactions are managed by the connection object. When you perform operations like insert, update, or delete, the changes are not immediately saved to the database. Instead, you need to commit these changes explicitly to make them permanent. This commit operation marks the end of a transaction.

**Example:** Handling a transaction that updates inventory levels and logs the update in an `inventory_log` table.

```
import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect('inventory.db')

c = conn.cursor()
```

```
try:
    # Start a transaction
    c.execute("BEGIN TRANSACTION;")

    # Update inventory levels
    c.execute("UPDATE products SET quantity = quantity - 10 WHERE name = 'Widget'")

    # Log the inventory update
    c.execute("INSERT INTO inventory_log (product_name, change, timestamp) VALUES (?, ?,
CURRENT_TIMESTAMP)", ('Widget', -10))

    # Commit the transaction
    conn.commit()
    print("Transaction completed successfully.")

except sqlite3.Error as e:
    # Rollback the transaction on error
    conn.rollback()
    print("Transaction failed. Changes rolled back.")
    print("Error:", e)

finally:
    # Close the database connection
    conn.close()
```

In this example, both the update to the products table and the insert into the inventory\_log table are part of a single transaction. If either operation fails, the except block is executed, rolling back the entire transaction to ensure the database's consistency is maintained. The finally block then ensures that the database connection is closed, whether the transaction succeeds or fails.

### 9.3. Working with Savepoints

SQLite also supports savepoints, which allow for more granular control over transactions by enabling nested transactions. A savepoint marks a point within a transaction that you can roll back to without rolling back the entire transaction.



**Example:** Using a savepoint to manage partial rollback within a transaction.

*try:*

*# Start a transaction*

*c.execute("BEGIN TRANSACTION;")*

*# Perform some updates*

*c.execute("UPDATE products SET quantity = 50 WHERE name = 'Gadget'")*

*# Create a savepoint*

*c.execute("SAVEPOINT sp1;")*

*# Perform more updates*

*c.execute("UPDATE products SET quantity = 30 WHERE name = 'Widget'")*

*# Something goes wrong, rollback to savepoint*

*c.execute("ROLLBACK TRANSACTION TO SAVEPOINT sp1;")*

*# Commit the transaction*

*conn.commit()*

*print("Transaction completed with partial rollback.")*

*except sqlite3.Error as e:*

*# Rollback the transaction on error*

*conn.rollback()*

*print("Transaction failed. Changes rolled back.")*

*print("Error:", e)*

*finally:*

*# Close the database connection*

*conn.close()*

In this case, the update to the 'Gadget' product is preserved, while the update to the 'Widget' product is rolled back to the state at the savepoint. This allows for more complex transaction management where certain operations can be tentatively performed and rolled back without affecting the entire transaction.

## 10. SUMMARY

Wrapping up our journey through SQL basics and the SQLite module in Python, it's been quite a ride, hasn't it? From understanding the foundational structure of relational databases to manipulating and managing data through SQL commands, we've covered substantial ground. Think of SQL as the language that lets you converse with databases, asking for the information you need or instructing changes in a structured, precise manner.

We started with the building blocks of databases – tables, rows, and columns, akin to the chapters, pages, and words in a book, each playing a crucial role in storing and organizing data. The concepts of primary and foreign keys were our guideposts, ensuring that each piece of data could be uniquely identified and related data could be interconnected, much like a well-organized index in a book.

Diving into the syntax and commands, we explored the realms of DDL, DML, DCL, and TCL. Creating and altering tables with DDL felt like sketching the blueprint of a data mansion, carefully designing each room (table) to hold specific pieces of information. With DML, we breathed life into these structures, inserting data, updating it, and occasionally clearing out rooms when they were no longer needed.

Then, there was the SQLite module, a light yet powerful tool that seamlessly integrated with Python, making it a go-to for applications requiring a compact, reliable database solution. The simplicity and zero-configuration setup of SQLite, combined with Python's versatility, opened up a world of possibilities for developing applications with embedded databases.

CRUD operations became our daily chores – creating new data entries, reading and retrieving information, updating records, and deleting what was no longer required. Each operation, a stroke of command, ensuring our database was a living, breathing entity, always in sync with our needs and changes.

Transactions introduced us to the art of maintaining data integrity, ensuring that every sequence of operations was completed with precision, leaving no room for errors or inconsistencies. The ACID properties of transactions were like the four pillars holding the sanctity of our database operations, ensuring reliability and trust in the data we manage.

As we close this chapter, remember that mastering SQL and understanding its integration with Python through modules like SQLite is a journey of continuous learning and practice. The real-world applications are vast and varied, from managing inventory systems for businesses to powering the back-end of web applications. So, keep experimenting, exploring, and enhancing your skills, and you'll find that SQL, along with Python, can open new doors to data management and analysis that you might not have imagined before.



## 11. GLOSSARY

**SQL (Structured Query Language):** A standardized programming language used for managing and manipulating relational databases.

**Relational Database:** A collection of data items organized as a set of formally described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables.

**Table:** A collection of related data held in a table format within a database. It consists of columns and rows.

**Row/Record:** A single, implicitly structured data item in a table.

**Column/Field:** A set of data values of a particular type, one for each row of the table.

**Primary Key:** A unique identifier for each record in a database table.

**Foreign Key:** A set of one or more columns in a database table that refers to the primary key in another table.

**DDL (Data Definition Language):** Part of SQL that deals with database schemas and descriptions, of how the data should reside in the database.

**DML (Data Manipulation Language):** The subset of SQL used for adding (inserting), deleting, and updating data in a database.

**DCL (Data Control Language):** A subset of SQL commands that include commands such as GRANT and REVOKE, which control access to data.

**TCL (Transaction Control Language):** Commands that deal with transaction operations within databases.

**SQLite:** A C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.

**sqlite3 Module:** A module in Python's Standard Library that provides an interface for interacting with SQLite databases.

**CRUD Operations:** Acronym for Create, Read, Update, and Delete operations, which represent the basic operations to manage data in a database.

**Transaction:** A sequence of database operations that are executed as a unit. Either all of the operations are executed, or none of them are.

**ACID Properties:** Set of properties (Atomicity, Consistency, Isolation, Durability) that guarantee database transactions are processed reliably.

**Commit:** The SQL command that ends a transaction by making all changes made during the transaction permanent.

**Rollback:** The SQL command that undoes all changes made during the current transaction.

**Join:** A SQL operation used to combine rows from two or more tables, based on a related column between them.

**Index:** A database object that speeds up the retrieval of rows from a table or view.

**View:** A virtual table based on the result set of an SQL statement. It contains rows and columns, just like a real table.

**Stored Procedure:** A set of SQL statements with an assigned name that's stored in the database in compiled form so that it can be shared by a number of programs.

**Trigger:** A database object that is automatically executed or fired when certain events occur.

## 12. SELF-ASSESSMENT QUESTIONS

1. SQL stands for \_\_\_\_\_ Query Language.
2. A \_\_\_\_\_ key uniquely identifies each record in a database table.
3. \_\_\_\_\_ is a subset of SQL used for adding, deleting, and updating data in a database.
4. A \_\_\_\_\_ in SQL is used to combine rows from two or more tables.
5. The \_\_\_\_\_ command is used to start a transaction in SQL.
6. SQLite databases are stored in a single \_\_\_\_\_ file.
7. In Python, the \_\_\_\_\_ module provides an interface to interact with SQLite databases.
8. The ACID property that ensures a transaction is either completely done or not done at all is \_\_\_\_\_.
9. A \_\_\_\_\_ view is a virtual table based on the result-set of an SQL statement.
10. \_\_\_\_\_ in a database ensures that once a transaction has been committed, it will remain so, even in the event of a system crash.
11. Which SQL command is used to remove rows from a table?
  - A. REMOVE
  - B. DELETE
  - C. DROP
  - D. ERASE
12. What does DDL stand for in SQL?
  - A. Data Definition Language
  - B. Data Duplication Language
  - C. Database Definition Language
  - D. Data Development Language
13. Which of the following is NOT a type of SQL join?
  - A. INNER JOIN
  - B. OUTER JOIN
  - C. SIDE JOIN
  - D. FULL JOIN
14. What command is used in SQL to save changes made during a transaction?
  - A. SAVE
  - B. COMMIT

- C. FINALIZE
  - D. COMPLETE
15. Which of the following operations is NOT part of CRUD operations in databases?
- A. Create
  - B. Read
  - C. Update
  - D. Concatenate
16. What does TCL in SQL stand for?
- A. Transaction Control Language
  - B. Table Creation Language
  - C. Tuple Control Language
  - D. Transaction Creation Language
17. Which Python function checks if an object is an instance of a class?
- A. type()
  - B. isinstance()
  - C. isobject()
  - D. isclass()
18. Which of the following is a valid SQLite data type?
- A. CHARACTER
  - B. TEXT
  - C. STRING
  - D. VARCHAR
19. What does ACID stand for in the context of database transactions?
- A. Atomicity, Consistency, Isolation, Durability
  - B. Accuracy, Clarity, Intensity, Dependability
  - C. Association, Consistency, Integrity, Durability
  - D. Atomicity, Clarity, Isolation, Dependability
20. In Python's SQLite3 module, what method is used to execute SQL commands?
- A. run()
  - B. perform()
  - C. execute()

D. do()

### 13. TERMINAL QUESTIONS

1. Explain the structure of a relational database and the importance of primary and foreign keys.
2. Describe the ACID properties of transactions in database systems.
3. Discuss the differences between INNER JOIN, LEFT JOIN, and FULL JOIN in SQL.
4. Explain the concept of database normalization and its benefits.
5. Describe the process of creating and using an index in a database.
6. Discuss the role of transactions in ensuring data integrity in databases.
7. Explain the difference between Data Definition Language (DDL) and Data Manipulation Language (DML) in SQL.
8. Describe the concept of database views and their uses.
9. Explain the importance of the sqlite3 module in Python for database interaction.
10. Discuss how stored procedures can be used in database management.
11. Write an SQL query to create a table named Employees with columns EmployeeID (Primary Key), FirstName, LastName, and Department.
12. Write a Python script to connect to an SQLite database named CompanyDB.
13. Using Python's sqlite3 module, write a script to insert the following data into the Employees table: EmployeeID=1, FirstName='John', LastName='Doe', Department='IT'.
14. Write an SQL query to fetch all records from the Employees table where the Department is 'IT'.
15. Write a Python function using sqlite3 to update the Department of an employee given the EmployeeID.
16. Write an SQL query to delete a record from the Employees table using a specific EmployeeID.
17. Using Python, write a script to begin a transaction, perform multiple database operations, and then commit the transaction.
18. Write an SQL command to add a new column Email to the Employees table.
19. Write a Python script to create a savepoint, perform some operations, and then rollback to the savepoint within a transaction.



20. Create an SQL query to perform an INNER JOIN between two tables, Employees and Departments, based on the Department column.

## 14. ANSWERS

### 14.1. Self-Assessment Questions

1. Structured
2. Primary
3. DML (Data Manipulation Language)
4. Join
5. BEGIN TRANSACTION
6. disk
7. sqlite3
8. Atomicity
9. Virtual
10. Durability
11. B) DELETE
12. A) Data Definition Language
13. C) SIDE JOIN
14. B) COMMIT
15. D) Concatenate
16. A) Transaction Control Language
17. B) isinstance()
18. B) TEXT
19. A) Atomicity, Consistency, Isolation, Durability
20. C) execute()

### 14.2. Terminal Questions

1. Understanding Relational Databases - This section introduces the concept of relational databases, emphasizing the importance of organizing data into tables consisting of rows and columns. It explains how each table represents a specific type of entity, with rows for instances of that entity and columns for attributes.

2. Transactions in SQL - Explore the "Transactions" section to understand the ACID properties that ensure transactions in database systems maintain data integrity, consistency, and isolation, even in the event of system failures or concurrent accesses.
3. SQL Joins - The "SQL Joins" subsection provides a comprehensive explanation of different types of joins, including INNER JOIN, LEFT JOIN, and FULL JOIN, and their roles in combining rows from two or more tables based on related columns.
4. Database Normalization - While not explicitly covered in the provided content, database normalization is a process of organizing database tables and their columns to reduce redundancy and dependency. It involves dividing large tables into smaller tables and defining relationships between them to ensure data integrity.
5. Creating and Using Indexes - Look into the "Advanced SQL Concepts" part for insights on creating and utilizing indexes in a database. Indexes are special lookup tables that database search engines can use to speed up data retrieval, significantly enhancing query performance.
6. Role of Transactions - The "Transactions" section will further clarify the pivotal role transactions play in maintaining the consistency and integrity of database operations, ensuring that all operations within a transaction block are successfully completed.
7. Differences between DDL and DML - The initial sections on "SQL Syntax" and subsequent discussions on DDL (Data Definition Language) and DML (Data Manipulation Language) provide clarity on their differences. DDL commands are used to define or modify database structures, whereas DML commands manage the data within those structures.
8. Database Views - In the "Advanced SQL Concepts" section, you'll find information on database views, which are virtual tables created by querying data from one or more real tables. Views simplify complex queries by presenting data in a specific format without altering the original data.
9. SQLite3 Module in Python - The "SQLite Module" section introduces the sqlite3 module in Python, explaining how it provides a lightweight disk-based database that

doesn't require a separate server process, making it ideal for embedding in applications.

10. Stored Procedures - While stored procedures are mentioned in the context of advanced SQL concepts, they are routines stored in the database that can be invoked and executed by the database engine. They help in performing complex operations and enhancing database performance.

11. CREATE TABLE Employees (EmployeeID int PRIMARY KEY, FirstName varchar(255), LastName varchar(255), Department varchar(255));

12. import sqlite3; conn = sqlite3.connect('CompanyDB.db');

13. c = conn.cursor(); c.execute("INSERT INTO Employees VALUES (1, 'John', 'Doe', 'IT')"); conn.commit();

14. SELECT \* FROM Employees WHERE Department = 'IT';

15. def update\_department(employee\_id, new\_department):

*conn = sqlite3.connect('CompanyDB.db')*

*c = conn.cursor()*

*c.execute("UPDATE Employees SET Department = ? WHERE EmployeeID = ?",  
(new\_department, employee\_id))*

*conn.commit()*

*conn.close()*

16. DELETE FROM Employees WHERE EmployeeID = ?; (Use the specific EmployeeID value in place of ? in your query execution.)

17. conn = sqlite3.connect('CompanyDB.db')

*conn.execute("BEGIN TRANSACTION;")*

*try:*

*# Perform multiple operations here*

*conn.commit()*

*except Exception as e:*

*conn.rollback()*

*finally:*

```
conn.close()
```

18. ALTER TABLE Employees ADD COLUMN Email varchar(255);

19. `conn = sqlite3.connect('CompanyDB.db')`

```
c = conn.cursor()
```

```
c.execute("BEGIN TRANSACTION;")
```

```
c.execute("SAVEPOINT sp1;")
```

```
# Perform some operations here
```

```
c.execute("ROLLBACK TO SAVEPOINT sp1;")
```

```
conn.commit()
```

```
conn.close()
```

20. SELECT Employees.\*, Departments.DepartmentName FROM Employees INNER JOIN Departments ON Employees.Department = Departments.DepartmentID;