Introduction

In this tutorial, you will learn how to use the `sqlite3` module of Python (`https://docs.python.org/3/library/sqlite3.html`) to work with SQLite databases.

The `sqlite3` module of Python implements an interface between the Python programming language and the SQLite DBMS. It is compliant with the 'DB-API 2.0', which is a specification described by 'PEP 249'; a Python proposal to encourage similarity between Python modules used to work with databases.

Prerequisites

This tutorial has been designed to practice locally on your machine. To get started, ensure Python and SQLite are installed on your machine. The easiest way to check if both are installed is by opening a terminal window and executing the following commands: `python --version` and `sqlite3 --version`. It might be the case that you need to refer to Python on your local machine by using the identifier `python3` instead of `python`. We recommend you make use of Python version ≥ 3, however, the module is available since Python version 2.5. The module's latest version requires SQLite 3.7.15 or newer to be installed.

Refer to the official documentation and reference material of these pieces of software for more information on how to install or upgrade your Python (`https://wiki.python.org/moin/BeginnersGuide/Download`) or SQLite (`https://www.sqlite.org/download.html`) installation if required.

If you prefer not to install any tools locally, an online integrated development environment (IDE) such as Visual Studio Code might also work, see `https://vscode.dev/`.

Getting Started

Python code can be run in various ways. As Python implements a Read-Eval-Print Loop (REPL), it can be used on a line-by-line basis through the terminal after executing `python` (or `python3`). An IDE can also be used. If you have performed a regular installation of Python, you should already have access to IDLE, Python's built-in IDE. Opening IDLE through the terminal can be done by executing `python -m idlelib`. A more elaborate Python IDE we can also recommend is PyCharm or the above-mentioned Visual Studio Code.

Now that we have got the setup out of the way, it is time to start coding! To make use of the `sqlite3` module, we need to import it to include it in our code. Using an IDE, this should generally be done once at the beginning of the script where you wish to make use of the module. When using the REPL, it should be done once per session (when you enter it). Type `import sqlite3`.

Exercises

Consider the SQL tables and data contained in `library.db` as can be created from the `library.sql` file of the previous exercise class. We assume a few records exist in its 'Loans' table during these exercises.

13) Basic Interaction with a Database

To work with the data stored in a database, we need to connect to it from our code. This can be achieved by calling the `connect()` function on the `sqlite3` module and providing the file path to the library database as the only parameter. The result of this operation is a `Connection` object, which should be named `con` for ease of reference.

I. Establish a connection to the database and assign the returned object to a variable named `con`.

Once a connection has been established, we can execute arbitrary SQL statements concerning the database. This can be done by calling the `execute()` function on the `Connection` object with a statement as an argument.

II. Query all rows from the 'Loans' table and assign the returned object to a variable called `res` (short for result).

The returned object is of the type `Cursor`. It contains a database cursor which was implicitly created when querying the database. This cursor allows us to process the results on a row-by-row basis. To fetch all rows that were queried we can call `fetchall()` on this object.

III. Print all rows to the screen.

When we are done working with our database, we can close the connection by calling `close()` on the `Connection` object.

14) SQL Injection

Earlier exercises had you create static queries for the database. Another type of query is a so-called dynamic query. It contains a part that changes on (an) external condition(s).

We want to create a dynamic query that changes based on the user, e.g. to retrieve only the loans made by the current user. To achieve this, imagine our Python application passes us a variable called `user_id`. We can mimic this behaviour by using `user_id = input('Enter a user_id:')`. A programmer familiar with Python could decide to implement this dynamic query using a formatted string, see below.

Dynamic Query Example

```
1 user_id = input('Enter a user_id:')
2 res = con.execute(f"""SELECT isbn
3                       FROM Loans
4                       WHERE mid={user_id}""")
5 print(res.fetchall())
```

While this code works, it has one major issue: it makes us prone to something called SQL injection. SQL injection is a technique of inserting malicious SQL statements into input fields for execution by the backend database. You will learn more about SQL injection during the second-year course 'Security'.

I. What happens when a user tricks our application into receiving the `user_id` '*`1 OR TRUE`*'?

II. Can you think of another SQL injection attack by manipulating the `user_id` input field without using the keyword `TRUE`?

III. Take the following hypothetical `execute()` parameter prone to SQL injection: `f"SELECT ... FROM {parameter} WHERE ..."`. Can you think of a way to prevent the `WHERE` clause from executing via the content of the parameter?

To protect against SQL injection, we should always make use of the placeholders provided by the `sqlite3` module. Two types of placeholders exist: question marks (qmark style) and named placeholders (named style). The latter uses a colon prefix, e.g.: '`:user_id`'. When using qmarks, `execute()` expects a sequence as the second argument (also for a single placeholder!), while it expects a dictionary when making use of the named style.

IV. Implement both qmark and named style, instead of using a formatted string, and attempt to perform the SQL injection attacks again. Do they work?

Make use of either qmark or named style when needing parameters in further exercises.

15) Executing SQL Statements

The library is thriving and wants to expand its offering to include video games. For the time being, they just need a way to store their offering digitally; you do not need to update the Loans relationship.

I. Make use of `execute()` to create a new table called 'Game' with the following schema: **Game**(gid, title, platform).

The module provides two other functions to execute arbitrary SQL. The first is `executemany()`. It executes the enclosed SQL statement for each item in the iterable provided as the second parameter. An example where this behaviour is especially useful is for the insertion of data. Instead of specifying multiple entries in the `VALUES` clause, you only need to specify one using placeholders (so three placeholders in total for the 'Game' table).

II. Fill the 'Game' table with three entries of mock-up data by using an `INSERT` statement with `executemany()`. As you are working with placeholders, do not forget to supply the second argument with data expected by `executemany()`.

Making use of an `INSERT` statement in combination with either `execute()` or `executemany()` implicitly opens something called a database transaction. We will get into the details of transactions in a later lecture. For now, in short, it ensures the statements in a transaction are either all executed, or none are, i.e. they are rolled back. As no further implicit transaction handling is performed after the execution of the statement(s), we need to explicitly 'close' the transaction. To do so and have the data

actually put in the database, we can call the `commit()` function on the `Connection` object. It is also possible to abort the transaction by calling `rollback()`. This clears all pending changes.

III. Ensure the data is put in the database by 'correctly' closing the implicitly opened transaction.

IV. Transactions are also implicitly opened for other SQL-DML keywords. Can you think of the three other keywords for which this occurs?

To retrieve the number of rows modified by a certain function call, the `rowcount` property can be read on the resulting `Cursor` object. To retrieve the number of rows modified in total, `total_changes` can instead be read on the `Connection` object.

V. Print the total number of changed rows on the screen, both in total and just by the previously executed query.

The third and final function the module provides for SQL execution is `executescript()`. It allows multiple - possibly different - SQL statements to be executed within the same function call. An important thing to note is that it does not perform implicit transaction control. It does commit any open transaction before the execution of its statement(s).

VI. Compare `execute()`, `executemany()` and `executescript()` for arbitrary SQL execution against each other. What do they have in common? What are their differences? When should you pick one over the other?

16) Query Results

The librarians do not trust your computer wizardry. To gain back their trust, you are tasked with performing various queries to show that their valuable data is stored safely.

I. Write a call to the `execute()` function to query all entries in the 'Loans' table. Save the response object in a variable called `res`.

There are multiple ways in which you can go through the resulting `Cursor` object:

- As you have seen in the first exercise, `fetchall()` returns all remaining rows as a list of tuples. If there are no remaining rows, an empty list is returned;
- `fetchone()` Returns the next result as a single tuple. If no more data is available, `None` is returned;
- `fetchmany()` Can be used to retrieve an arbitrary amount of results. By default, `fetchmany()` fetches the amount set using the `arraysize` property on the `Cursor` object which is, by default, set to one. For performance reasons, it is recommended to set a consistent size using this property. It is, however, possible to override this property by providing a size as an argument to `fetchmany()`.

An alternative method to the above is using a for-loop on the `Cursor` object. In its simplest form, it works like `fetchall()`. You can, however, break out of the loop at any time of your liking. In this way, it can also act like the other two functions.

II. Print all queried rows on a new line by making use of a for-loop.

III. Execute both `fetchone()` and `fetchall()` after the for-loop on `res`. Is the result what you expected?

As you might have noticed, when iterating through the `Cursor` object, you can only make use of the results once. This behaviour is called consumption. It is sometimes also referred to as a function side-effect.

The library has grown significantly while you were working on these exercises. Using `fetchall()` or a for-loop to retrieve the results at once is no longer possible due to memory issues. Therefore, out of the above-mentioned methods, only `fetchone()` and `fetchmany()` remain reasonably usable.

If you want to try this out yourself, feel free to run the code below on your machine. On the author's machine, this resulted in a result of about 5 GiB. Do adjust the number of records and for-loop iterations to how strong you estimate your machine to be.

Library Growth Simulator

```
con.execute('INSERT INTO Loans VALUES(?, ?, ?, ?)',
            [(i,) for i in range(10000, 2000000)])
con.commit()
query = 'SELECT * FROM Loans'
for i in range(10):
    query += ' UNION ALL SELECT * FROM Loans'
res = con.execute(query)
print('Done, fetching all:')
print(res.fetchall())  # / fetchone() / fetchmany()
```

IV. Compare `fetchone()` and `fetchmany()` to each other. What are their advantages and disadvantages?

V. Print all queried rows to the screen again; this time by making use of `fetchmany()` with a size of two. If you ran the example code above, feel free to increase the size.

You have finished this week's exercise class! You should now understand the basic principles behind an interface and how to work with one. As a prior and further reading, we suggest the official documentation at `https://docs.python.org/3/library/sqlite3.html`. It might also be interesting to take a look at how databases can be combined with well-known Python libraries such as Pandas or Numpy. Another 'library' we can suggest is the advanced SQLAlchemy database toolkit.

Go to the next page for extra fun bonus challenges to solve!

Bonus Exercises

This section acts as a bonus for those interested. It goes into more advanced and specific topics. The topics within this set of exercises do not depend on one another, so feel free to skip the ones you find too difficult or simply not interesting. The difficulty of a topic is displayed right next to it.

17) Discussing SQL Statements (easy)
We can use the `sqlite3` module to check whether a string contains a complete SQL statement by calling `complete_stament()` on the module with the string as an argument. You are given four queries and their intent with regard to `library.db`:

- All records in the 'Book' table: `SELECT * FROM Book;`
- All adult (18 or older) members: `SELECT * FROM Member WHERE age > 18`
- All records in the database: `SELECT *`
- All records in the 'Dvd' table: `SELECT * FROM Dvd;`

I. For each query, use `complete_statement()` to determine its completeness.

When discussing queries, we can also say something about their syntactical and semantical properties. If we look at the syntax of a query, we check whether it is valid SQL. When considering the semantics of a query, we compare the intent of a query against its result. Something is semantically correct if and only if its result matches its intent.

II. For each query, determine whether it is *syntactically* correct (syntax).

III. For each query, determine whether it is *semantically* correct (intent).

IV. Is it possible to have a statement that is *syntactically* incorrect, but *semantically* correct?

18) Performing a Backup (medium)
The module provides a function to perform a database backup called `backup()`, which can be called on the `Connection` object you want to back up. As a mandatory argument, this function also takes a `Connection` object, which refers to the target to store the backup at. One of the other arguments it takes is called `progress`, which should be a callback function to report progress information. It provides three integer arguments to this callback function: the status of the last iteration, the remaining number of pages to be copied and the total number of pages.

I. Write a function that takes the three arguments given by `backup()` to display backup progress to the screen.

II. Perform a backup operation using `backup()` with your callback function. To better be able to see your callback function in action, it might help to increase the `sleep` (in seconds) parameter of `backup()` and to decrease its `pages` parameter to something low above zero, such as one.

19) Aggregate Functions (medium, recommended as it is quite fun!) The module allows you to combine queries with Python function calls. These together form a very powerful and customisable tool for you as a programmer. This exercise will focus on creating our own aggregate function called `range_sum`.

As the name suggests, our aggregate function must only sum values within a specified range, in contrast to the built-in `sum()`, which sums all retrieved values. A call to our aggregate function within a SQL statement would look like this: `SELECT range_sum(2, 8, attribute) FROM table;`. The first two arguments specify the range start and end (both inclusive) and the third the attribute to `SELECT`.

To define our own aggregate function, we need various methods: one to initialise its starting value: `__init__()`; one to act on the retrieval of a new row: `step()`; and one to return the final result after all rows have been processed: `finalize()`.

The `step()` function is allowed to take an arbitrary amount of arguments to fit any use case. In our case, it will be provided three arguments: the beginning of the range, the end of the range and the queried attribute's value. It should be able to take these arguments as parameters.

As the functions are all related, it is best to group them into a class. As not everyone might be familiar with OOP programming in Python yet, a base is given below:

Custom Aggregate Class

```
class RangeSum:
    def __init__(self):
        self.sum = 0

    def step(self):
        pass  # Remove on implementation

    def finalize(self):
        pass  # Remove on implementation
```

I. Write the `step()` and `finalize()` functions.

To make the module aware of our custom aggregate function we need to register it. This can be done by calling the `create_aggregate()` function on the `Connection` object. It expects the following arguments: the name of the aggregate function (we defined it as `range_sum` in the example), the number of arguments it takes, which should be equal to the number of arguments of `step()` while not including `self`, and the name of the class containing the functions related to our aggregate function.

II. Register the custom aggregate function and make use of `execute()` and `fetchone()` to try it out.

20) Adapters and Converters (hard)

As you might know, SQLite supports five types: `NULL`, `INTEGER`, `REAL`, `TEXT` and `BLOB`. This differs from Python, which comes with more built-in primitive types: `int`, `real`, `list`, `tuple` and `string` to name a few. As you have seen earlier, we were able to directly use queried results in our code, e.g. to print them to the screen. You might be wondering how this is possible, as `TEXT` does not exist in Python and `string` does not exist in SQLite, for example. The `sqlite3` module's type system provides this seamless, automatic, conversion for the types in Table 1.

Python Type	SQLite Type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

Table 1: Automatic type conversion by the `sqlite3` module.

It is possible to build upon this type system to extend its behaviour, which allows the module to automatically transform custom Python classes as well. This has been made available through the process of *adapting* and *converting*. *Adapters* transform custom Python types to SQLite values, *adapting* them to SQLite, while *converters* convert SQLite values to custom Python types.

Note that we convert to SQLite **values**, not types. The important difference here is that by using *converters* we encode the custom Python object as a value within an already existing SQLite type. In other words, we are not creating custom SQLite types; we are using those that already exist, but interpret their contents differently.

An example would be a two-dimensional point. To encode this, we can decide to store it as the SQLite type `TEXT` in the following format: `x,y`. As not everyone might be familiar with OOP programming in Python yet, the `Point` class is given below:

Custom Point Class

```
1 class Point:
2     def __init__(self, x, y):
3         self.x, self.y = x, y
```

I. Write an adapter function called `adapt_point()`. It should take a `Point` object as an argument and return a string with the point encoded as the textual representation `x,y`.

II. Write a converter function called `convert_point()`. It should take a bytes object given by SQLite as an argument and return a new `Point` instance. You can assume that only valid points are stored in the database.

To make the module aware of your custom *adapter* and *converter*, you need to register them. To register the custom *adapter* call `register_adapter()` on the module, which takes the class and function name as arguments. To register the custom *converter* call `register_converter()` on the module, which takes the name to be used in SQL for the custom type, as well as the function name as arguments.

III. Register both the *adapter* and *converter*.

Last but not least, we need to tell SQLite which attributes within a table should make use of the custom *adapter* and *converter* functions for conversion and adaption to occur automatically. This can be done in three ways:

- Have the module parse the information from the **query** attribute name: `PARSE_COLNAMES`. This introduces a new format where the type name is wrapped in square brackets, e.g. `SELECT attribute as "attribute [type_name]" FROM table;`;
- Have the module parse it by using the declared type used when `CREATING` the table using SQL-DDL: `PARSE_DECLTYPES`;
- Both of the above, where `PARSE_COLNAMES` always takes precedence over `PARSE_DECLTYPES`: `PARSE_COLNAMES | PARSE_DECLTYPES`.

To use one of these methods, provide the `detect_types` argument with the wanted method's name to the `connect()` function.

IV. Go back to the line of code where you made the initial connection to your database using `connect()`. Set its `detect_types` parameter to `PARSE_COLNAMES`, `PARSE_DECLTYPES` or `PARSE_COLNAMES | PARSE_DECLTYPES`.

Depending on which method you chose, you either need to implement your custom type's name in your query, or define the type of an attribute within the table equal to your custom type's name.

V. `CREATE` a new table with a point attribute, `INSERT` a new point and `fetchone()` the result to verify the process of adapting and converting is working as expected.

21) Row Factories (medium)

As you have seen in an earlier exercise, fetching a row returns a tuple. This is the default behaviour of the module and can be changed according to your liking. Another class that comes with the module is called `Row`. It is an optimised implementation of something similar to a regular dictionary. An implementation that processes rows and returns an object for the user to make use of the retrieved row is called a row factory. `Row` is an example of a row factory.

It is recommended to set the type of row returned by the module on the `Connection` object by using its `row_factory` attribute to ensure all cursors derived from this connection return the same, expected, type.

I. Change the row factory to the `Row` class by using the `row_factory` attribute on the `Connection` object.

II. Write a query that results in at least one record from your database. Use `fetchone()` and index an attribute within the fetched row by name to print it to the screen.

To create a custom row factory, all you need is a function that accepts two arguments: a `Cursor` object and a row, which is always given to a row factory as a tuple.

III. Write a function to be used as a row factory that returns the current row as a frozen set.

IV. Set the used row factory to your custom function.

V. Retrieve multiple rows from your database. Make use of `fetchall()` and print the resulting list.