HyperionDev

SQL and SQLite

Task

Visit our website

Introduction

In this task, you will learn a language used for manipulating and managing databases: Structured query language (SQL). Alongside mastering SQL, you will also gain hands-on experience with SQLite, a popular and lightweight database management system that uses SQL to interact with its data.

Structured query language (SQL)

SQL is a database language that is composed of commands that enable users to create databases or table structures, perform various types of data manipulation and data administration, as well as query the database to extract useful information. SQL is supported by all relational database management system (RDBMS) software. SQL is portable, which means that a user does not have to relearn the basics when moving from one RDBMS to another, because all RDBMSs use SQL in almost the same way.

SQL is easy to learn, as its vocabulary is relatively simple. Its basic command set has a vocabulary of fewer than 100 words. It is also a declarative language, which means that the user specifies what must be done and not how it should be done. Users do not need to know the physical data storage format or the complex activities that take place when an SQL command is executed in order to issue a command.

SQL functions fit into two general categories:

1. **Data definition language (DDL)** includes commands that enable users to define, edit, and delete data structures and schemas stored within a database management system. A database schema outlines the structure of a relational database, specifying how data is organised and connected. This includes defining the names and contents of tables, the types of data stored in each field, and how different tables relate to one another. Using commands like `CREATE`, `ALTER`, and `DROP`, a DDL can create new databases and tables, modify column definitions or table properties, add or remove constraints, and delete entire data structures when no longer needed. It also provides fine-grained control over the metadata systems that organise and categorise data.

2. **Data manipulation language (DML)** includes commands that enable users to access, modify, and manipulate data stored in a database. Key DML operations map to the CRUD acronym – create new entries, read and retrieve existing records, update or edit stored values, and delete data. With commands like `SELECT`, `INSERT, UPDATE`, and `DELETE`, DML provides powerful tools for curating, shaping, cleansing, and managing large datasets, such as query filtering on specific criteria. Manipulating data dynamically is vital for impactful analytics.

The commands in each category are given in the tables below. We will explore some of these commands in the rest of this lesson.

The table below lists the SQL DDL commands:

Command	**Description**
`CREATE SCHEMA AUTHORIZATION`	Creates a database schema.
`CREATE TABLE`	Creates a new table in the user's database schema.
`NOT NULL`	Ensures that a column will not have null values.
`UNIQUE`	Ensures that a column will not have duplicate values.
`PRIMARY KEY`	Defines a primary key for a table.
`FOREIGN KEY`	Defines a foreign key for a table.
`DEFAULT`	Defines a default value for a column when no value is given.
`CHECK`	Used to validate data in an attribute.
`CREATE INDEX`	Creates an index for the table.
`CREATE VIEW`	Creates a dynamic subset of rows or columns from one or more tables.
`ALTER TABLE`	Modifies a table (adds, modifies, or deletes attributes or constraints).
`CREATE TABLE AS`	Creates a new table based on a query in the user's database schema.
`DROP TABLE`	Permanently deletes a table.
`DROP INDEX`	Permanently deletes an index.

`DROP VIEW`	Permanently deletes a view.

Table source (Rob & Coronel, 2009)

The table below lists the SQL DML commands:

Command	**Description**
`INSERT`	Inserts rows into a table.
`SELECT`	Select attributes from rows in one or more tables or views.
`WHERE`	Restricts the selection of rows based on a conditional expression.
`GROUP BY`	Groups the selected rows based on one or more attributes.
`HAVING`	Restricts the selection of grouped rows based on a condition.
`ORDER BY`	Orders the selected rows based on one or more attributes.
`UPDATE`	Modifies an attribute's values in one or more tables' rows.
`DELETE`	Deletes one or more rows from a table.
`COMMIT`	Permanently saves data changes.
`ROLLBACK`	Restores data to its original values.
Comparison Operators	`=, <, >, <=, >=, <>`
Logical Operators	`AND, OR, NOT`
Special Operators	Used in conditional expressions.
`BETWEEN`	Checks whether an attribute value is within a range.

`IS NULL`	Checks whether an attribute value is null.
`LIKE`	Checks whether an attribute value matches a given string pattern.
`IN`	Checks whether an attribute value matches any value within a value list.
`EXISTS`	Checks whether a subquery returns any rows.
`DISTINCT`	Limits values to unique values.
`Aggregate Functions`	Used with `SELECT` to return mathematical summaries on columns.
`COUNT`	Returns the number of rows with non-null values for a given column.
`MIN`	Returns the minimum attribute value found in a given column.
`MAX`	Returns the maximum attribute value found in a given column.
`SUM`	Returns the sum of all values for a given column.
`AVG`	Returns the average of all values for a given column.

Table source (Rob & Coronel, 2009)

Creating tables

To create new tables in SQL, you use the `CREATE TABLE` statement. Pass all the columns you want in the table, as well as their data types, as arguments to the `CREATE TABLE` function. The table will be organised by columns and rows, like a spreadsheet. Each column is called a field or attribute, and has a field name which functions like the column heading. Each field holds data on a specific topic, where the field name usually indicates the topic. In the example table below, the fields are `StudentNumber`, `Name`, `Surname`, `CellNumber`, and `Address`. Each row, on the other hand, is called a record, and each record holds a full set of data for a particular entity/thing/person/etc. In the example table below, the entities we're storing data about are students, and each row holds a record of a single student's data.

StudentNumber	Name	Surname	CellNumber	Address
4f8149817	Liano	Charook	082 283 9009	3 Maple Str
5e9285991	Bianca	Manan	072 329 5571	17 Willow Ave
9b7744992	Cameron	Devilliers	072 410 9077	9 Birch Lane

The syntax of the `CREATE TABLE` statement is shown below:

```
CREATE TABLE table_name (

    column1 datatype constraint,

    column2 datatype constraint,

    column3 datatype constraint,

    ....

);
```

Note the optional constraint arguments. They are used to specify rules for data in a table. Constraints that are commonly used in SQL include:

- `NOT NULL`: Ensures that a specific column within a table cannot store null values, guaranteeing that data is always present in that column.
- `UNIQUE`: Ensures that all values in a specified column or combination of columns are distinct, preventing duplicate entries and maintaining data integrity.
- `DEFAULT`: Specifies a predefined value that is automatically inserted into a column if no other value is provided during data entry, ensuring data consistency and reducing null values.
- `INDEX`: Creates an index linked to a specific column or set of columns that is used to create and retrieve data from the database quickly. This works the same way as an index in a book, where keywords in the index enable us to quickly find those topics in the book, except that in a database the 'keyword' is a field name and indexing it helps the computer find and access the data in that field more quickly. For instance, if you frequently run queries to retrieve employees based on their `EmployeeID`, you should create an index on this column for faster lookup.

Additionally, note the use of a semicolon at the end of the statement. It indicates the completion of a single SQL command. While optional in some SQL environments, using semicolons consistently improves code readability and maintainability.

Let's look at another instance. To create a table called Employee that contains five columns (`EmployeeID, LastName, FirstName, Address, and PhoneNumber`), you would use the following SQL:

```
CREATE TABLE Employee (

    EmployeeID int,

    LastName varchar(255),

    FirstName varchar(255),

    Address varchar(255),

    PhoneNumber varchar(255)

);
```

The `EmployeeID` column is of type int and will, therefore, hold an integer value. The `LastName`, `FirstName`, `Address`, and `PhoneNumber` columns are of type `varchar` and will, therefore, hold characters. The number in brackets indicates the maximum number of characters, which in this case is 255.

The `CREATE TABLE` statement above will create an empty Employee table that will look like this:

EmployeeID	LastName	FirstName	Address	PhoneNumber

When creating tables, it is advisable to add a **primary key** to one of the columns, as this will help keep entries unique and will speed up `SELECT` queries. Primary keys must contain unique values and cannot contain null values. A table can only contain one primary key, however, the primary key may consist of a single column or a combination of multiple columns.

You can add a primary key when creating the `Employee` table as follows:

```
CREATE TABLE Employee (

    EmployeeID int NOT NULL,
    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Address varchar(255),

    PhoneNumber varchar(255),

    PRIMARY KEY (EmployeeID)

);
```

To name a primary key constraint and define a primary key constraint on multiple columns, you would use the following SQL syntax:

```
CREATE TABLE Employee (

    EmployeeID int NOT NULL,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Address varchar(255),

    PhoneNumber varchar(255),

    CONSTRAINT PK_Employee PRIMARY KEY (EmployeeID, LastName)

);
```

In the example above, there is only one primary key named `PK_Employee`. However, the value of the primary key is made up of two columns: `EmployeeID` and `LastName`.

Inserting rows

The table that we have just created is empty and needs to be populated with rows or records. We can add entries to a table using the `INSERT INTO` command.

There are two ways to write the `INSERT INTO` command: inserting values for specific columns and inserting values for all columns. It's generally a good practice to explicitly specify the columns for which you are providing values during an `INSERT` statement. Explicit column specification makes the code more readable and reduces the risk of errors, especially when the table schema changes or new columns are added.

1. **Inserting values for specific columns:**

If you only want to insert data into specific columns and want to specify both the column names and the corresponding values, you can use this method. The syntax is as follows:

```
INSERT INTO table_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);
```

For instance, to add an entry to the `Employee` table using this approach, you would write:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)

VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

This method is useful when you want to insert data into specific columns and leave other columns with default values or `NULL`. For example, as the `FirstName` and `Address` fields don't have a `NOT NULL` constraint, they do not necessarily have to be included when inserting. The `FirstName` and `Address` fields for this record will be `NULL`:

```
INSERT INTO Employee (EmployeeID, LastName, PhoneNumber)

VALUES (1321, 'Jones', '0827546787');
```

To insert multiple rows using this method, you can provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. Here's an instance:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)

VALUES

      (1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),

      (5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),

      (9112, 'Davies', 'Emily', '25 Maple Ave', '0876543210');
```

2. **Inserting values for all columns:**

   If you want to add values for all the columns in the table and the order of the values matches the exact order of the columns, you can simply not specify the column names. The syntax for this approach is as follows:

```
INSERT INTO table_name

VALUES (value1, value2, value3, ...);
```

For instance, to add an entry to the `Employee` table, you would do the following:

```
INSERT INTO Employee

VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

To insert multiple rows using this method, you can simply provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. For example:

```
INSERT INTO Employee

VALUES

      (1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),

      (5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),

      (9112, 'Davies', 'Emily', '25 Maple Ave', '0876543210');
```

Retrieving data from a table

The `SELECT` statement is used to fetch data from a database. The data returned is stored in a result table, known as the result set. The syntax of a `SELECT` statement is as follows:

```
SELECT column1, column2, …

FROM table_name;
```

Here, `column1, column2, ...` refer to the column names of the table from which you want to select data. The following example below selects the `FirstName` and `LastName` columns from the `Employee` table:

```
SELECT FirstName, LastName

FROM Employee;
```

If you want to select all the columns and rows in the `Employee` table, use the following syntax:

```
SELECT * FROM Employee;
```

The asterisk (*) indicates that we want to fetch all of the columns, without excluding any of them.

You can also order and filter the data that is returned when using the `SELECT` statement using the `ORDER BY` and `WHERE` commands.

Order by

You can use the `ORDER BY` command to sort the results returned in ascending or descending order based on one or more columns. The `ORDER BY` command sorts the records in ascending order by default. You need to use the `DESC` keyword to sort the records in descending order.

The `ORDER BY` syntax is as follows:

```
SELECT column1, column2, …

FROM table_name

ORDER BY column1, column2, ... ASC|DESC;
```

The example below selects all employees in the `Employee` table and sorts them in descending order (4, 3, 2, 1 … for numbers, and Z…A for letters), based on the values in the `FirstName` column:

```
SELECT *

FROM Employee

ORDER BY FirstName DESC;
```

Where

The `WHERE` clause allows us to filter data depending on a specific condition. The syntax of the `WHERE` clause is as follows:

```
SELECT column1, column2, …
```

```
FROM table_name
WHERE condition;
```

The following SQL statement selects all the employees with the first name `'John'`, in the `Employee` table:

```
SELECT *
FROM Employee
WHERE FirstName = 'John';
```

Note that SQL requires single quotes around text values; however, you do not need to enclose numeric fields in quotes. You should also note that, for conditions in SQL, we use a single '=', which is equivalent to the '==' used in Python, JavaScript, and Java.

You can use logical operators (`AND, OR`) and comparison operators (`=,<,>,<=,>=,<>`) to make `WHERE` conditions as specific as you like.

For instance, suppose you have the following table that contains the most sold albums of all time:

Artist	**Album**	**Released**	**Genre**	**sales_in_millions**
Michael Jackson	*Thriller*	1982	pop	70
AC/DC	*Back in Black*	1980	rock	50
Pink Floyd	*The Dark Side of the Moon*	1973	rock	45
Whitney Houston	*The Bodyguard*	1992	soul	44

You can select the records that are classified as rock and have sold under 50 million copies by simply using the `AND` operator as follows:

```
SELECT *
FROM albums
WHERE Genre = 'rock' AND sales_in_millions <= 50
```

```
ORDER BY Released
```

WHERE statements also support some special operators to customise queries further:

- IN: Compares the column to multiple possible values and returns true if it matches at least one.
- BETWEEN: Checks if a value is within an inclusive range.
- LIKE: Searches for a pattern match, which is specific character patterns within text data. For instance, using the pattern specification S% searches for all text that starts with an 'S' followed by any number of other characters (no matter which characters). The percentage sign is referred to as a wildcard, which represents any characters. An underscore represents any single character. For example, J__n can be used to search for any text that starts with a 'J' followed by two characters and ends with an 'n'.

For example, if we want to select the pop and soul albums from the table above, we can use:

```
SELECT *

FROM albums

WHERE Genre IN ('pop', 'soul');
```

Or, if we want to get all the albums released between 1975 and 1985, we can use:

```
SELECT *

FROM albums

WHERE Released BETWEEN 1975 AND 1985;
```

We can also find all albums sung by artists whose names start with an M:

```
SELECT *

FROM albums

WHERE Artist LIKE 'M%';
```

Using aggregate functions

SQL has many functions that are helpful. Some of the most regularly used ones are:

- `COUNT()`: Returns the number of rows.
- `SUM()`: Returns the total sum of a numeric column.
- `AVG()`: Returns the average of a set of values.
- `MIN() / MAX()`: Gets the minimum or maximum value from a column.
- `GROUP BY`: Group rows are returned by a query and divided into summary rows based on the values in one or more columns.

For example, to get the most recent year in the album table, we can use:

```
SELECT MAX(Released)

FROM albums;
```

Or to get the number of albums released between 1975 and 1985, we can use:

```
SELECT COUNT(*)

FROM albums

WHERE Released BETWEEN 1975 AND 1985;
```

The `GROUP BY` clause is often used in conjunction with the other functions. The following SQL command calculates the total sales by genre:

```
SELECT Genre, SUM(sales_in_millions) AS total_sales

FROM albums

GROUP BY Genre;
```

Retrieving data across multiple tables

In complex databases, there are often several tables connected in some way. A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them. Look at the two tables below:

`VideoGame` table:

ID	**Name**	**DeveloperID**	**Genre**
1	*Super Mario Bros.*	2	platformer
2	*World of Warcraft*	1	MMORPG
3	*The Legend of Zelda*	2	adventure

`GameDeveloper` table:

ID	**Name**	**Country**
1	Blizzard	USA
2	Nintendo	Japan

The `VideoGame` table contains information about various video games, while the `GameDeveloper` table contains information about the developers of the games. The `VideoGame` table has a `DeveloperID` column that holds the game developer's ID, which represents the ID of the respective developer from the `GameDeveloper` table. `DeveloperID` points to a specific developer with a matching ID (primary key) in the `GameDeveloper` table; a column that can be used to link two tables like this is called a foreign key. A foreign key in one table **must always correspond** to a primary key in another table.

From the tables above, we can see that a developer called Blizzard from the USA developed the game entitled *World of Warcraft*. The foreign key logically links the two tables, and allows us to access and use the information stored in both of them at the same time.

If we want to create a query that returns everything we need to know about the games, we can use `INNER JOIN` to acquire the columns from both tables.

```sql
SELECT VideoGame.Name, VideoGame.Genre, GameDeveloper.Name, GameDeveloper.Country

FROM VideoGame

INNER JOIN GameDeveloper

ON VideoGame.DeveloperID = GameDeveloper.ID;
```

Notice that in the `SELECT` statement above, we specify the name of the table and the column from which we want to retrieve information, and not just the name of the column as we have done previously. This is because we are getting information from more than just one table, and tables may have columns with the same names. In this case, both the table `VideoGame` and the table `GameDeveloper` contain columns called `Name`. In the next section, you will see how to use aliases to further address this issue.

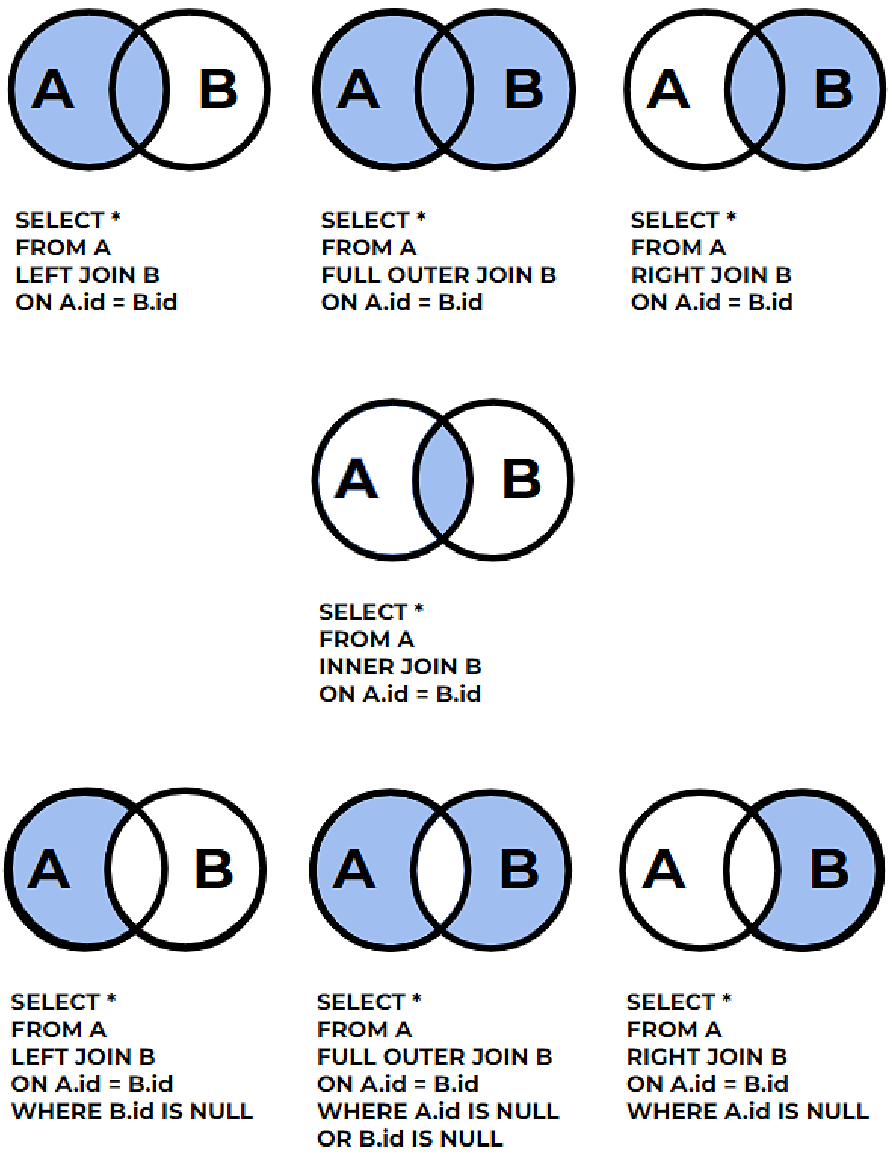
Also, notice that we use the `ON` clause to specify how we link the foreign key in one table to the corresponding primary key in the other table. The query above would result in the following dataset being returned:

`VideoGame.Name`	`VideoGame.Genre`	`GameDeveloper.Name`	`GameDeveloper.Country`
Super Mario Bros.	platformer	Nintendo	Japan
World of Warcraft	MMORPG	Blizzard	USA
The Legend of Zelda	adventure	Nintendo	Japan

The `INNER JOIN` is the simplest and most common type of `JOIN`. However, there are many other different types of joins in SQL. Let's take a look at these and what they do.

- `INNER JOIN`: Returns records that have matching values in both tables.
- `LEFT JOIN`: Returns all records from the left table, and the matched records from the right table.
- `RIGHT JOIN`: Returns all records from the right table, and the matched records from the left table.
- `FULL JOIN`: Returns all records when there is a match in either the left or right table.

The figure below provides a graphical representation of the above joins, plus some extras that are less common. Look carefully at the Venn diagrams and ensure you understand exactly which records would be returned from tables A and B if the given SQL code was executed.



```
SELECT *
FROM A
LEFT JOIN B
ON A.id = B.id
```

```
SELECT *
FROM A
FULL OUTER JOIN B
ON A.id = B.id
```

```
SELECT *
FROM A
RIGHT JOIN B
ON A.id = B.id
```

```
SELECT *
FROM A
INNER JOIN B
ON A.id = B.id
```

```
SELECT *
FROM A
LEFT JOIN B
ON A.id = B.id
WHERE B.id IS NULL
```

```
SELECT *
FROM A
FULL OUTER JOIN B
ON A.id = B.id
WHERE A.id IS NULL
OR B.id IS NULL
```

```
SELECT *
FROM A
RIGHT JOIN B
ON A.id = B.id
WHERE A.id IS NULL
```

Graphical representation of common SQL joins

Using aliases

Notice that in the `VideoGame` and `GameDeveloper` tables, there are two columns called `Name`. This can become confusing. Aliases are used to give a table or column a temporary name. An alias only exists for the duration of the query and is often used to make column names more readable.

The alias column syntax is:

```
SELECT column_name AS alias_name
```

```
FROM table_name;
```

The `alias` table syntax is:

```
SELECT column_name(s)

FROM table_name AS alias_name;
```

The following SQL statement creates an alias for the `Name` column from the `GameDeveloper` table:

```
SELECT Name AS Developer

FROM GameDeveloper;
```

See how aliases have been used below:

```
SELECT games.Name, games.Genre, devs.Name AS Developer, devs.Country

FROM VideoGame AS games

INNER JOIN GameDeveloper AS devs

ON games.DeveloperID = devs.ID;
```

As you can see, this starts to get quite difficult to follow and read, as the query statements grow. In SQL, the `WITH` clause, also known as a Common Table Expression (CTE), allows you to define a temporary result set that you can reference within the context of a larger SQL query. It enhances the readability and maintainability of complex queries by breaking them down into smaller, named, and reusable components. The `WITH` clause example is focused solely on retrieving developer names from the `GameDeveloper` table:

```
WITH GameInfo AS (

       SELECT  games.ID,  games.Name  AS  GameName,  games.Genre,  devs.Name  AS
Developer

    FROM VideoGame AS games

    INNER JOIN GameDeveloper AS devs ON games.DeveloperID = devs.ID

)
```

The `GameInfo` CTE includes information about the video games and their developers by joining the `VideoGame` and `GameDeveloper` tables. The main query then selects distinct developer names from the CTE:

```
SELECT DISTINCT Developer

FROM GameInfo;
```

This approach allows you to create a CTE that encompasses the logic of joining the tables and then reuse the CTE in subsequent queries.

Updating data

The `UPDATE` statement is used to modify the existing rows in a table.

To use the `UPDATE` statement, you:

- Choose the table with the row you want to change.
- Set the new value(s) for the desired column(s).
- Select which of the rows you want to update using the `WHERE` statement. If you omit this, all rows in the table will change.

The syntax for the update statement is:

```
UPDATE table_name

SET column1 = value1, column2 = value2, …

WHERE condition;
```

Take a look at the following `Customer` table:

CustomerID	CustomerName	Address	City
1	Maria Anderson	23 York St	New York
2	Jackson Peters	124 River Rd	Berlin
3	Thomas Hardy	455 Hanover Sq	London
4	Kelly Martins	55 Loop St	Cape Town

The following SQL statement updates the first customer `(CustomerID = 1)` with a new address and a new city:

```
UPDATE Customer

SET Address = '78 Oak St', City = 'Los Angeles'

WHERE CustomerID = 1;
```

Deleting rows

Deleting a row is a simple process. All you need to do is select the right table and row that you want to remove. The `DELETE` statement is used to delete existing rows in a table.

The `DELETE` statement syntax is as follows:

```
DELETE FROM table_name

WHERE condition;
```

The following statement deletes the customer `Jackson Peters` from the `Customer` table:

```
DELETE FROM Customer

WHERE CustomerName = 'Jackson Peters';
```

You can also delete all the data inside a table, i.e. all rows, without deleting the table:

```
DELETE FROM table_name;
```

When dealing with foreign keys, it's important to understand the concept of referential integrity. Referential integrity ensures that relationships between tables remain valid, meaning that a foreign key in one table must correspond to a primary key in another table. When you want to delete records from a table that is referenced by foreign keys in other tables, you need to be careful to maintain referential integrity.

Therefore, deletion is a last resort because it can lead to data inconsistency and integrity issues if not handled carefully. Always ensure that deleting records will not violate referential integrity and consider alternative approaches to ensure that the database remains well-structured and consistent.

Deleting tables

The `DROP TABLE` statement is used to remove every trace of a table from a database. The syntax is as follows:

```
DROP TABLE table_name;
```

For instance, if we want to delete the table `Customer`, we do the following:

```
DROP TABLE Customer;
```

If you want to delete the data inside a table, but not the table itself, you can use the `TRUNCATE TABLE` statement:

```
TRUNCATE TABLE table_name;
```

While the `TRUNCATE TABLE` and `DELETE FROM` statements perform similarly in terms of removing data from a table, there are a few important differences to consider when deciding which statement to use:

`TRUNCATE TABLE`	**`DELETE FROM table_name`**
A data definition language command (DDL) command that's automatically committed. This means data cannot be recovered, as it is a non-transactional operation (it cannot be rolled back	A data manipulation language (DML) command that is not automatically committed. This means data can be recovered, as it supports transactional operations and logs row-level details.

once executed) and does not generate any undo logs.	
Removes all data rows and resets certain attributes associated with the table's structure, such as the value of the identity columns, to their initial values.	Removes all data rows while retaining attributes associated with the table's structure.
Faster speed and performance, as it removes all rows as a single operation.	Slower speed and performance, as it removes each row one by one.
Use case: If you have a transactional database where you want to do a bulk purge of all the transactional data accumulated over time and reclaim storage space.	**Use case**: If you want to delete specific data, based on certain conditions, with the added flexibility to roll back if needed.

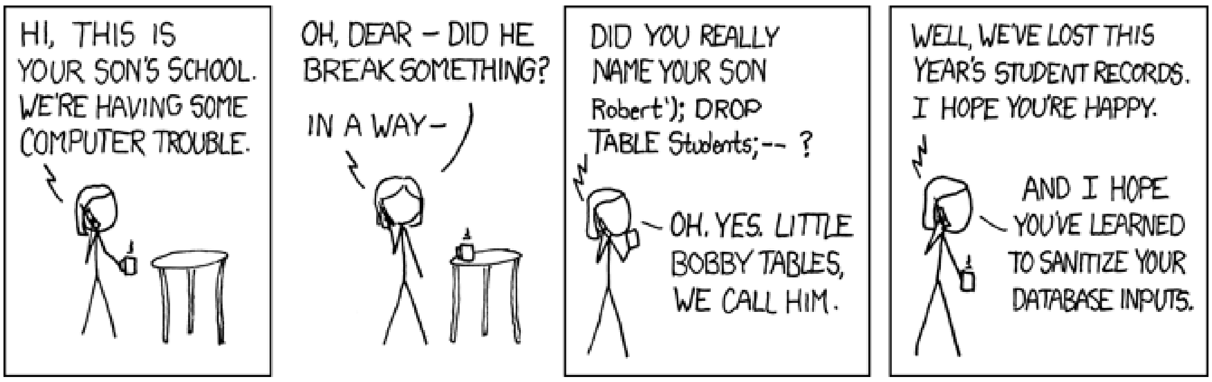
Take note

The behaviour of `TRUNCATE TABLE` statement can vary across different database providers. While some database systems support rollback, others may not e.g. in the Oracle DB. Therefore, it's important to consult the documentation specific to your database to understand whether rollback is supported. In SQLite, the `TRUNCATE` statement is not supported. Instead, you can use the `DELETE FROM` statement without a `WHERE` clause to achieve similar results.

Extra resource

If you would like to know more about SQL, we highly recommend reading the book ***Database Design – 2nd Edition*** by Adrienne Watt. Chapters 15 and 16, and Appendix C of this book provide more detail regarding what has been covered in this task.

Here's a little SQL humour related to deleting tables. Sanitising database inputs consists of removing any unsafe characters from user inputs. See if you can understand what happens in the cartoon below and why. What does “--” mean in SQL (see panel three), and why is this relevant? See if you can find out.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Comic (n.d.)

SQLite

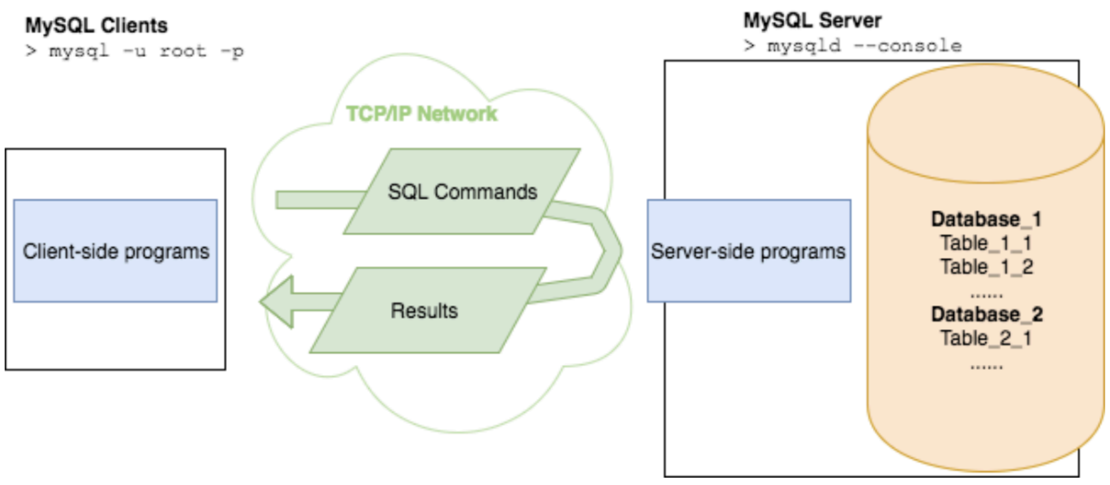
SQLite is a lightweight, self-contained, and serverless database management system (DBMS). Unlike other database systems like MySQL or PostgreSQL, SQLite doesn’t require a separate server process, making it an ideal choice for small to medium-sized applications, embedded systems, and testing. At its core, SQLite uses SQL (Structured Query Language) to interact with the database. SQL is the standard language for managing and manipulating databases, and SQLite supports a wide range of SQL commands.

SQLite features

Important features to note about SQLite are that it is self-contained, serverless, transactional, and requires zero-configuration to run. Let’s look more closely at these properties.

- **Self-contained:** This means that SQLite does not need much support from the operating system or external libraries. This makes it suitable for use in embedded devices like mobile phones, iPods, and game devices that lack the infrastructure provided by a regular computer. In Python, you can access SQLite databases using the sqlite3 module, which is part of Python's standard library. This means you do not need to manage SQLite source code files like sqlite3.c or sqlite3.h yourself. Instead, simply import the sqlite3 module to interact with SQLite databases in your Python projects.

- **Serverless:** In most cases, RDBMSs require a separate server to receive and respond to requests sent from the client, as shown in the diagram below.



SQLite Is Severless (SQlite, 2024)

Such systems include MySQL, MariaDB, and the Java Database Client (JDBC). These clients have to use the TCP/IP protocol to send and receive responses. This is referred to as the Client/Server architecture. SQLite does not make use of a separate server and, therefore, does not utilise Client/Server architecture. Instead, the entire SQLite database is embedded into the application that needs to access the database.

- **Transactional:** all transactions in SQLite are atomic, consistent, isolated, and durable (**ACID**-compliant). In other words, if a transaction occurs that attempts to make changes to the databases, the changes will either be made in all the appropriate places (in all linked tables and affected rows) or not at all. This is to ensure data integrity (i.e. avoid conflicting records in different places due to some being updated and others not).
- **Zero configuration required:** You do not need to install SQLite prior to using it in an application or system. This is because of the previously described serverless characteristic.

Python's SQLite module

It is easy to create and manipulate databases with Python. To enable the use of SQLite with Python, the Python standard library includes a module called `sqlite3`. To use this module, we need to add an `import` statement to our Python script:

```
import sqlite3
```

We can then use the function `sqlite3.connect()` to connect to the database. We pass the name of the database file to this function to open or create the database.

```
# Creates or opens a file called student_db with an SQLite3 DB
db = sqlite3.connect("student_db.db")
```

Creating and deleting tables

To make any changes to the database, we need a **cursor object**, which is an object that is used to execute SQL statements. Next, we use `.commit()` to save changes to the database. It is important to remember to commit changes since this ensures the atomicity of the database. If you close the connection using `close()` or the connection to the file is lost, changes that have not been committed will be lost.

Below we create a student table with `id`, `name`, and `grade` columns:

```
cursor = db.cursor()  # Get a cursor object

# Execute a SQL command to create the student table
cursor.execute('''
    CREATE TABLE student(
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database to ensure they are saved
db.commit()
```

In the above code snippet, a cursor object is obtained from the database connection (db). Subsequently, a SQL query is executed using the cursor to create a new table named "student" with columns named id (as the primary key), name, and grade. The `db.commit()` statement is used to commit the transaction, finalising the table creation in the database.

Using IF NOT EXISTS

When working with databases and creating tables, it is often necessary to ensure that a table does not already exist before attempting to create the table. The `IF NOT EXISTS` clause can be used to create a table only if it does not already exist, which helps prevent errors that might occur if the table is already present in the database.

Let's have a look at an example demonstrating how the `CREATE TABLE` statement can be modified to include the `IF NOT EXISTS` clause, ensuring the table is created only if it does not already exist.

```
# Get the cursor object
cursor = db.cursor()

# Create the student table if it does not exist
cursor.execute('''
    CREATE TABLE IF NOT EXISTS student (
        id INTEGER PRIMARY KEY,
        name TEXT,
        grade INTEGER
    )
''')

# Commit the changes to the database
db.commit()
```

Always remember that the `commit()` function is invoked on the `db` object, not the `cursor` object. If we type `cursor.commit()`, we will get the following error message:

```
AttributeError: 'sqlite3.Cursor' object has no attribute 'commit'
```

Inserting into the database

To insert data into a database we use prepared statements. This is a secure method for executing SQL queries with Python. Prepared statements involve using placeholders, such as "?", in your SQL query instead of directly including data. This approach ensures that your data is handled safely and efficiently. Avoid using string operations or concatenation to construct SQL queries, as these methods can be less secure.

In this example, we are going to insert two students into the database; whose information is stored in Python variables.

```
name1 = 'Andres'
grade1 = 60

name2 = 'John'
grade2 = 90

# Insert student 1
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name1, grade1))

print('First user inserted')

# Insert student 2
cursor.execute('''
    INSERT INTO student(name, grade)
    VALUES (?, ?)
''', (name2, grade2))

print('Second user inserted')

db.commit()
```

In the example above, the values of the Python variables are passed inside a **tuple**. You could also use a dictionary with the named style placeholder:

```
name3 = 'Sheila'
grade3 = 40

# Insert student 3 using named parameters
cursor.execute('''
    INSERT INTO student (name, grade)
    VALUES (:name, :grade)
''', {'name': name3, 'grade': grade3})

print('Third user inserted')
```

If you need to insert several users, use `executemany` and a list with the tuples:

```
students_grades = [(name1, grade1), (name2, grade2), (name3, grade3)]

cursor.executemany(
    '''INSERT INTO student(name, grade) VALUES(?, ?)''', students_grades
)

db.commit()
```

Each record inserted into the table gets a unique id value starting at one and ascending in increments of one for each new record. If you need to get the id of the row you just inserted, use `lastrowid:`

```
# Get the ID of the last inserted row
```

```
last_row_id = cursor.lastrowid
print(f'Last row ID: {last_row_id}')
```

Use `rollback()` to roll back any change to the database since the last call to commit:

```
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (65, 2))

db.rollback()
```

Retrieving data

To retrieve data, execute a `SELECT` SQL statement against the `cursor` object and then use `fetchone()` to retrieve a single row or `fetchall()` to retrieve all the rows.

Here is an example using `fetchone()` to retrieve the first student record that matches the specified ID:

```
# Define the ID of the student we want to retrieve
id = 3

# Execute a query to select the name and grade of the student with the
specified ID
cursor.execute('''SELECT name, grade FROM student WHERE id = ?''', (id,))

# Fetch the first row that matches the query
student = cursor.fetchone()

# Print the retrieved student's name and grade
print(student)
```

Note: When using a single variable in a query with placeholders, ensure you include a comma to create a single-element tuple for example '`(id,)`'. This is necessary for the query to work correctly.

If you use `fetchone()` and there are multiple rows that match the criteria, only the first one will be retrieved. If you expect multiple rows, rather use `fetchall()`.

To retrieve all student records where the grade is below a specified threshold we can use `fetchall()` as shown below:

```
# Define the grade threshold
grade_threshold = 80

# Execute a query to select all students with grades less than the specified
```

```python
threshold
cursor.execute('SELECT name, grade FROM student WHERE grade < ?',
(grade_threshold,))

# Fetch all rows that match the query
students = cursor.fetchall()

# Print each student's name and grade
print(f'Students with a grade less than {grade_threshold}:')
for student in students:
    print(f'{student[0]} : {student[1]}')
```

In the above example, the code selects all student names and grades from the table where each student's grade is less than a specified threshold. This query returns a result set, which is a list of tuples, with each tuple containing a pair of student names and their corresponding grades. By looping through this list, you can use indexing (`student[0]` to access the name and `student[1]` to access the grade) to retrieve and display the name and grade for each student.

Alternatively you can use the `cursor` object which works as an iterator, invoking `fetchall()` automatically:

```python
cursor.execute('''SELECT name, grade FROM student''')

# Iterate over the result set returned by the query
for row in cursor:
    # Each 'row' is a tuple where row[0] is the student's name and row[1]
    # is their grade.
    # Print the student's name and grade in a formatted string
    print(f'{row[0]} : {row[1]}')
```

Updating and deleting data

Updating or deleting data is similar to inserting data:

```python
# Update user with id 1
grade = 100
user_id = 1
cursor.execute('''UPDATE student SET grade = ? WHERE id = ?''', (grade,
user_id))

# Delete user with id 2
user_id = 2
cursor.execute('''DELETE FROM student WHERE id = ?''', (user_id,))
```

```python
# Drop the student table
cursor.execute('''DROP TABLE student''')

# Commit the changes
db.commit()
```

When we are done working with the db, we need to close the connection. Failing to close the connection could result in issues such as incomplete transactions, data corruption, and resource leaks.

```python
db.close()
```

SQLite database exceptions

It is very common for exceptions to occur when working with databases, so it is important to handle these exceptions in your code.

In the example below, we use a `try/except/finally` clause to catch any exception in the code. We put the code that we would like to execute, but that may throw an exception (or cause an error) in the `try` block. Within the `except` block, we write the code that will be executed if an exception does occur. If no exception is thrown, the except block will be ignored. The `finally` clause will always be executed, whether an exception was thrown or not. When working with databases, the `finally` clause is very important, because it always closes the database connection correctly. View this resource to find out more **about exceptions**.

```python
import sqlite3

try:
    # Creates or opens a file called student_db with an SQLite3 DB
    db = sqlite3.connect('student_db.db')
    # Get a cursor object
    cursor = db.cursor()
    # Checks if the table "users" exists and if not creates it
    cursor.execute(
        '''
        CREATE TABLE IF NOT EXISTS
        users(id INTEGER PRIMARY KEY, name TEXT, grade INTEGER)
        '''
    )
    # Commit the change
    db.commit()
```

```
# Catch the exception
except Exception as e:
    # Roll back any change if something goes wrong
    db.rollback()
    raise e
finally:
    # Close the db connection
    db.close()
```

Notice that the `except` block of our `try/except/finally` clause in the example above will be executed if any type of error occurs:

```
 # Catch the exception
except Exception as e:
    raise e
```

This is called a catch-all clause. In a real application, you should catch a specific exception. To see what type of exceptions could occur, see **DB-API 2.0 exceptions**.

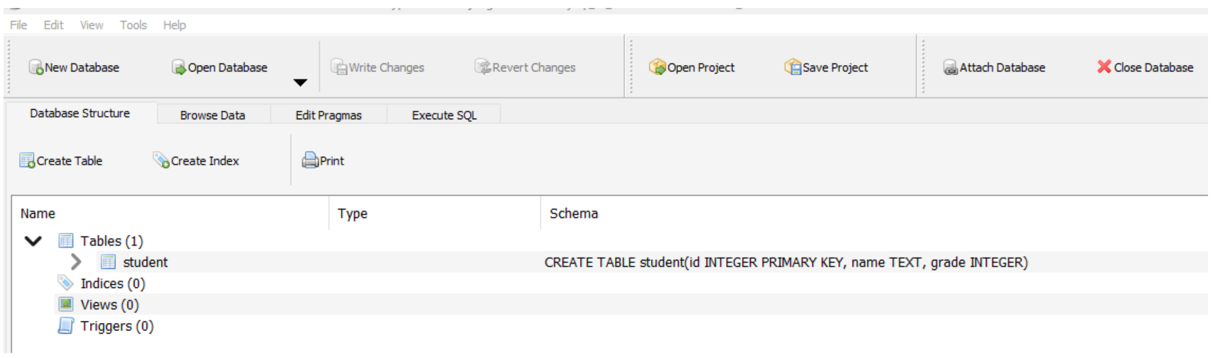
DB Browser for SQLite

DB Browser for SQLite is a free, open-source tool that allows you to interactively browse and manage databases created with SQLite, a popular database management system. With DB Browser, you can easily view, edit, and manipulate the data stored in your SQLite databases, as well as create new tables, indices, and relationships between them. It's a great way to explore and understand the structure and content of your databases, and it's available for Windows, macOS, and Linux platforms **here**.

Visualising our student_db database

Now, we want to visualise the **student_db.db** database we created in the above SQLite operations. To do this, we are going to import the **student_db.db** database as follows:

1. Open the DB Browser for SQLite.
2. To import a database, click on "File" in the menu bar, and then select "Open Database..." from the drop-down menu.
3. In the "Open File" dialogue box, navigate to the location where your database file is saved (it should have a **.db** extension) and select it.
4. Click "Open" to import the database into DB Browser for SQLite.
5. The Database Structure tab should then become visible, as seen below.



Name	Type	Schema
Tables (1)		
student		CREATE TABLE student(id INTEGER PRIMARY KEY, name TEXT, grade INTEGER)
Indices (0)		
Views (0)		
Triggers (0)		

Once the database is imported, you can start exploring it by clicking on the different tabs in the main window.

- The "Database Structure" tab displays all the tables in the database, along with their columns and data types.
- The "Browse Data" tab shows the actual data stored in each table.
- The "Execute SQL" tab allows you to execute SQL queries against the database.

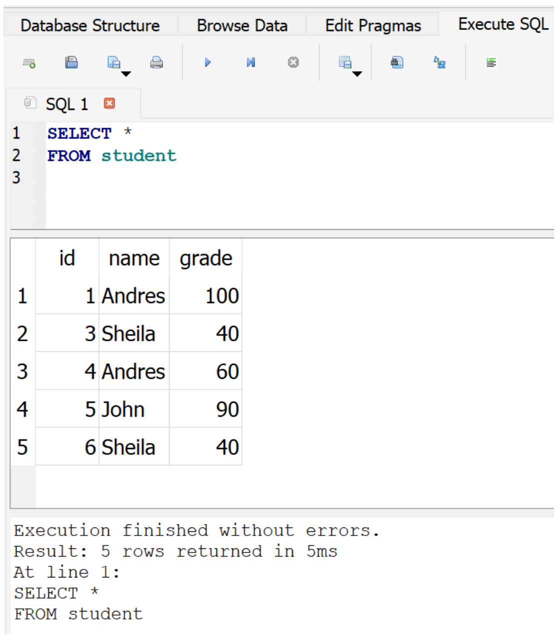
Running SQL queries

If you navigate to the “Execute SQL” tab, you will be presented with a three-paned window and a small toolbar. The initial pane is labelled “SQL 1”. This is where we will type our queries as follows:

1. In the “SQL 1” pane, input the following:

```
SELECT * FROM student
```

2. In the small toolbar, click the Run/Play button.
3. The query should execute and present the student table from the student_db database.



```sql
SELECT *
FROM student
```

	id	name	grade
1	1	Andres	100
2	3	Sheila	40
3	4	Andres	60
4	5	John	90
5	6	Sheila	40

```
Execution finished without errors.
Result: 5 rows returned in 5ms
At line 1:
SELECT *
FROM student
```

DB Browser for SQLite stands out as a valuable tool for efficiently managing SQLite databases. Whether you are exploring existing databases or creating new structures, this free and open-source application provides a user-friendly interface across various operating systems. With its versatility and accessibility, DB Browser for SQLite proves to be an indispensable companion for database exploration and manipulation.

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select "Request Review", the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you've done that, feel free to progress to the next task.

Take note

Read and run the accompanying example files provided before doing the task to become more comfortable with the concepts covered in this task.

Auto-graded task 1

Answer the following questions:

- Go to the **DB Fiddle Online SQL Editor**. This is where you can write and test your SQL code. Once you are happy with your code, paste it into a text file and save the file in your task folder as **Student.txt**.
- Write the SQL code to create a table called `Student`. The table structure is summarised in the table below.

Note that `STU_NUM` must be set up as the primary key.

Attribute Name	**Data Type**
STU_NUM	CHAR(6)
STU_SNAME	VARCHAR(15)
STU_FNAME	VARCHAR(15)
STU_INITIAL	CHAR(1)
STU_STARTDATE	DATE
COURSE_CODE	CHAR(3)
PROJ_NUM	INT(2)

- After you have created the table, write the SQL code to enter the following rows of data into the table as below:

STU_ NUM	STU_ SNAME	STU_ FNAME	STU_ INITIAL	STU_ STARTDATE	COURSE_ CODE	PROJ_ NUM
01	Snow	Jon	E	2014-04-05	201	6
02	Stark	Arya	C	2017-07-12	305	11
03	Lannister	Jamie	C	2012-09-05	101	2
04	Lannister	Cercei	J	2012-09-05	101	2
05	Greyjoy	Theon	I	2015-12-09	402	14
06	Tyrell	Margaery	Y	2017-07-12	305	10
07	Baratheon	Tommen	R	2019-06-13	201	5

- Write the SQL code to return all records which have a `COURSE_CODE` of 305.
- Write the SQL code to change the course code to 304 for the person whose student number is 07.
- Write the SQL code to delete the row of the person named Jamie Lannister, who started on 5 September 2012, whose course code is 101 and project number is 2. Use logical operators to include all of the information given in this problem.
- Write the SQL code to change the `PROJ_NUM` to 14 for all those students who started before 1 January 2016 and whose course code is at least 201.
- Write the SQL code that will delete the `Student` table entirely. Hint: Use `DROP TABLE`.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Auto-graded task 2

Follow these steps:

- Create a Python file called **database_manip.py**. Write the code to do the following tasks:
  - Create a table called `python_programming`.
  - Insert the following new rows into the `python_programming` table:

id	**name**	**grade**
55	Carl Davis	61
66	Dennis Fredrickson	88
77	Jane Richards	78
12	Peyton Sawyer	45
2	Lucas Brooke	99

  - Select all records with a `grade` between 60 and 80.
  - Change Carl Davis's `grade` to 65.
  - Delete Dennis Fredrickson's row.
  - Change the `grade` of all students with an `id` greater than 55 to 80.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.

Reference list

Rob, P., & Coronel, C. (2009). *Database Systems: Design, Implementation, and Management*, (8th ed.). Boston, Massachusetts: Course Technology.

SQLite. (2024). *SQLite is Severless*. SQLite. **https://www.sqlite.org/serverless.html**

Xkcd. (n.d.) *Exploits of a Mom*. Xkcd. **https://xkcd.com/327/**