



TASK

Working with SQL

[Visit our website](#)

Introduction

WELCOME TO THE WORKING WITH SQL TASK!

In this task, you will learn the language used for manipulating and managing databases: Structured Query Language (SQL). In addition, you will be introduced to a public domain relational database engine, SQLite.



Get in touch
Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at <https://discord.com/invite/hyperdev> where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

INTRODUCTION TO SQL

Structured Query Language (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 DBMS 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 will use SQL in almost the same way.

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

According to Rob and Coronel (2009), SQL functions fit into two general categories:

1. SQL is a data definition language (DDL): it includes commands that allow one to define or create database objects (such as tables, indexes and views). SQL also includes commands to define access rights to the database objects created.
2. SQL is a data manipulation language (DML): it includes commands that allow one to manipulate the data stored in the database. For example, commands to insert, update, delete and retrieve data.

The table below lists the SQL data definition 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, p. 225)

The table below lists the SQL data manipulation 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: Adapted from (Rob & Coronel, 2009, p. 225)

We will consider some of these commands in the rest of this task.

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 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 column cannot have a null value
- UNIQUE: Ensures that all values in a column are different
- DEFAULT: Sets a default value for a column when no value is specified
- INDEX: Creates an index which is used to create and retrieve data from the database very quickly

To create a table called Employee, for example, 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's 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 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 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: ID 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:

1. Do not specify the column names where you intend to insert the data. This can be done if you are adding values for all of the columns of the table. However, you should ensure that the order of the values is in the same order as the columns in the table. The syntax will be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

For example, to add an entry to the Employee table (created previously with SQL), you would do the following:

```
INSERT INTO employee
VALUES (1, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

2. The other way to insert data into a table is to specify both the column names and the values to be inserted. The syntax will be as follows:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

Therefore, to add an entry to the Employee table using this method, you would do the following:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address, PhoneNumber)
VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

The second method for inserting data into a table is often used when you don't want to add data to all the columns in the table.

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;
```

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

```
SELECT FirstName, LastName  
FROM Employee;
```

If you want to select all the columns in the table, you use the following syntax:

```
SELECT * FROM table_name;
```

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

For example, suppose you have the following table which 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 only 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 further customise queries:

- 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

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;
```

AGGREGATE FUNCTIONS

SQL has many functions that do all sorts of helpful stuff. 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

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(album)
WHERE released BETWEEN 1975 AND 1985;
```

RETRIEVING DATA ACROSS MULTIPLE TABLES

In complex databases, there are often several tables connected to each other 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

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

GameDeveloper

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 ID, which represents the ID of the respective developer from the GameDeveloper table. DeveloperID is a foreign key that points to a specific developer with a matching ID (primary key) in the GameDeveloper table. Therefore, from the tables above we can see that a developer called “Blizzard” from the USA developed the game, “World of Warcraft”. Therefore, the foreign key logically links the two tables and allows us to 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.

```
SELECT VideoGame.Name, VideoGame.Genre, GameDeveloper.Name,  
GameDeveloper.Country  
FROM VideoGame  
INNER JOIN GameDeveloper  
ON VideoGame.DeveloperID = GameDeveloper.ID;
```

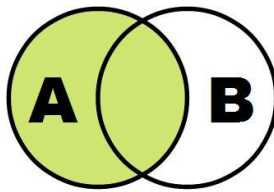
Notice that in the SELECT statement above, we specify the name of the table and the column from which we want to retrieve information, and not just the name of the column as we have done previously. This is because we are getting information from more than just one table, and it is possible that tables 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 data set being returned:

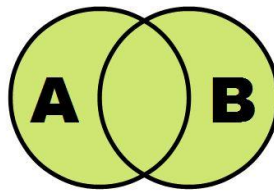
VideoGame.Name	VideoGame.Genre	GameDeveloper.Name	GameDeveloper.Country
Super Mario Bros.	platformer	Nintendo	Japan
World of Warcraft	MMORPG	Blizzard	USA
The Legend of Zelda	adventure	Nintendo	Japan

The INNER JOIN is the simplest and most common type of JOIN. However, there are many other different types of JOINS in SQL, namely:

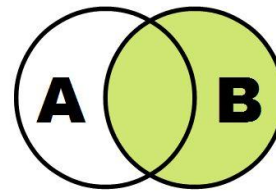
- INNER JOIN: returns records that have matching values in both tables
- LEFT JOIN: returns all records from the left table, and the matched records from the right table
- RIGHT JOIN: returns all records from the right table, and the matched records from the left table
- FULL JOIN: returns all records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



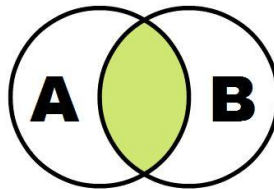
```
SELECT *
FROM A
LEFT JOIN B
ON A.id = B.id
```



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

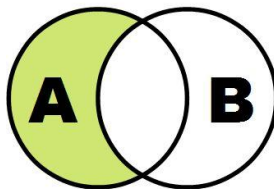


```
SELECT *
FROM A
RIGHT JOIN B
ON A.id = B.id
```

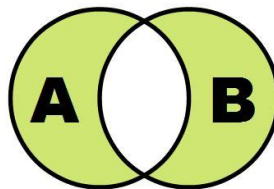


```
SELECT *
FROM A
INNER JOIN B
ON A.id = B.id
```

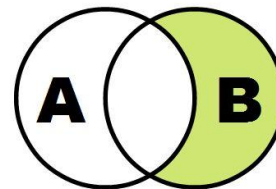
Copyright © 2012 www.mattimattila.fi



```
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 ([Mattila, 2012](#))

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;
```

UPDATING DATA

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

To use the UPDATE statement you:

- Choose the table where the row you want to change is located.
- Set the new value(s) for the wanted 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 rows in a table without deleting the table:

```
DELETE * FROM table_name;
```

DELETING TABLES

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

```
DROP TABLE table_name;
```


For example, 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;
```

INTRODUCTION TO SQLITE

SQLite is built in Python to provide a relational database management system. It is very easy to set up, very fast and lightweight, and thus referred to as 'lite'. Here are some very important features to note about SQLite: self-contained, serverless, with zero-configuration needed to run and transactional.

- **Self-Contained**

This means that it does not need much support from the operating system or external libraries. This makes it suited for use in embedded devices like mobile phones, iPods and game devices that lack the infrastructure of a computer. The source code is found in files called sqlite3.c and the header file sqlite3.h. When you want to use SQLite in an application, ensure that you have these files in your project directory when compiling your code.

- **Serverless**

In most cases, relational database management systems require a separate server to receive and respond to requests sent from the client.

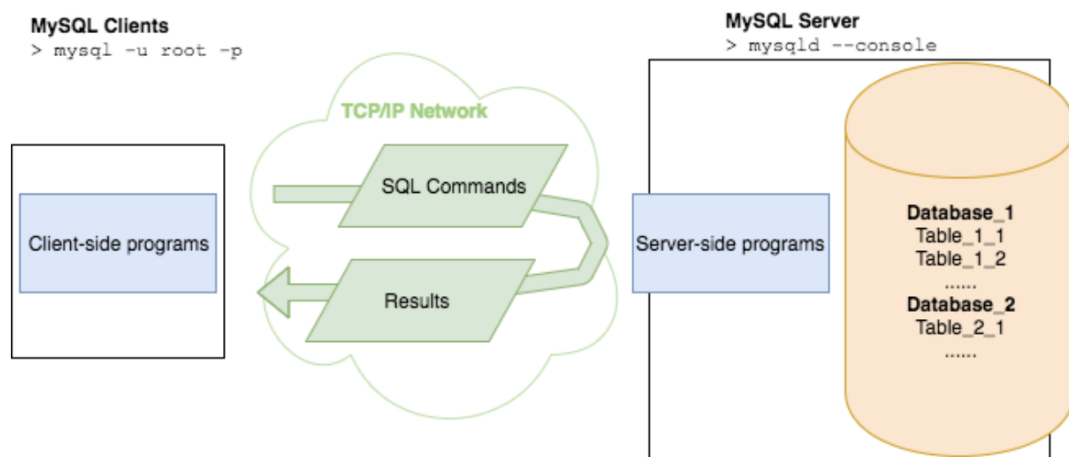


Image source: [SQLite Is Serverless](#)

Such systems include MySQL and 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. On the other hand, SQLite does not make use of a separate server. While using SQLite, the application reads and writes directly to the database files stored on the application server disk.

- **Zero-configuration**

You don't need to install SQLite prior to using it in an application or system. This is because of the serverless characteristic described previously.

- **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.

PYTHON'S SQLITE MODULE

It is easy to create and manipulate databases with Python. To allow us to use SQLite with Python, the Python Standard Library includes a module called "sqlite3". To use the SQLite3 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 open or create it.

```
# Creates or opens a file called student_db with a SQLite3 DB

db = sqlite3.connect('data/student_db')
```

Creating and Deleting Tables

To make any changes to the database, we need a [cursor object](#). A cursor object is an object that is used to execute SQL statements. Next, `.commit` is used 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

cursor.execute('''
    CREATE TABLE student(id INTEGER PRIMARY KEY, name TEXT,
                           grade INTEGER)
''')
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, we use the cursor again to execute a SQL statement. When using data stored in Python variables to insert data into a database table, use the "?" placeholder. It is not secure to use string operations or concatenation to make your queries.

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

```
cursor = db.cursor()
```

```

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 below, the values of the Python variables are passed inside a [tuple](#) (you could also use a dictionary):

```

name3 = 'Sheila'
grade3 = 40

cursor.execute(''INSERT INTO student(name, grade)
               VALUES(:name,:grade)'',
               {'name':name3, 'grade':grade3})

```

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

```

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

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

db.commit()

```

If you need to get the id of the row you just inserted, use **lastrowid**:

```

id = cursor.lastrowid
print('Last row id: %d' % 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))
# The user's phone is not updated

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.

```
id = 3
cursor.execute('''SELECT name, grade FROM student WHERE id=?''', (id,))
student = cursor.fetchone()

print(student)
```

The cursor object works as an iterator, invoking **fetchall()** automatically:

```
cursor.execute('''SELECT name, grade FROM student''')
for row in cursor:
    # row[0] returns the first column in the query (name), row[1] returns
    # email column.
    print('{0} : {1}'.format(row[0], row[1]))
```

Updating and Deleting Data

Updating or deleting data is practically the same as inserting data:

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

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

cursor.execute('''DROP TABLE student''')

db.commit()
```

When we are done working with the DB, we need to close the connection:

```
db.close()
```

SQLite Database Exceptions

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

In the example below, we used 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. Find out more about exceptions [here](#).

```
try:
    # Creates or opens a file called student_db with a SQLite3 DB
    db = sqlite3.connect('student_db')
    # Get a cursor object
    cursor = db.cursor()
    # Check if table users does not exist and create it
    cursor.execute('''CREATE TABLE IF NOT EXISTS
                        java_programming(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](#).

Instructions

First, read **example.py**; open it using your favourite editor.

- **example.py** should help you understand some simple Python.
- You may run `example.py` to see the output. Feel free to write and run your own example code before doing the Task to become more comfortable with Python.



Extra resource

If you'd like to know more about SQL, we highly recommend reading the book, "[Database Design - 2nd Edition](#)" by Adrienne Watt. Chapters 15, 16 and Appendix C of this book provide more detail regarding what has been covered in this task.

Compulsory Task 1

Answer the following questions:

- Go to SQLiteOnline's [SQLite Browser IDE](#). This is where you can write and test your SQL code using their databases. Once you are happy with it, paste your code in a text file named **student.sql** and save it in your task folder.
- Write the SQL code to create a table called Student. The table structure is summarised in the table below (Note that STU_NUM is 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 in question 1, write the SQL code to enter the first two rows of the table as below:

STU_NUM	STU_SNAME	STU_FNAME	STU_INITIAL	STU_STARTDATE	COURSE_CODE	PROJ_NUM
01	Snow	John	E	05-Apr-14	201	6
02	Stark	Arya	C	12-Jul-17	305	11

- Assuming all the data in the Student table has been entered as shown below, write the SQL code that will list all attributes for a COURSE_CODE of 305.

STU_NUM	STU_SNAME	STU_FNAME	STU_INITIAL	STU_STARTDATE	COURSE_CODE	PROJ_NUM
01	Snow	Jon	E	05-Apr-14	201	6
02	Stark	Arya	C	12-Jul-17	305	11
03	Lannister	Jamie	C	05-Sep-12	101	2
04	Lannister	Cercei	J	05-Sep-12	101	2
05	Greyjoy	Theon	I	9-Dec-15	402	14
06	Tyrell	Margaery	Y	12-Jul-17	305	10
07	Baratheon	Tommen	R	13-Jun-19	201	5

- 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 that will 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 all of the data inside a table, but not the table itself.
- Write the SQL code that will delete the Student table entirely.

Compulsory 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 people with an id below than 55.

If you are having any difficulties, please feel free to contact our specialist team [on Discord](#) for support.

Completed the task(s)?

Ask an expert to review your work!

[Review work](#)



Rate us

Share your thoughts

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

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

[Click here](#) to share your thoughts anonymously.

.

REFERENCES

Patel, J. (2012). PHP and MySQL: Practice It Learn It. Jitendra Patel.

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

W3Schools.com. (n.d.). SQL Joins. Retrieved April 15, 2019, from W3Schools.com: https://www.w3schools.com/sql/sql_join.asp