# Study Unit 6 Basic SQL in Python

## Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the operations on tables in databases
2. Design Python programmes for data retrieval

## Overview

In this unit, we will discuss how to use Python to execute SQL commands for database management. We will first introduce SQLite3, a standard Python package that allows programmers to use the Python environment to send SQL statements to the database. We will also discuss how we can generate SQL statements by Python programs for data query and to present their outputs as a pandas DataFrame. Eventually, we will also use SQLite3 to save the changes of the database to a physical file on the computer.

## Chapter 1 Introduction to SQL and SQLite3

Lesson Recording - Introduction to SQL

### 1.1 Introduction to SQL

SQL (Structured Query Language) is the most common and popular programming language designed for database management. There have been various versions of SQL including procedural extensions released such as PSQL, T-SQL, SQL/PSM, etc. Due to the large amount of internal and external data available nowadays, managing, maintaining, and updating database have become compulsory for many organisations. And the demand of SQL specialists has been increasing tremendously.

Many analytics software have integrated SQL as part of their tools. In Python, the package sqlite3 provides the possibility to embed SQL codes into Python programs to facilitate connections to databases and query the data in it. Since sqlite3 belongs to the built-in packages of Python, no installation using pip is required.

Same as scikit-learn, the analytics package introduced in the previous study unit, sqlite3 works hand-in-hand with the pandas packages since both of them are designed for data management. As a result, we can convert output table of SQL queries to pandas DataFrames and vice versa anytime.

|  |
| --- |
| **Example (Students’ score, cont’d):** In this study unit, we will return to our students’ examination score example from Study Units 1 & 2 and manage a database with a table containing the personal information of the students and another table in which their scores of two examinations are stored. Before we can create a database and the tables in it, we need to import the packages sqlite3 and pandas into our Python program first.    Figure 6.1 Importing sqlite3 and pandas  Unlike other packages, the commands in sqlite3 must be executed through a cursor object after the connection between Python and the database has been initiated. As a result, we do not necessarily need an alias to abbreviate sqlite3 for more convenience when writing our program. |

**Read**

Refer to the link below for more details on the sqlite3 package of Python:

<https://docs.python.org/3/library/sqlite3.html>

### 1.2 Creating Data Files in Python

Before creating a database and its tables by SQL respectively, we need to know how data entered by users at runtime of a Python program can be stored in, e.g., comma separated .csv text files. The advantage of storing data in an external file in comparison to a pandas DataFrame is that almost every software such as Excel, SPSS, SAS, etc. has a module to convert text files into its own data file format. In other words, text files are highly compatible and therefore are a good medium of data storage. Furthermore, their file size is usually comparatively small since they only contain the data and the delimiters, and no other information such as cell formatting. As a result, when exchange of data is required within an organisation or between data providers and clients, they are the most suitable format since they would not use up as much upload and download volume or time as other data file formats.

In Study Unit 1, we developed a program to let the users enter the examination scores of a course. However, the program only allows data entry but provides no mechanism to save the data in an external file. In other words, once we quit the program, all the data we have entered will not be accessible or found anymore. Here, we will introduce the Python commands to write data into a .csv file as well as to read the data from it.

In Python, we need to open an external text file with the open() function first before we can write data into it or read data from it.

|  |
| --- |
| with open("file\_name", mode = "r/x/w/a/+") as file\_object:  instructions |

The with statement is usually used in combination with the open() function. The data stored in the text file called file\_name will then be stored in the file\_object for further process. With the parameter mode, we can choose the operations that we are permitted to carry out with the file. Here is a list of some of the available modes:

Table 6.1 List of Some Available Modes of the open() Function

|  |  |
| --- | --- |
| **Character** | **Description** |
| "r" | open for reading (default) |
| "x" | open for exclusive creation, failing if the file already exists |
| "w" | open for writing, truncating the file first |
| "a" | open for writing, appending to the end of the file if it exists |
| "+" | open for updating without truncation (reading and writing) |

(Source: <https://docs.python.org/3/library/functions.html#open>)

Once a file is opened in a particular mode, we are only allowed to execute the permitted operations until it is reopened. For instance, if we have opened a file in reading mode, Python will block us from writing, appending, or updating any content of the file and return an error message to us if our code intends to do so. The difference between both the writing modes "w" and "a" is that "w" overwrites the entire original content in the file by our new entries while "a" appends the new entries to the end of the file while keeping the original content.

Note that it is allowed to combine the "r" or "a" modes with the "+" mode. In both cases, we give the permission for the file to be read and updated. The main difference here is that if we open the file in the "r+" mode, Python will be able to read the entire existing content from the beginning of the file and write the new entries at the end of the file. But if the file is opened in the "a+" mode, Python will not be able to read the entire existing content and just append the updates at the end of the file. Furthermore, if the text file does not exist, it will be created in the "a+" mode, while Python will return an error in the "r+" mode.

After the user has finished entering the data of one record, Python should write this record to the text file with the following syntax.

|  |
| --- |
| file\_object.write(data\_row) |

The name of file\_object must be identical with the one defined in the with open() statement. The object data\_row is a string in which the entered data are stored. After all the data have been saved to the file\_object (Python will transfer the data to the real text file in the background), we need to close the file properly to release its access to other parties.

|  |
| --- |
| file\_object.close() |

Though closing the external files may not affect the program flow directly, it is still important and a good programming habit to do so at the end of the code.

|  |
| --- |
| **Example (Cont’d):** The following program creates an interface to let the users enter the personal information of the students, including their first and last names, gender, birthday, nationality, and study programme. The input mechanism is the same as those introduced in Study Unit 1. For simplicity, we have omitted some of the control mechanisms to prevent invalid inputs here, which are actually essential and extremely important for a program to be executed correctly. After entering the data of one student, Python will convert the entries to a comma separated string and store it as a new line into the file “Student\_DB.csv” for later use.    Figure 6.2 Program to Input Data into a .csv File  In the first line, we store the file name in a variable called target\_file in case we will need it multiple times in the later part of our program. By choosing the "r+" mode in the with open command in the second line, we instruct Python to open the file for reading and writing, and the existing contents of target\_file should be stored in the object called write\_to\_file. The option of reading the stored data in the file would be particularly useful if we needed the number of existing records. This statistic is particularly helpful for two tasks: i) if the file is empty, we will need to add a line of column headers to the record which we also include in lines 4 to 6, ii) we can use the current number of records as the ID number of a new student. We would not have been able to execute these tasks if we had opened target\_file in the "a+" mode instead.  Since .csv text files are comma separated, we need to insert a comma between each column when concatenating the data to a string before storing it to target\_file. It is also important to put the escape sequence \n at the end of the string so that the next record will be stored in a new line. Figure 6.3 shows how the interface looks like when we run our program in JupyterLab.    Figure 6.3 Interface to Input Data in JupyterLab |

To read the existing entries of a text file, we can use a for-loop to go through them line by line and print the content to the screen.

|  |
| --- |
| for line in file\_object:  print(line) |

Once again, the file\_object must be identical with the one defined in the with open() statement. Note that Python is able to separate the records in the text file by recognising the line break (or escape sequence \n) at the end of every line. As a result, the for-loop automatically will run through all the lines and store the content in the variable line, which will then be printed to the screen by the print() function.

|  |
| --- |
| **Example (Cont’d):** Figure 6.4 shows our program to print the records stored in “Student\_DB.csv” to the screen line by line.    Figure 6.4 Print the Records in a .csv File to the Screen Line by Line  Same as the input program, we store the file name in a variable called source\_file in case we will still need it in the later part of our program. Subsequently, we open the file in the "r" mode and store the data in the object read\_from\_file. The advantage of opening the file in reading mode is that the data stored in the .csv file is safe from accidental update or removal. The data will then be printed to the screen line by line with the subsequent for-loop. |

**Read**

Refer to the link below for more details and examples on the open() function:

<https://docs.python.org/3/library/functions.html#open>

**Read**

Refer to the link below for more details and examples on reading and writing files in Python:

<https://docs.python.org/3/tutorial/inputoutput.html#tut-files>

### 1.3 Importing Data to SQL with SQLite3

A database is an organised collection of data, and the data are stored in tables. The data in each table is a specific set of the records in the database, such as a company’s record of its employees, customers, sales, and suppliers. These tables are usually directly or indirectly connected with each other, but it is not a compulsory requirement for them to be put in the same database. Furthermore, the tables have similar structure as the pandas DataFrames: their columns represent the features of the data, or variables, and the records are stored in rows.

To work with databases in Python, we need to generate a “connection” between Python and the databases first. We can use the connect() function of the sqlite3 package for this purpose.

|  |
| --- |
| connection\_object = sqlite3.connect("database\_name") |

If the database already exists, connect() will simply create a connection between the two platforms and let the user gain access to the existing database. On the other hand, if the database is new, connect() will create a new database with the name used in the string "database\_name" and link it with Python directly. Once the connection has been established, we can create SQL syntaxes as strings or string variables in Python, and then send these string objects to SQL for execution. The .cursor() method creates a cursor object to take over this task.

|  |
| --- |
| cursor\_object = connection\_object.cursor() |

To create a table with imported data from a .csv file, we need to first read in the file as a pandas database in Python with the pandas.read\_csv() function, which we have introduced in Chapter 1 of Study Unit 4, then send the data object to the database by the .to\_sql() method of the pandas package.

|  |
| --- |
| data\_object = pandas.read\_csv("csv\_file\_name.csv")  data\_object.to\_sql("table\_name", connection\_object, if\_exists) |

With the parameter if\_exists, we can replace ("replace"), append ("append"), or let Python create an error message ("fail") if the table already exists in the database.

|  |
| --- |
| **Example (Cont’d):** In the first step, we establish a Python connection called conn to the database named "StudentsDB.db" and create a cursor object cur for later use.    Figure 6.5 Establish a Connection between Python and a Database  In the next step, we import the student data created with the program in Figure 6.2 as a pandas DataFrame called students.    Figure 6.6 Import a.csv file Dataset as a pandas DataFrame  Subsequently, we can create a new table in the database called students from this pandas DataFrame. Since we have read in the entire dataset from the .csv file, we can ask Python to replace the existing table if it exists.    Figure 6.7 Export pandas DataFrame to Database by SQL  The parameter index instructs Python to write the row index as a column with column name index\_label in the table. Since the default value here is True, we need to specify it in the .to\_sql() method if we do not wish to include this column. |

With the cursor object being created, we can execute the SQL commands by sending them as strings through the cursor object with the .execute() method.

|  |
| --- |
| cursor\_object.execute("SQL\_command\_string") |

Note that SQL is a separated programming language for database management and its commands are therefore not the same as those in Python. Furthermore, SQL commands are not case sensitive and should end with a semi-colon (this is usually optional, but sometimes the semi-colons are useful to separate the commands that are sent to SQL for execution at the same time).

To select a table from the database, we can send a SELECT statement to SQL.

|  |
| --- |
| SELECT \* FROM table\_name; |

The asterisk (\*) in the SELECT statement is to instruct SQL to take all columns from the table. Once the query has been carried out, we can print one record of the result to the screen by the .fetchone() method.

|  |
| --- |
| cursor\_object.fetchone() |

If we would like Python to print all records from the query result to the screen, we can use the .fetchall() method instead.

|  |
| --- |
| cursor\_object.fetchall() |

Note that once we have applied the .fetchone() or .fetchall() methods, the data records in the query result are literally fetched and no longer available. In other words, if we re-apply the .fetchone() or .fetchall() methods, we will see either no records or some of them missing. If we wish to select and check out the same table again, we will have to redo the query.

|  |
| --- |
| **Example (Cont’d):** In Figure 6.8, we use the .execute() method to send the SELECT statement to SQL for selecting the table students from the database.    Figure 6.8 Select a Table from the Database  Subsequently, we print out one of the records for checking purpose.    Figure 6.9 Fetch One Record for Printing  In the final step, we fetch all the records in the cursor object cur for printing.    Figure 6.10 Fetch All Records for Printing  From Figure 6.10, we can see that the first record has already been fetched in Figure 6.9 and is therefore not included in the output.  If we re-fetch the records from the query output after applying fetchall() once before, SQL will return an empty object to us.    Figure 6.11 Re-Fetch Records after Applying fetchone() or fetchall() |

**Read**

Refer to the link below for more details and examples on the to\_sql() function of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌to_sql.html>

**Read**

Refer to the link below for more details and examples on the

connect(), .cursor(), .execute(), .fetchone(), and .fetchall() functions and methods of the sqlite3 package:

<https://docs.python.org/3/library/sqlite3.html>

## Chapter 2 Data Query

Lesson Recording - Data Query with SQL

### 2.1 Selecting Table

In Chapter 1, we have already introduced the SELECT statement of the standard SQL in its simplest form for table selection. In the following, we will discuss some further options provided by the SELECT statement to optimise our data query.

The SELECT statement also allows us to select some of the variables from the table instead of all of them. But in some cases, we may not even know the variables that the table contains or how their names are correctly spelt. In this case, we can use the .description attribute to extract the variable names from the last queried table.

|  |
| --- |
| cursor\_object.description |

Note that .description is an attribute and not a method. As a result, there are no brackets and arguments behind it. The returned object is a collection of tuples where the first item of each tuple is the column name, and the last six items are None.

|  |
| --- |
| **Example (Cont’d):** In Figure 6.12, we will use the .description attribute to extract the column names of the table students.    Figure 6.12 Extract Column Names of a Table by the .description Attribute  Unfortunately, the returned object does not have a particularly useful form for further usage. Since we only need the first item of each tuple, we can run a for-loop within a list to extract it.    Figure 6.13 Generate Variable List of a Table  While the for-loop is running through the cur.description object, the current tuple is stored in the object column, from which the item with the index 0 will then be put in the list named cols. |

From the previous Chapter, we learn that we can fetch the records from a table for further processes such as printing. However, the data are stored in tuples and when they are printed, we do not see them as table such as a pandas DataFrame. Furthermore, once they are fetched, our program has no more access to the queried table. As a result, it may be desirable to store the result of the query in a pandas DataFrame. In fact, pandas provides the method .from\_records() for this purpose.

|  |
| --- |
| query\_object = pd.DataFrame.from\_records  (data = cursor\_object.fetchall(), columns) |

The .from\_records() method is actually created to convert structured or n-dimensional record arrays to pandas DataFrames. Nevertheless, it suits our purpose perfectly by passing the resulting object of the fetchall() function as the parameter data to the .from\_records() method. With the columns parameter, we can specify our own column names of the output DataFrame. The default value here is None, and the corresponding column names would be simply the column indices.

|  |
| --- |
| **Example (Cont’d):** We can now start a new data query by selecting the entire table students and then store the result of the query in a pandas DataFrame. The query step is almost identical to the syntax in Figure 6.8. The only difference here is that the .execute() method will not carry out our SQL command directly, but a string variable called sql\_select in which the SELECT statement is stored for future use.    Figure 6.14 Execute an SQL Statement Stored in a String Variable  In the next step, we can generate the data by applying the .fetchall() method on the cursor object and convert it to a pandas DataFrame. We will also specify the list cols which was generated in Figure 6.13 as our column names here.    Figure 6.15 Convert the Result of an SQL Query to a pandas DataFrame  Figure 6.15 shows the table students after being converted to a pandas DataFrame, a format that has become well-known to us from the previous study units. |

**Read**

Refer to the link below for more details and examples on the .description attribute of the sqlite3 package:

<https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.description>

**Read**

Refer to the link below for more details and examples on the .from\_records() function of the pandas package:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.‌from_records.html>

**Read**

Refer to the links below for more details and examples on the SELECT statement of SQL:

<https://www.w3schools.com/sql/sql_select.asp>

<https://www.sqlitetutorial.net/sqlite-select/>

### 2.2 Sorting Data

In SQL, we can add the keyword ORDER BY to the SELECT statement to sort the data of a table by some of its variables in the ascending or descending orders.

|  |
| --- |
| SELECT \* FROM table\_name  ORDER BY var1\_name, var2\_name ASC|DESC; |

Note that if we intend to sort the table by multiple variables, we will need to separate their names by commas. The sequence of the variables in this list also reflects the sorting hierarchy. That is, the data are sorted by the first variable in the list initially, those tied records are then sorted by the second variable, and so on. If the data have to be sorted in the descending order by a particular variable, we must specify the DESC option behind the variable name. Since the default value here is ASC, we can omit this option for a variable if the data should be sorted in its ascending order.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to sort the students by their age and nationality in the table students. Since the table only contains their birthdays, stored in the variable Birthday, we need to sort the records in the descending order so that their age will be ordered naturally.    Figure 6.16 Sort Students by Their Birthdays and Nationalities  The second sorting criterion, Nationality, is put after a comma to separate it from the other sorting criteria. The option ASC can actually be omitted here. From the result, we can see that the students with ID number 9 and ID number 6 share the same birthday. Since student number 9 is from Indonesia, which is alphabetically before Singapore, this record is sorted in front of the student number 6. It is also noteworthy that the birthday of the student number 8 is missing. As a result, it appears at the end of the table after sorting the students in the descending order of their birthdays. |

**Read**

Refer to the links below for more details and examples on the ORDER BY option of SQL:

<https://www.w3schools.com/sql/sql_orderby.asp>

<https://www.sqlitetutorial.net/sqlite-order-by/>

### 2.3 Filtering Data

The main purpose of data query is to request the records of interest from the available tables. And more often, it is not the entire table that we are actually looking for, instead we would like to have records that fulfil certain criteria. In SQL, we can use the WHERE clause in the SELECT statement to filter the useful records for us.

|  |
| --- |
| SELECT \* FROM table\_name  WHERE var\_name = value; |

In the above syntax, the selection criterion is presented in its simplest form: records will only be selected if one of the variables is equal to a certain value. We can also construct other criteria by using the operators listed in the following table.

Table 6.2 Operators in the WHERE Clause

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| <> | Not equal (Note: In some SQL versions it may be written as !=) |
| BETWEEN | Between a certain range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

(Source: <https://www.w3schools.com/sql/sql_where.asp>)

Same as the if-command in Python, we can also link multiple criteria in one statement using the AND, OR and NOT operators.

Sometimes, we would rather not obtain records that contain missing values in one or more variables from a query. In this case, we need to add the IS NOT NULL syntax to the WHERE clause.

|  |
| --- |
| SELECT \* FROM table\_name  WHERE var\_name IS NOT NULL; |

If the statement were written without the NOT operator, SQL would return all records with missing values in the variable var\_name to us.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to select students of the analytics programme from the table students. The query can simply be carried out by adding the criterion Program = 'Analytics' to the WHERE syntax.    Figure 6.17 Select only Analytics Students from the Table students  If the value is a string such as “Analytics” in the above syntax, we will need to put it in a pair of quotation marks. And it is important here to pay attention to when and where single or double quotation marks should be used.  Suppose we would like to narrow down our query to only analytics students with ID numbers between 5 and 10.    Figure 6.18 Select only Analytics Students with ID number between 5 And 10  Suppose these are not the students that we are actually looking for, and we would like to select the other analytics students. All we need to modify in the above syntax is to change the operator from BETWEEN to NOT BETWEEN for the ID variable.    Figure 6.19 Select only Analytics Students with ID number not between 5 And 10  In the next query, we would like to select all analytics students who are not from Singapore or China. With the IN operator, we can specify the two values “Singapore” and “China” that we are searching for in the column Nationality. Note that these values must be put in a pair of round brackets. Finally, the NOT operator should be added to the syntax IN ('Singapore', 'China') to negate it.    Figure 6.20 Select Analytics Students not from Singapore or China  Now, all analytics students whose first names start with an “M” should be selected. Here, we can use the LIKE operator and the value is 'M%'. The (%) sign is a wildcard that represents zero, one, or multiple characters. In other words, our value is an M followed by a string of arbitrary length.    Figure 6.21 Select Analytics Students with First Names that Start with “M”  Instead of searching for first names starting with a particular letter, we can also select records with first names that contain a pre-defined string, which is “ar” in the following example.    Figure 6.22 Select Analytics Students with First Names that Contain “ar”  Same as our code in Figure 6.22, we can use the (%) wildcard within the string to specify the position of our pre-defined string. Unsurprisingly, we can also select first names that end with a certain character, which is “s” here.    Figure 6.23 Select Analytics Students with First Names that End with “s”  If we wish to select first names that start and end with some pre-defined characters, we can put the wildcard (%) between the start and the end characters. In the following example, we would like to select students with first names that start with an “M” and end with an “l”.    Figure 6.24 Select Analytics Students with First Names that Start with “M” and End with “l”  The other type of wildcard for the LIKE operator is the underscore sign (\_), which represents a single character. In the following example, we would like to select all students whose last names start with “Ta” and followed by exactly one character.    Figure 6.25 Select Analytics Students with Last Names that Have the Pattern “Ta\_”  The syntax in Figure 6.25 will result in selecting all last names with three characters that start with “Ta”. In other words, last names such as “Tang” will not be included.  In Figure 6.26, cases with missing value in Birthday will be selected. This step is usually helpful to let data analysts study the cases with missing values first and then decide to remove them from the dataset or not.    Figure 6.26 Select Students with No Birthday Record  If we simply want to select all students whose birthday records are not missing, we will just need to replace IS NULL by IS NOT NULL in the SELECT statement.    Figure 6.27 Select Students with Non-Missing Birthday Records |

In SQL, we can also select particular columns from a table in the data query. The asterisk (\*) in the SELECT statement should be replaced by a list of selected variables in this case.

|  |
| --- |
| SELECT var\_name1, var\_name2, …  FROM table\_name WHERE criteria; |

It is also possible to use Python programming to manipulate the SELECT statement as string before sending it to SQL. That is, we can create our own variable list as string in our Python program first and then combine it with the rest of the statement. This approach will give us the flexibility to generate different variable lists for data query depending on the requirement of the situation.

|  |
| --- |
| **Example (Cont’d):** In Figure 6.28, we select the columns ID, LastName, FirstName, Nationality and Program from the table students. At the same time, we are only interested in students outside the analytics program. However, these records should also be sorted by their nationalities.    Figure 6.28 Select Data of Non-Analytics Students from Certain Columns of a Table  The parameter columns in the from\_records() function must be replaced by a new variable list since the list cols which we have generated in Figure 6.13 contains more column names than the table we query here. As a result, it is more practical to store the names of the selected variable in a list or a tuple first, and then concatenate them to a single string by the join() method.    Figure 6.29 Concatenate Items of a Tuple to a String  In the first line, we define the tuple named sel\_cols with the names of the selected variables in it. In the second line, the .join() method is applied to ", ", the separator between the variable names in the output string. Basically, the .join() method runs through all the items in sel\_cols and adds them with ", " (except for the last item for which the separator is not necessary) to the string sel\_cols\_str one after another.  In the next step, we first split the SELECT statement in Figure 6.28 into three parts: "Select ", the variable list and the rest. The original variable list can then be replaced by the string variable sel\_cols\_str. Subsequently, we can concatenate these three partitions of the string into one by “adding” them together as shown in Figure 6.30. Furthermore, we can also use the variable list sel\_cols as the value for the parameter columns in the from\_records() function.    Figure 6.30 Select Certain Columns of a Table by a Variable List String  Hereafter, we can obtain different columns of the table by just changing the variable names in sel\_cols\_str and re-run the code in Figure 6.30. Note that the same technique can also be applied to the other parts of the SELECT statement. |

**Read**

Refer to the links below for more details and examples on the WHERE clause of SQL:

<https://www.w3schools.com/sql/sql_where.asp>

<https://www.sqlitetutorial.net/sqlite-where/>

**Read**

Refer to the link below for more details and examples on the AND, OR, NOT operators of SQL:

<https://www.w3schools.com/sql/sql_and_or.asp>

**Read**

Refer to the links below for more details and examples on the IN operator of SQL:

<https://www.w3schools.com/sql/sql_in.asp>

https://www.sqlitetutorial.net/sqlite-in/

**Read**

Refer to the links below for more details and examples on the BETWEEN operator of SQL:

<https://www.w3schools.com/sql/sql_between.asp>

<https://www.sqlitetutorial.net/sqlite-between/>

**Read**

Refer to the links below for more details and examples on the LIKE operator of SQL:

<https://www.w3schools.com/sql/sql_like.asp>

https://www.sqlitetutorial.net/sqlite-like/

**Read**

Refer to the link below for more details and examples on the wildcard characters in SQL:

https://www.w3schools.com/sql/sql\_wildcards.asp

**Read**

Refer to the links below for more details and examples on NULL values in SQL:

<https://www.w3schools.com/sql/sql_null_values.asp>

https://www.sqlitetutorial.net/sqlite-is-null/

**Read**

Refer to the link below for more details and examples on the .join() methods:

<https://docs.python.org/3/library/stdtypes.html#str.join>

## Chapter 3 Joining Tables

Lesson Recording - Join Tables with SQL

The tables in a database are usually connected in some ways. For instance, in the database of a bank, there may be a table with the personal records of all the customer relationship managers and another table with the records of all customers, including their transaction records and the names of their relationship manager. With these data, the bank can query on the sales records of each manager within a certain period. In this case, we are joining tables from a database to gain cross-table information.

### 3.1 Inner Join

In SQL, there are many ways to join two or more tables: INNER JOIN, LEFT JOIN, CROSS JOIN, etc. Depending on the structure of the tables, these join techniques usually result in different output tables. We will discuss the inner join method in this section.

|  |
| --- |
| SELECT \* FROM table1\_name  INNER JOIN table2\_name  ON table1\_name.match\_var = table2\_name.match\_var; |

The INNER JOIN clause is used within the SELECT statement. It selects only records of table1 that can be matched by records in table2. The rows of table1 or table2 for which SQL cannot find any matches in the opposite table will be dropped from the query. SQL compares the values of each one matching variable from the two tables specified by the user. Usually, these variables should represent the same feature in both tables such as the employee number or customer ID. Note that it is also possible to extend the match to multiple pairs of variables if necessary.

The matching condition should be provided after the ON keyword. To indicate the original table of the matching variables, the name of the table must be specified before each matching variable and separated by a dot (.). With the ON keyword, the matching variables do not need to have the same name in their original tables. But if they do, we can shorten the syntax above by the USING keyword.

|  |
| --- |
| SELECT \* FROM table1\_name  INNER JOIN table2\_name USING(match\_var); |

The name of the original table does not need to be mentioned in the bracket of the USING keyword. If, however, the column name only exists in one of the tables, SQL will return an error message to us. Another difference between the ON and USING keywords is that both matching variables will be included in the output table if we use the ON keyword, whereas only the matching variable of table1 will remain if the USING keyword is used for matching.

If more than one records in table2 are found matching to a record in table1, SQL will append each of them to a copy of the matched record of table1. This is the so-called 1:n matching.

|  |
| --- |
| **Example (Cont’d):** In the following, we will create a new table called grades which contains the scores of the students in the two previous examinations. The table will then be joined with the table students. In Figure 6.31, the data stored in a .csv file will be transferred to grades by the .to\_sql() function.    Figure 6.31 Create a New Table Called grades in the Database  In Figure 6.32, we query the entire table grades from the database and convert it to a pandas Database for printing and controlling purposes.    Figure 6.32 Contents of the Table grades  Subsequently, we join the two tables, students and grades, with the INNER JOIN clause. In our first attempt, we use the ON keyword to match the student ID of the two tables.    Figure 6.33 Inner Join the Tables students and grades by the ON Keyword  We can see from Figure 6.33 that most of the students have two rows in the output table. Since the majority of them have taken part in both examinations, they should have indeed two records each in the table grades. With the INNER JOIN clause, SQL generates one copy of each student’s record in students and appends the scores and course codes found in grades to the records of the corresponding student.  Furthermore, we can also see from Figure 6.33 that there are two columns named ID. As mentioned above, the matching variables of both tables will be kept in the output table. If we intended to use this table for further process, the ambiguous variable name could cause troubles in data query. It would therefore be important to either rename or drop one of the ID columns from the table.  Since the matching variable in both tables share the same name, ID, we can also apply the USING keyword instead.    Figure 6.34 Inner Join the Tables students and grades by the USING Keyword |

**Read**

Refer to the links below for more details and examples on the INNER JOIN clause of SQL:

<https://www.w3schools.com/sql/sql_join_inner.asp>

<https://www.sqlitetutorial.net/sqlite-inner-join/>

### 3.2 Left Join

Another way to join two or more tables of a database is the left join method.

|  |
| --- |
| SELECT \* FROM table1\_name alias1  LEFT JOIN table2\_name alias2  ON alias1.match\_var = alias2.match\_var; |

We add the new options alias1 and alias2 to this syntax. These aliases are usually abbreviated references of table1 and table2 and can be useful when there are two variables with the same name in both tables. Note that the aliases are not limited to the LEFT JOIN clause, but applicable to any SELECT statement.

We can also use USING instead of ON for the matching condition.

|  |
| --- |
| SELECT \* FROM table1\_name  LEFT JOIN table2\_name  USING(match\_var); |

In left join, SQL searches for records from table2 that match the records from table1 based on the matching condition. If no records in table2 can be found for a record from table1, that record of table1 will still be kept in the output table. The values of the variables originated from table2 will be missing values in this case.

|  |
| --- |
| **Example (Cont’d):** The tables students and grades will be merged by the LEFT JOIN method in the following.    Figure 6.35 Left Join the Tables students and grades  In Figure 6.35, we can see that the student with ID number 7 has missing data in the variables Grade and Course, both variables originated from the table grades. We can therefore conclude that this student has not taken part in both the examinations of Course101 and Course102.  We can also sort the joined table by some selected variables. All we need to do is to add the ORDER BY keyword to the SELECT statement as shown in Figure 6.36.    Figure 6.36 Sort the Left Joined Table by the Course Code and the Students’ Last Name  In Figure 6.36, we have sorted the output table by the course code and the last name of the students, both in ascending order. By sorting the data this way, we can see the examination scores of one course as consecutive records, and students from the same course are sorted in the same order as most probably the official student list, namely by their last names in alphabetical order.  In Figure 6.37, all records from the joined table that have missing values in the variable Grade are selected.    Figure 6.37 Select Records with Missing Data in grade  We add the aliases s for the table students and g for the table grades to the SELECT statement. They are included in the matching criterion since the ON keyword is used here for record matching. Furthermore, we just want the joined table to contain the columns ID, LastName, FirstName and Course. Since there is an ID column in both tables and only the one from the table students should be carried over to the output table, we need to specify its original table with the alias s. |

**Read**

Refer to the links below for more details and examples on the LEFT JOIN clause of SQL:

<https://www.w3schools.com/sql/sql_join_left.asp>

<https://www.sqlitetutorial.net/sqlite-left-join/>

### 3.3 Cross Join

With the cross join method, SQL produces the cartesian product of the involved tables.

|  |
| --- |
| SELECT \* FROM table1\_name  CROSS JOIN table2\_name; |

The cartesian product usually refers to the collection of all cross-item combinations resulting from two arrays. In terms of the cross join method, it means that every record of table1 is merged with *all* records of table2. In other words, if table1 and table2 have m and n records, respectively, there will be a total of m×n records in the output table. Since no matches are required here, the ON and USING keywords can be omitted in the SELECT statement.

|  |
| --- |
| **Example (Cont’d):** Now we cross join the tables students and grades.    Figure 6.38 Cross Join the Tables students and grades  Every row of the table students is now cross-combined with all the rows of the table grades. In other words, the output table contains records in which students are assigned to grades that they did not score themselves. It is obvious that this data query does not make much sense in terms of its logical structure and analytical value. Nevertheless, cross join could be useful if we had another table with data that are equal to every student. We could then merge the same records to every row of the table students. |

**Read**

Refer to the link below for more details and examples on the CROSS JOIN clause of SQL:

<https://www.sqlitetutorial.net/sqlite-cross-join/>

### 3.4 Outer Join

With the outer join method, or full outer join, SQL produces the union of the involved tables. In other words, not only records from both tables that can be matched by the matching criterion will be selected, records from either one table that cannot find any match from the opposite side will also be carried over in the output table. However, their values in the variables originated from the other tables will be None.

One difficulty of applying the outer join method in Python is the fact that it is simply not supported by the sqlite3 package, although the OUTER JOIN clause is actually available in other SQL versions. Nevertheless, we can combine the LEFT JOIN clause with the UNION ALL operator to create the same result as the OUTER JOIN clause.

|  |
| --- |
| SELECT var\_list FROM table1\_name alias1  LEFT JOIN table2\_name alias2 USING(matching\_var) UNION ALL  SELECT var\_list FROM table2\_name alias2  LEFT JOIN table1\_name alias1 USING(matching\_var) WHERE alias1.var\_name IS NULL; |

In the first SELECT statement, the left join method is applied and the records from table2 are matched with the records from table1. As discussed in Chapter 3.2, all records from table1 are selected in the output table here regardless the matching results. In the second SELECT statement, the left join method is applied to table1 and table2 in the opposite roles. As a result, all records from table2 are selected here. However, we need to drop those matches that are already included in the first SELECT statement to prevent duplicates. Logically, these records must contain data from both tables, and those from table2 with no matching records must have missing data in the columns of table1. As a result, we can simply use this result as our selection criterion. Both SELECT statements are connected by the UNION ALL operator to produce a combined table of the two queries.

Since the sequence of the columns are naturally different in the output tables of the two queries, the UNION ALL operator would simply append the data of the second query to those of the first query regardless their original columns if we just used the asterisk (\*) in both SELECT statements. To avoid such mess in the output data, we must specify the same list of variable names in both queries so that the sequence of the columns are identical.

|  |
| --- |
| **Example (Cont’d):** Before we use outer join to merge the tables students and grades, we need to create a list with the variable names of both tables to ensure that the same sequence of columns are produced in the output table of both queries.  In the column list, we must specify the original table of each variable by the aliases so that every column in the output table is uniquely defined. The easiest way to generate such a list is to define a string with all the variable names written in it. However, if the involved tables have many columns, we will need to write a very long string in our program which is rather not efficient. Figure 6.39 shows a Python program which generates a variable list from the column names of both tables together with the corresponding aliases.    Figure 6.39 Generate Variable List from Both Tables with Aliases  In the first two lines, the column names of the tables students and grades are stored in the lists students\_cols and grades\_cols by the same technique as shown in Figure 6.13. In the third and fourth line, we remove both ID variables from the lists since they will be merged into one column eventually and therefore do not need aliases. In the fifth line, we first add the aliases "s." and "g." as strings to each item of the respective variable list by running through them in a for-loop. Subsequently, the two lists are concatenated, and the item "ID" is added back to the front of the resulting list which is now called cols\_list. In line six, cols\_list is converted to a string with commas separating the column names. As shown in Figure 6.39, an alias is added to all names except for the ID variable, which has now become the first column of the output tables in both SELECT statements.    Figure 6.40 Outer Join the Tables students and grades (1)    Figure 6.41 Outer Join the Tables students and grades (2)  From Figure 6.40, we can see that the student number 7, who has no exam records found in the table grades, is included in the output table. This is not surprising since it is the same result as the left join method. In Figure 6.41, the students with ID numbers 21 to 24 only seem to have some exam results but no personal records found in the table students. Basically, such cases can only be detected if we left join the table grades with the table students, which is exactly our second SELECT statement in Figure 6.40. |

**Read**

Refer to the link below for more details and examples on the UNION ALL operator of SQL:

<https://www.sqlitetutorial.net/sqlite-union/>

**Read**

Refer to the link below for more details and examples on the .remove() method:

<https://www.w3schools.com/python/ref_list_remove.asp>

## Chapter 4 Grouping Data

Lesson Recording - Group Data with SQL

### 4.1 Combining Records into Groups

In SQL, we can combine records of a table into groups based on one or more categorical variables. In most cases, the grouping is combined with the calculation of some statistics for each group by the aggregate functions.

Table 6.3 List of Aggregate Functions in SQL

|  |  |
| --- | --- |
| **Aggregate Functions** | **Description** |
| AVG | Average of the specified columns in a group |
| COUNT | Number of rows in a group |
| MAX | Maximum value of the specified columns in a group |
| MIN | Minimum value of the specified columns in a group |
| STDDEV | Standard deviation of the specified columns in a group |
| SUM | Sum of the specified columns in a group |
| VARIANCE | Variance of the specified columns in a group |

To group records of a table together, we need to add the GROUP BY statement to the SELECT statement.

|  |
| --- |
| SELECT var\_list AGGREGATE\_FUNCTION(var\_name)  FROM table\_name GROUP BY groupvar1\_name, groupvar2\_name, …; |

In the variable list of the SELECT statement, we can also specify the aggregate function and the variable for which the aggregated statistics of each group should be calculated. The GROUP BY statement is followed by the variable names based on which the groups should be formed, and the variable names must be separated by commas here. If the table is grouped by more than one variables, the groups will be formed by the cartesian products of the categories in the variables.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to group the students by their study programmes and the number of students in each programme should be counted.    Figure 6.42 Count the Number of Students in Each Programme  In the above syntax, we only select the column program for our output tables since we are only interested in the number of students in each of its categories. The count of students is added as a variable to the query, and the name of it is simply taken over from the SELECT statement, namely Count(Program).  We can also group and count our students by their nationalities and the result will be sorted by the counts in the descending order.    Figure 6.43 Sort the Nationalities of the Students by Their Counts  In Figure 6.43, we add the name Count to the variable list in the SELECT statement which will be used as the variable name for Count(Nationality). In other words, we can specify a name for the column of the aggregated statistics by placing it behind the aggregate function in the SELECT statement.  Next, we will group the students by their study programmes and nationalities. The frequency of each nationality in each study programme should be counted as well.    Figure 6.44 Group the Students by their Study Programmes and Nationalities  The grouping variables, separated by a comma, are listed in the GROUP BY statement. To count the frequency of the cartesian product of nationality and program, we need to put a multiplication operator (\*) between the two variables inside the COUNT function. From Figure 6.44, we can see that there are altogether eleven analytics students who are from Singapore, and two students of the same programme actually come from Malaysia, and so on. |

**Read**

Refer to the links below for more details and examples on the GROUP BY operator of SQL:

<https://www.w3schools.com/sql/sql_groupby.asp>

<https://www.sqlitetutorial.net/sqlite-group-by/>

**Read**

Refer to the link below for more details and examples on the MIN()and MAX() functions of SQL:

<https://www.w3schools.com/sql/sql_min_max.asp>

Refer to the link below for more details and examples on the COUNT(), AVG(), and SUM() functions of SQL:

<https://www.w3schools.com/sql/sql_count_avg_sum.asp>

### 4.2 Filtering Groups

As demonstrated in the previous section, we can compute aggregated statistics after grouping the data. Furthermore, we can also filter the groups by some specified conditions. The filtering process for grouped data is carried out by the HAVING clause.

|  |
| --- |
| SELECT var\_list AGGREGATE\_FUNCTION(var\_name)  FROM table\_name GROUP BY groupvar1\_name, groupvar2\_name, … HAVING conditions; |

Nevertheless, we can extend this SELECT statement with a WHERE clause that are used to filter the records before the grouping takes place, or with the ORDER BY keyword to sort the grouped table by the aggregated statistics, or the INNER JOIN/LEFT JOIN/CROSS JOIN clauses to merge columns from other tables before the grouping and the calculation of aggregated statistics are carried out.

|  |
| --- |
| **Example (Cont’d):** Suppose we would like to compare the average grades of the students based on their age. Instead of calculating their age, we can also just use their birthyears which can be extracted from the variable Birthday in the table students by the SQLite function STRFTIME().    Figure 6.45 Calculate Average Grade of Students from Different Birthyears  To retrieve the exam grades, we need to join the tables students and grades here. Since it does not make much sense to include the records with examination grades but no matching personal data as well as students who did not participate in any of the exams, inner join is applied. In the output table, we group the students by their birthyears generated by STRFTIME('%Y', Birthday), which is included in the column list in the SELECT statement. The parameters in the function indicate that the year ('%Y') should be extracted from the dates stored in the variable Birthday. We can also see from Figure 6.45 that once the column name of an aggregated statistic has been established within the SELECT statement, we can use it in the other parts of the syntax such as WHERE, GROUP BY, ORDER BY, etc.  In the next query, the students are grouped by their study programme and their average grades in each course in which they have taken the examination will be determined.    Figure 6.46 Calculate Average Grade of Different Programmes in Different Courses  In the above program, we have excluded those records in the joined table that have missing values in the variable Grade. In other words, these were records from the table students for which matching could be found in the table grades, but their data in the column Grade were missing. We have also counted the number of different IDs in each of the groups to determine the number of students who participated in the corresponding examination. The counts can be found in the new column named NumStudents. As a result, there were 15 analytics students who took part in the examination of Course101 and their average grade is 69.67. Other programmes such as accountancy and business have only got one student each to participate in this examination. Therefore, their low average scores here are unreliable statistics for any inference.  If we would like to select only programmes with 5 students or more participated in the exams, we can use the HAVING clause for filtering the grouped table.    Figure 6.47 Selected Courses with NumStudents >= 5 Grouped by Study Programme  Now we will create a grouped table in which the average grade of each student in the two exams are calculated. The number of exam participations will be counted for each of them and stored as a new variable called NumCourse. Eventually, the output table should be sorted by the students’ average grades in the descending order and contains only students with an average grade of at least 40 marks.    Figure 6.48 Listing Students with AverageGrade >= 40  In the first step, we drop variables that are less relevant to this query such as Birthday, Nationality and Course. Same as our code in Figure 6.47, records with missing data in the variable Grade will be dropped from the table. After grouping the table by the ID of the students and their average grades have been calculated, students will be sorted by their average grades and those with less than 40 marks will not be selected. |

**Read**

Refer to the links below for more details and examples on the HAVING clause in SQL:

<https://www.w3schools.com/sql/sql_having.asp>

<https://www.sqlitetutorial.net/sqlite-having/>

**Read**

Refer to the link below for more details and examples on the STRFTIME() function in SQLite:

<https://sqlite.org/lang_datefunc.html>

**Read**

Refer to the link below for more details and examples on date data types in SQL:

<https://www.w3schools.com/sql/sql_dates.asp>

## Chapter 5 Editing Data

Lesson Recording - Edit Data with SQL

### 5.1 Inserting Records

In the previous chapters, we have been introduced to methods for extracting and reshaping information from a database. Nevertheless, SQL also provides the possibility for us to change the data or even the structure of a table. In this section, we will discuss how to insert new records to a table.

|  |
| --- |
| INSERT INTO table\_name (var\_list)  VALUES (value\_list); |

In the INSERT INTO statement, the variable list added behind the table name should be a subset of the column names in the table. Its length must be identical with the length of the value list, and both lists must be put in parentheses. It is also important to ensure that the sequence of the elements in the value list corresponds to the sequence of the variables so that the values are assigned to the correct column eventually. For inserting multiple records, the value list of each record must be wrapped up by a pair of brackets and every list must be separated by a comma from one another. Furthermore, the values of the variables excluded in the INSERT INTO statement for the new records will be None. If we intent to provide values to all columns, the variable list including the brackets can also be omitted from the syntax.

|  |
| --- |
| **Example (Cont’d):** In Chapter 4, we have often come across the student with ID number 7 who has no matching records found in the table grades. Suppose we have now received his grades for Course101 and Course102, which are 62 and 54, respectively, we can insert these two records into the table.    Figure 6.49 Insert Multiple New Records for a Student into the Table grades  Recall from Figure 6.41 that the student with ID number 21 has got a grade in Course102 but no personal record in the table students. We have now received her personal data and would like to insert it into the table.    Figure 6.50 Insert a New Record into the Table students without Variable List  Different from the program in Figure 6.49, the variable list is omitted in the INSERT INTO statement here. Nevertheless, the behaviour of the program remains unchanged since the data of all variables are completely available and they are put in the right sequence in the syntax. |

**Read**

Refer to the links below for more details and examples on the INSERT INTO clause in SQL:

<https://www.w3schools.com/sql/sql_insert.asp>

<https://www.sqlitetutorial.net/sqlite-insert/>

### 5.2 Updating Records

We can also update or edit the data of existing records by the UPDATE statement.

|  |
| --- |
| UPDATE table\_name  SET var1\_name = value1, var2\_name = value2, … WHERE condition; |

The concept of updating data in the tables by SQL is slightly different from editing the contents of a spreadsheet. Here, we need to state certain conditions in a WHERE clause which must be fulfilled by a record in order to get itself updated. In other words, if the condition is true to more than one records, all of them will be modified simultaneously. Hence, depending on the nature of the update, the condition must be specified precisely so that the update is not applied to the wrong records. If the WHERE clause is omitted, *all* records in the involved table will be updated.

With the keyword SET, we can specify the columns that SQL should update and their new values. The UPDATE statement is particularly useful to replace missing values or outliers in a dataset.

|  |
| --- |
| **Example (Cont’d):** From Figure 6.41, we can see that student with ID number 20 has two records in the table grades. However, her grade in Course102 is missing. Suppose she had to take the supplementary exam due to illness and her grade is therefore only available with some weeks of delay.    Figure 6.51 Update the Value of a Selected Record in the Variable Grade  It requires two conditions here to find the target record for updating: the student ID number must be 20 and the course must be “Course102”. If the ID number were not specified, every student’s grade in Course102 would become 69. On the other hand, if Course were not included as one of the conditions, this student’s grades of Course101 and Course102 would both change to 69. |

**Read**

Refer to the links below for more details and examples on the UPDATE clause in SQL:

<https://www.w3schools.com/sql/sql_update.asp>

<https://www.sqlitetutorial.net/sqlite-update/>

### 5.3 Deleting Records

Deleting records from a table works in a very similar way as updating data in rows. That is, conditions must be set so that records can be selected for removal.

|  |
| --- |
| DELETE FROM table\_name  WHERE condition; |

Same as the UPDATE statement, it is very important to specify the correct records for deletion. If the condition is too vague, there can be more records deleted than originally intended. Note that once a row has been dropped from a table, there is no possibility to undo it in SQL.

|  |
| --- |
| **Example (Cont’d):** After the personal data of student with ID number 21 has been inserted to the table students as shown in Figure 6.50, it is decided that all records of students with ID number 22 to 24 should be deleted from the table grades since no matching personal data could be found for them.    Figure 6.52 Delete Records with ID Number >=22 from the Table grades  The condition for deletion is a rather simple one: ID >= 22. Since it was the records with the highest ID numbers being deleted, we can sort the table by ID in the descending order to check whether the deletion has been carried out properly. |

**Read**

Refer to the links below for more details and examples on the DELETE clause in SQL:

<https://www.w3schools.com/sql/sql_ref_delete.asp>

<https://www.sqlitetutorial.net/sqlite-delete/>

### 5.4 Altering Tables

In the previous sections, we have discussed changing the rows of a table. In this section, we will introduce a method to alter a table by editing its columns. With the ALTER TABLE statement, we can rename a table, rename a column, or add a column. In the first step, we introduce the following syntax to add a column to a table.

|  |
| --- |
| ALTER TABLE table\_name  ADD column\_name; |

Unlike the other SQL versions, SQLite3 only allows adding one column at a time.

We can rename a column with the following version of the ALTER TABLE statement.

|  |
| --- |
| ALTER TABLE table\_name  RENAME old\_column\_name TO new\_column\_name; |

Here, we can only rename one column at a time as well.

|  |
| --- |
| **Example (Cont’d):** We would now like to add a new column called email to the table students to store their email addresses.    Figure 6.53 Add a New Column called email to the Table students  The variable is appended to the rightmost edge of the table with None as its value.    Figure 6.54 The Table students after a New Column email Being Added  Suppose the email address format of this university is “FirstName.LastName @ouruni.edu.sg”. We can use the || operator of SQL to concatenate the values in the columns FirstName and LastName with the string "@ouruni.edu.sg" as the values of the new column email. Moreover, we can also convert all characters of the email address to lower case with the LOWER() function of SQL.    Figure 6.55 Update Values in the Column email  The update here does not require the WHERE clause because all email addresses can be generated with the identical syntax. Certainly, the procedure here is strongly simplified since we do not consider the possibilities of a) students using user-defined email address, b) assigning the same email address to multiple students because their names are identical, c) white spaces in the students’ name if they have middle names, d) special characters in their names such as å, é, ñ, or ß. We need a much more sophisticated program to deal with all these issues, and we will omit it since it is out of the scope of our discussion here.  Finally, since the names of all the other columns start with a capital letter, we will therefore rename the column from email to Email.    Figure 6.56 Rename the Column from email to Email |

Beside altering the content of a table, we can also create a new table in the database.

|  |
| --- |
| CREATE TABLE table\_name (column1\_name, column2\_name, …); |

And it is also possible to drop a table from the database.

|  |
| --- |
| DROP TABLE table\_name; |

Certainly, we also have the possibility to rename a table.

|  |
| --- |
| ALTER TABLE table\_name RENAME TO new\_table\_name; |

The syntax for renaming a table is fairly similar to the one for renaming a column in a table. The only difference is that there should be no name between RENAME and TO when giving a new name to a table.

If we want to copy the data from one table to another one which has the same column structure, we can modify and apply the INSERT INTO statement for this purpose.

|  |
| --- |
| INSERT INTO target\_table\_name SELECT value\_list  FROM source\_table\_name; |

Instead of having a variable list and a value list, we can have the SELECT statement embedded in the above syntax. As a result, data are queried from another table first and then inserted into the target table. We can also use additional clauses such as ORDER BY, WHERE, INNER JOIN, etc., to sort or select specific records, or merge multiple tables before inserting them.

|  |
| --- |
| **Example (Cont’d):** In the following, we will first merge the last and the first names of the students to one new column called Name in the table students. Subsequently, we will duplicate the table and drop the variables LastName and FirstName from the new one and rearrange the sequence of the columns. The reasons we do not carry out the alteration in the existing table students is that we would like to keep the original record just in case we will need it again.    Figure 6.57 Add a New Column Called Name to the Table students  In the next step, we convert the students’ last name and first name to the format "LastName, FirstName" and store it in a new variable called Name.    Figure 6.58 Concatenate Last Name and First Name as a New Variable  Same as the creation of the email address in Figure 6.55, we use the || operator to merge the last name, the separating comma and the first name together.  We can now copy the data to a new table called students2. However, we would like to exclude the columns LastName and FirstName in the new table since they are now completely represented by Name. Furthermore, as we can see from Figure 6.58, all new variables are appended to the rightmost end of the table. Thus, we need to rearrange the columns to our need first before copying them to the new table.    Figure 6.59 Create a New Table named students2  In the first line, we generate a list of variables in the sequence of how they should be inserted in students2. The reason of using a Python string variable to store them here is that we will need to use it later for the INSERT INTO statement again. In the second line, we can then create the new table students2 and use the variable list string sel\_col to define the variables and their sequence in the new table.    Figure 6.60 Query Data from students and Insert Them Into students2  In Figure 6.60, we combine the SELECT and the INSERT INTO statements to transfer the data from students to students2. We insert the variable list string sel\_cols, which has already been used in the CREATE TABLE statement in Figure 6.59, into the SELECT statement. Hence, we have ensured that the sequence and names of the columns are identical in both the programs. In addition, we added the ORDER BY keyword to the SELECT statement so that the data are now sorted by the ID numbers of the students.  Figure 6.61 shows the table students2 after the data have been inserted.    Figure 6.61 Output of students2 after Inserting the Data  Unlike other SQL versions, SQLite does not support DROP COLUMN in the ALTER TABLE statement. Therefore, the method introduced in Figure 6.58 and Figure 6.59, namely, to transfer all data except the variables that should be removed to a new table, is the only way to delete a column. |

**Read**

Refer to the links below for more details and examples on the ALTER TABLE statement in SQL:

<https://www.w3schools.com/sql/sql_alter.asp>

<https://www.sqlitetutorial.net/sqlite-alter-table/>

**Read**

Refer to the link below for more details and examples on the RENAME TO keywords of the ALTER TABLE statement in SQL:

<https://www.sqlitetutorial.net/sqlite-rename-column/>

**Read**

Refer to the link below for more details and examples on the LOWER() function in SQL:

<https://www.w3resource.com/sql/character-functions/lower.php>

**Read**

Refer to the links below for more details and examples on the CREATE TABLE statement in SQL:

<https://www.w3schools.com/sql/sql_create_table.asp>

<https://www.sqlitetutorial.net/sqlite-create-table/>

**Read**

Refer to the links below for more details and examples on the DROP TABLE statement in SQL:

<https://www.w3schools.com/sql/sql_drop_table.asp>

<https://www.sqlitetutorial.net/sqlite-drop-table/>

### 5.5 Committing Changes in Database

It is often important to check on the existing content in the database and clear up tables that are no longer necessary. In SQLite, there is a master table called sqlite\_master which holds the schema of the entire database. We can therefore query the names of the existing tables in a database by a SELECT statement.

|  |
| --- |
| SELECT name FROM sqlite\_master WHERE type = 'table'; |

This syntax is a fixed expression and needs no adjustments. Basically, it queries the names of all the existing tables (obviously stored as values in the column name) from sqlite\_master in the same fashion as we query data from a table. Armed with the result of this query, we can decide on whether taking actions or not on the tables.

So far, all procedures have actually been carried out on the virtual platform. That means, the changes are only stored in the virtual memory of our computer and not saved to the hard disk yet. Therefore, before closing the database, we need to commit all changes through the connection object back to the physical file of our database.

|  |
| --- |
| connection\_object.commit() |

The .commit() method of the sqlite3 package must be applied to the connection object. It works in the same way as the “save” function in most of the software. That means, we can place it in our Python program wherever we think we need to save the changes before they are lost. However, we cannot undo the changes once they are committed.

Finally, we can close the connection to the database by the .close() method.

|  |
| --- |
| connection\_object.close() |

Note that the .close() method does not call the .commit() method automatically. In other words, if we close the connection before committing the changes to the physical file, all the modifications in the database will be lost.

|  |
| --- |
| **Example (Cont’d):** In the first step, we would like to obtain a list of all the existing tables in the database.    Figure 6.62 Check on Existing Tables in the Database  As expected and confirmed by Figure 6.62, students, grades and students2 are the tables that our database contains. Suppose we have now decided not to keep the table students anymore since students2 actually contains the same data, we can apply the DROP TABLE statement to remove it.    Figure 6.63 Dropping Table students from the Database  In Figure 6.63, we have added the SELECT statement to check on the names of the existing tables in the database after dropping students. The output of the query shows us that the table students has indeed disappeared.  Since students no longer exists, we can use students2 to replace student by renaming it accordingly.    Figure 6.64 Renaming students2 to student  By adding the same SELECT statement to check on the names of the existing tables, we can see that students2 has indeed been renamed to student.  We can now use the object conn, which connects our database to the physical file “StudentsDB.db” according to Figure 6.5, to commit and save all the changes in our database. Subsequently, we can also close the connection to the database.    Figure 6.65 Commit Changes of the Database to Physical File and Close Connection |

**Read**

Refer to the link below for more details and examples on the .commit() method in the sqlite3 package:

<https://docs.python.org/3/library/sqlite3.html#connection-objects>

## Summary

In this study unit, we have first learned how to write Python programs to store data entered by a user to a .csv text file. After the sqlite3 package has been introduced, we are able to connect Python with the databases and convert external data sources saved as .csv text files to database tables by SQL. With the SELECT statement, we could execute different types of data query such as sorting and filtering data. Here, we have used Python programming to generate SELECT statements flexibly and to convert the query output to pandas DataFrames for better presentation in the Python environment. We have also come across the four methods for joining two or more tables of a database: inner join, left join, cross join, and outer join. The output of these methods could vary strongly since they select the records differently. The option of grouping the data in a table and calculating some aggregated statistics has also been illustrated. And we have learned that groups can be filtered by the aggregated results. With the ALTER TABLE statements, we can also add, rename, or delete columns and alter thereby the structure of a table. Finally, we need to apply the .commit() method of the sqlite3 package to save the modification of the database to a physical file in our computer.

## References

pandas (n.d.). *pandas.DataFrame.from\_records*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.from_records.html>

pandas (n.d.). *pandas.DataFrame.to\_sql*. The pandas development team. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.‌DataFrame.to_sql.html>

Python.org (n.d.). *Built-in Functions*. Python Software Foundation. <https://docs.python.org/3/library/functions.html#open>

Python.org (n.d.). *Built-in Types*. Python Software Foundation. <https://docs.python.org/3/library/stdtypes.html#str.join>

Python.org (n.d.). *Input and Output*. Python Software Foundation. <https://docs.python.org/3/tutorial/inputoutput.html#tut-files>

Python.org (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation.   
<https://docs.python.org/3/library/sqlite3.html>

Python.org (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation.   
<https://docs.python.org/3/library/sqlite3.html#connection-objects>

Python.org (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation. <https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.description>

Python Tutorial (n.d.). *Python List remove() Method*. w3schools.com. <https://www.w3schools.com/python/ref_list_remove.asp>

SQL Tutorial (n.d.). *SQL ALTER Keyword*. w3schools.com. <https://www.w3schools.com/sql/sql_alter.asp>

SQL Tutorial (n.d.). *SQL AND, OR and NOT Operators*. w3schools.com. <https://www.w3schools.com/sql/sql_and_or.asp>

SQL Tutorial (n.d.). *SQL BETWEEN Operator*. w3schools.com. <https://www.w3schools.com/sql/sql_between.asp>

SQL Tutorial (n.d.). *SQL COUNT(), AVG() and SUM() Functions*. w3schools.com. <https://www.w3schools.com/sql/sql_count_avg_sum.asp>

SQL Tutorial (n.d.). *SQL CREATE TABLE Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_create_table.asp>

SQL Tutorial (n.d.). *SQL DELETE Keyword*. w3schools.com. <https://www.w3schools.com/sql/sql_ref_delete.asp>

SQL Tutorial (n.d.). *SQL DROP TABLE Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_drop_table.asp>

SQL Tutorial (n.d.). *SQL GROUP BY Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_groupby.asp>

SQL Tutorial (n.d.). *SQL HAVING Clause*. w3schools.com. <https://www.w3schools.com/sql/sql_having.asp>

SQL Tutorial (n.d.). *SQL IN Operator*. w3schools.com. <https://www.w3schools.com/sql/sql_in.asp>

SQL Tutorial (n.d.). *SQL INNER JOIN Keyword*. w3schools.com. <https://www.w3schools.com/sql/sql_join_inner.asp>

SQL Tutorial (n.d.). *SQL INSERT INTO Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_insert.asp>

SQL Tutorial (n.d.). *SQL LEFT JOIN Keyword*. w3schools.com. <https://www.w3schools.com/sql/sql_join_left.asp>

SQL Tutorial (n.d.). *SQL LIKE Operator*. w3schools.com. <https://www.w3schools.com/sql/sql_like.asp>

SQL Tutorial (n.d.). *SQL MIN() and MAX() Functions*. w3schools.com. <https://www.w3schools.com/sql/sql_min_max.asp>

SQL Tutorial (n.d.). *SQL NULL Values*. w3schools.com. <https://www.w3schools.com/sql/sql_null_values.asp>

SQL Tutorial (n.d.). *SQL ORDER BY Keyword*. w3schools.com. <https://www.w3schools.com/sql/sql_orderby.asp>

SQL Tutorial (n.d.). *SQL SELECT Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_select.asp>

SQL Tutorial (n.d.). *SQL UPDATE Statement*. w3schools.com. <https://www.w3schools.com/sql/sql_update.asp>

SQL Tutorial (n.d.). *SQL WHERE Clause*. w3schools.com. <https://www.w3schools.com/sql/sql_where.asp>

SQL Tutorial (n.d.). *SQL Wildcards*. w3schools.com. https://www.w3schools.com/sql/sql\_wildcards.asp

SQL Tutorial (n.d.). *SQL Working With Dates*. w3schools.com. <https://www.w3schools.com/sql/sql_dates.asp>

SQLite (n.d.). *Date And Time Functions*. SQLite.  
<https://sqlite.org/lang_datefunc.html>

SQLite Tutorial (n.d.). *SQLite ALTER TABLE*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-alter-table/>

SQLite Tutorial (n.d.). *SQLite BETWEEN*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-between/>

SQLite Tutorial (n.d.). *SQLite Create Table*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-create-table/>

SQLite Tutorial (n.d.). *SQLite Cross Join*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-cross-join/>

SQLite Tutorial (n.d.). *SQLite Delete*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-delete/>

SQLite Tutorial (n.d.). *SQLite Drop Table*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-drop-table/>

SQLite Tutorial (n.d.). *SQLite Group By*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-group-by/>

SQLite Tutorial (n.d.). *SQLite Having*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-having/>

SQLite Tutorial (n.d.). *SQLite In*. SQLite Tutorial. https://www.sqlitetutorial.net/sqlite-in/

SQLite Tutorial (n.d.). *SQLite Inner Join*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-inner-join/>

SQLite Tutorial (n.d.). *SQLite Insert*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-insert/>

SQLite Tutorial (n.d.). *SQLite IS NULL*. SQLite Tutorial. https://www.sqlitetutorial.net/sqlite-is-null/

SQLite Tutorial (n.d.). *SQLite Left Join*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-left-join/>

SQLite Tutorial (n.d.). *SQLite Like*. SQLite Tutorial. https://www.sqlitetutorial.net/sqlite-like/

SQLite Tutorial (n.d.). *SQLite Order By*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-order-by/>

SQLite Tutorial (n.d.). *SQLite Rename Column*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-rename-column/>

SQLite Tutorial (n.d.). *SQLite Select*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-select/>

SQLite Tutorial (n.d.). *SQLite Union*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-union/>

SQLite Tutorial (n.d.). *SQLite Update*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-update/>

SQLite Tutorial (n.d.). *SQLite Where*. SQLite Tutorial. <https://www.sqlitetutorial.net/sqlite-where/>

w3resource (n.d.). *SQL LOWER() function*. w3resource.com. <https://www.w3resource.com/sql/character-functions/lower.php>

## Formative Assessment

1. Which of the following modes of the open() function does not allow you to write to the source file?
   1. "a"
   2. "r"
   3. "r+"
   4. "w"
2. What role does the cursor object play in Python?
3. It carries our commands from Python to SQL.
4. It connects Python with the database.
5. It is the database object in Python.
6. It defines the table in the database with direct access.
7. After querying a SQL table with 50 observations, we use fetchone() to check on the outcome of one record for merely one time. How many records will remain available in the query output after that?
   1. 0
   2. 1
   3. 49
   4. 50
8. Which statement/clause/keyword cannot be embedded in the SELECT statement?
   1. CROSS JOIN
   2. GROUP BY
   3. ORDER BY
   4. RENAME TO
9. Table A has 50 records and table B has 70. A total of 40 records could be matched based on certain conditions. What would be the number of records in the output table if we merged A and B by inner join and cross join, respectively?
   1. 40 and 3500
   2. 50 and 3500
   3. 40 and 50
   4. 50 and 70
10. Which of the following values would not be selected given the following SQL command?

SELECT \* FROM city\_list WHERE city LIKE 'S%g%';

* 1. Singapore.
  2. SINGAP.
  3. S’PORE
  4. SG

1. Which of the following is one of the main differences between USING and ON in an INNER JOIN clause?
   1. Only ON is allowed to use in an INNER JOIN clause.
   2. The matching variable from the second table will not be carried over to the output table of the query.
   3. Missing values of the matching variable from the first table will be replaced by the values of the matching variable from the second table.
   4. We can omit the names of the original tables when using ON in the INNER JOIN clause.
2. In a table called cars, there is a variable named type which has 4 values: A, B, C and D. Type A has 20 records, Type B has 10, Type C has 34, Type D has 7 records. What would be the output table of the following query?

SELECT type, COUNT(type)   
FROM cars   
GROUP BY type   
WHERE COUNT(type) >= 20;

* 1. Empty table
  2. Error
  3. Type C, 34
  4. Type A, 20  
     Type C, 34

1. Which of the following functions is directly supported by SQLite or SQLite3?
   1. OUTER JOIN
   2. DROP COLUMN
   3. REARRANGE COLUMN
   4. RENAME TABLE TO
2. Which of the following is no more possible after the connection to a database is closed?
   1. Check the name of the existing tables in a database
   2. Connect to another database
   3. Rename the database file using file explorer of the operating system
   4. Extract information from a Python object which contains the content of an SQL query

## Suggested Solutions

1. Which of the following modes of the open() function does not allow you to write to the source file?
   1. "a"

*Incorrect. We can append new records to the file in the appending mode.*

* 1. **"r"**

**Correct. We can only extract data from the file in the reading mode.**

* 1. "r+"

*Incorrect. We can add new contents to the file in the updating mode.*

* 1. "w"

*Incorrect. We can write to the file in the writing mode.*

1. What role does the cursor object play in Python?
2. **It carries our commands from Python to SQL.**

**Correct. We send our SQL commands through the cursor object from Python to SQL.**

1. It connects Python with the database.

*Incorrect. The connection object connects Python to the database.*

1. It is the database object in Python.

*Incorrect. There is no direct database object in Python.*

1. It defines the table in the database with direct access.

*Incorrect. The cursor object does not specify the table in the database that we are working on.*

1. After querying a SQL table with 50 observations, we use fetchone() to check on the outcome of one record for merely one time. How many records will remain available in the query output after that?
   1. 0

*Incorrect. The fetchone() function only fetches one record from the query output. So, there must be more than 0 records remaining.*

* 1. 1

*Incorrect. The fetchone() function only fetches one record from the query output. So, there must be more than 1 records remaining.*

* 1. **49**

**Correct. Since the fetchone() function only fetches one record from the query output, there must be 49 records remaining.**

* 1. 50

*Incorrect. The fetchone() function fetches one record from the query output anyway. So, there must be less than 50 records remaining.*

1. Which statement/clause/keyword cannot be embedded in the SELECT statement?
   1. CROSS JOIN

*Incorrect. The CROSS JOIN clause must be embedded in the SELECT statement.*

* 1. GROUP BY

*Incorrect. The GROUP BY statement must be embedded in the SELECT statement.*

* 1. ORDER BY

*Incorrect. The ORDER BY keyword must be embedded in the SELECT statement.*

* 1. **RENAME TO**

**Correct. The RENAME TO keyword must be embedded in the ALTER TABLE statement.**

1. Table A has 50 records and table B has 70. A total of 40 records could be matched based on certain conditions. What would be the number of records in the output table if we merged A and B by inner join and cross join, respectively?
   1. **40 and 3500**

**Correct. Inner join creates the intersection set of both tables, i.e., 40, and cross join returns the cartesian product of both tables, i.e., 3500.**

* 1. 50 and 3500

*Incorrect. Only if A were left joined by B, the number of records of A would return, i.e., 50, and cross join returns the cartesian product of both tables, i.e., 3500.*

* 1. 40 and 50

*Incorrect. Inner join creates the intersection set of both tables, i.e., 40, but only if A were left joined by B, the number of records of A would return, i.e., 50.*

* 1. 50 and 70

*Incorrect. Only if A were left joined by B, the number of records of A would return, i.e., 50, and only if B were left joined by A, the number of records of B would return, i.e., 70.*

1. Which of the following values would not be selected given the following SQL command?

SELECT \* FROM city\_list WHERE city LIKE 'S%g%';

* 1. Singapore.

*Incorrect.* *Since “S” and “g” are parts of this string, SQL would select this value.*

* 1. SINGAP.

*Incorrect. Since “S” and “g” are parts of this string and SQL is not case sensitive, SQL would select this value.*

* 1. **S’PORE**

**Correct. Since “g” is not a sub-string of this string, SQL would not select this value.**

* 1. SG

*Incorrect. Since “S” and “g” are parts of this string and SQL is not case sensitive, SQL would select this value.*

1. Which of the following is one of the main differences between USING and ON in an INNER JOIN clause?
   1. Only ON is allowed to use in an INNER JOIN clause.

*Incorrect. USING and ON are allowed to use in all JOIN clauses.*

* 1. **The matching variable from the second table will not be carried over to the output table of the query.**

**Correct. Only the matching variable from the first table will be carried over to the output table when USING is used in the JOIN clauses.**

* 1. Missing values of the matching variable from the first table will be replaced by the values of the matching variable from the second table.

*Incorrect. It is simply impossible to find matches for missing values in the matching variables from both tables. As a result, there cannot be any replacement.*

* 1. We can omit the names of the original tables when using ON in the INNER JOIN clause.

*Incorrect. We must include the names of the original tables when using ON in the INNER JOIN clause.*

1. In a table called cars, there is a variable named type which has 4 values: A, B, C and D. Type A has 20 records, Type B has 10, Type C has 34, Type D has 7 records. What would be the output table of the following query?

SELECT type, COUNT(type)   
FROM cars   
GROUP BY type   
WHERE COUNT(type) >= 20;

* 1. Empty table

*Incorrect.* *The result would only be possible if the last line were HAVING COUNT(type) >= 40.*

* 1. **Error**

**Correct. We cannot select groups using the WHERE clause on the aggregated results. SQL will return an error to us.**

* 1. Type C, 34

*Incorrect. The result would only be possible if the last line were HAVING COUNT(type) >= 30.*

* 1. Type A, 20  
     Type C, 34

*Incorrect. The result would only be possible if the last line were HAVING COUNT(type) >= 20.*

1. Which of the following functions is directly supported by SQLite or SQLite3?
   1. OUTER JOIN

*Incorrect. OUTER JOIN is not supported by SQLite. We can only outer join two tables using two LEFT JOIN clauses and connecting them with the UNION ALL operator.*

* 1. DROP COLUMN

*Incorrect. DROP COLUMN is not supported by SQLite. We can only drop a column from a table by creating a new table first, defining all the columns except the one that should be dropped and transferring the corresponding data to the new table.*

* 1. REARRANGE COLUMN

*Incorrect. REARRANGE COLUMN is not an SQL function at all. We can rearrange the columns from a table by creating a new table first, defining all the columns in the new sequence and transferring the corresponding data to the new table.*

* 1. **RENAME TABLE TO**

**Correct. RENAME TABLE TO is supported by SQLite. But it is not a stand-alone statement. We can rename the table using the ALTER TABLE RENAME TO statement.**

1. Which of the following is no more possible after the connection to a database is closed?
   1. **Check the name of the existing tables in a database**

**Correct. To check the name of the existing tables in a database, we need to send a query to the master table sql\_master, which means that we need the connection to the database.**

* 1. Connect to another database

*Incorrect. Once the connection to one database is closed, we can connect Python to another database.*

* 1. Rename the database file using file explorer of the operating system

*Incorrect. If we rename the database file outside Python or SQL, we can do it after the connection is closed and the file is not opened by another program anymore.*

* 1. Extract information from a Python object which contains the content of an SQL query

*Incorrect. Since it is a Python object, we do not need a connection to the database to work on it.*