<https://digitology.tech/posts/uchebnik-po-sqlite3-v-python/>

SQLite – это C библиотека, реализующая легковесную дисковую базу данных (БД), не требующую отдельного серверного процесса и позволяющую получить доступ к БД с использованием языка запросов SQL. Некоторые приложения могут использовать SQLite для внутреннего хранения данных. Также возможно создать прототип приложения с использованием SQLite, а затем перенести код в более многофункциональную БД, такую как PostgreSQL или Oracle.

[Модуль sqlite3](https://digitology.tech/docs/python_3/library/sqlite3.html" \o "Модуль sqlite3) реализует интерфейс SQL, соответствующий спецификации DB-API 2.0, описанной в PEP 249.

## Создание соединения

Чтобы воспользоваться SQLite3 в Python необходимо импортировать [модуль sqlite3](https://digitology.tech/docs/python_3/library/sqlite3.html" \o "модуль sqlite3), а затем создать объект подключения к БД.

Объект подключения создается с помощью метода connect():

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | **con** = sqlite3.connect('mydatabase.db') |

## Курсор SQLite3

Для выполнения операторов SQL, нужен объект курсора, создаваемый методом cursor().

Курсор SQLite3 – это метод объекта соединения. Для выполнения операторов SQLite3 сначала устанавливается соединение, а затем создается объект курсора с использованием объекта соединения следующим образом:

|  |  |
| --- | --- |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | cursorObj = con.cursor() |

Теперь можно использовать объект курсора для вызова метода execute() для выполнения любых запросов SQL.

## Создание базы данных

После создания соединения с SQLite, файл БД создается автоматически, при условии его отсутствия. Данный файл создаётся на диске, но также можно создать базу данных в оперативной памяти, используя параметр «:memory:» в методе connect. При этом база данных будет называется инмемори.

Рассмотрим приведенный ниже код, в котором создается БД с блоками try, except и finally для обработки любых исключений:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | **from** sqlite3 **import** Error |
|  |  |
|  | **def** **sql\_connection**(): |
|  |  |
|  | **try**: |
|  |  |
|  | con = sqlite3.connect(':memory:') |
|  |  |
|  | print("Connection is established: Database is created in memory") |
|  |  |
|  | **except** Error: |
|  |  |
|  | print(Error) |
|  |  |
|  | **finally**: |
|  |  |
|  | con.close() |
|  |  |
|  | sql\_connection() |

Сначала импортируется модуль sqlite3, затем определяется функция с именем sql\_connection. Внутри функции определен блок try, где метод connect() возвращает объект соединения после установления соединения.

Затем определен блок исключений, который в случае каких-либо исключений печатает сообщение об ошибке. Если ошибок нет, соединение будет установлено, тогда скрипт распечатает текст «Connection is established: Database is created in memory».

Далее производится закрытие соединения в блоке finally. Закрытие соединения необязательно, но это хорошая практика программирования, позволяющая освободить память от любых неиспользуемых ресурсов.

## Создание таблицы

Чтобы создать таблицу в SQLite3, выполним запрос Create Table в методе execute(). Для этого выполним следующую последовательность шагов:

1. Создание объекта подключения
2. Объект Cursor создаётся с использованием объекта подключения
3. Используя объект курсора, вызывается метод execute с SQL запросом create table в качестве параметра.

Давайте создадим таблицу Employees со следующими колонками:

employees (id, name, salary, department, **position**, hireDate)

Код будет таким:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | **from** sqlite3 **import** Error |
|  |  |
|  | **def** **sql\_connection**(): |
|  |  |
|  | **try**: |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **return** con |
|  |  |
|  | **except** Error: |
|  |  |
|  | print(Error) |
|  |  |
|  | **def** **sql\_table**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute("CREATE TABLE employees(id integer PRIMARY KEY, name text, salary real, department text, position text, hireDate text)") |
|  |  |
|  | con.commit() |
|  |  |
|  | con = sql\_connection() |
|  | sql\_table(con) |

В приведенном выше коде определено две функции: первая устанавливает соединение; а вторая - используя объект курсора выполняет SQL оператор create table.

Метод commit() сохраняет все сделанные изменения. В конце скрипта производится вызов обеих функций.

Для проверки существования таблицы воспользуемся браузером БД для sqlite.

## Вставка данных в таблицу

Чтобы вставить данные в таблицу воспользуемся оператором INSERT INTO. Рассмотрим следующую строку кода:

**cursorObj**.execute("INSERT INTO employees VALUES(1, 'John', 700, 'HR', 'Manager', '2017-01-04')")

Также можем передать значения / аргументы в оператор INSERT в методе execute (). Также можно использовать знак вопроса (?) в качестве заполнителя для каждого значения. Синтаксис INSERT будет выглядеть следующим образом:

cursorObj.execute('''INSERT INTO employees(id, name, salary, department, position, hireDate) VALUES(?, ?, ?, ?, ?, ?)''', entities)

Где картеж entities содержат значения для заполнения одной строки в таблице:

entity = (2, 'Andrew', 800, 'IT', 'Tech', '2018-02-06')

Код выглядит следующим образом:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_insert**(con, entities): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('INSERT INTO employees(id, name, salary, department, position, hireDate) VALUES(?, ?, ?, ?, ?, ?)', entities) |
|  |  |
|  | con.commit() |
|  |  |
|  | entities = (2, 'Andrew', 800, 'IT', 'Tech', '2018-02-06') |
|  |  |
|  | sql\_insert(con, entities) |

## Обновление таблицы

Предположим, что нужно обновить имя сотрудника, чей идентификатор равен 2. Для обновления будем использовать инструкцию UPDATE. Также воспользуемся предикатом WHERE в качестве условия для выбора нужного сотрудника.

Рассмотрим следующий код:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_update**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('UPDATE employees SET name = "Rogers" where id = 2') |
|  |  |
|  | con.commit() |
|  |  |
|  | sql\_update(con) |

Это изменит имя Andrew на Rogers.

## Оператор SELECT

Оператор SELECT используется для выборки данных из одной или более таблиц. Если нужно выбрать все столбцы данных из таблицы, можете использовать звёздочку (\*). SQL синтаксис для этого будет следующим:

**select** \* **from** table\_name

В SQLite3 инструкция SELECT выполняется в методе execute объекта курсора. Например, выберем все стрики и столбцы таблицы employee:

**cursorObj**.execute('SELECT \* FROM employees ')

Если нужно выбрать несколько столбцов из таблицы, укажем их, как показано ниже:

**select** column1, column2 from tables\_name

Например,

**cursorObj**.execute('SELECT id, name FROM employees')

Оператор SELECT выбирает все данные из таблицы employees БД.

## Выборка всех данных

Чтобы извлечь данные из БД выполним инструкцию SELECT, а затем воспользуемся методом fetchall() объекта курсора для сохранения значений в переменной. При этом переменная будет являться списком, где каждая строка из БД будет отдельным элементом списка. Далее будет выполняться перебор значений переменной и печатать значений.

Код будет таким:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_fetch**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('SELECT \* FROM employees') |
|  |  |
|  | rows = cursorObj.fetchall() |
|  |  |
|  | **for** row **in** rows: |
|  |  |
|  | print(row) |
|  |  |
|  | sql\_fetch(con) |

Также можно использовать fetchall() в одну строку:

[print(row) for row in cursorObj.fetchall()]

Если нужно извлечь конкретные данные из БД, воспользуйтесь предикатом WHERE. Например, выберем идентификаторы и имена тех сотрудников, чья зарплата превышает 800. Для этого заполним нашу таблицу большим количеством строк, а затем выполним запрос.

Можете использовать оператор INSERT для заполнения данных или ввести их вручную в программе браузера БД.

Теперь, выберем имена и идентификаторы тех сотрудников, у кого зарплата больше 800:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_fetch**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('SELECT id, name FROM employees WHERE salary > 800.0') |
|  |  |
|  | rows = cursorObj.fetchall() |
|  |  |
|  | **for** row **in** rows: |
|  |  |
|  | print(row) |
|  |  |
|  | sql\_fetch(con) |

В приведенном выше операторе SELECT вместо звездочки (\*) были указаны атрибуты id и name.

## SQLite3 rowcount

Счётчик строк SQLite3 используется для возврата количества строк, которые были затронуты или выбраны последним выполненным запросом SQL.

Когда вызывается rowcount с оператором SELECT, будет возвращено -1, поскольку количество выбранных строк неизвестно до тех пор, пока все они не будут выбраны. Рассмотрим пример:

print(cursorObj.execute('SELECT \* FROM employees').rowcount)

Поэтому, чтобы получить количество строк, нужно получить все данные, а затем получить длину результата:

|  |  |
| --- | --- |
|  | rows = cursorObj.fetchall() |
|  | print(len(rows)) |

Когда оператор DELETE используется без каких-либо условий (предложение where), все строки в таблице будут удалены, а общее количество удаленных строк будет возвращено rowcount.

print(cursorObj.execute('DELETE FROM employees').rowcount)

Если ни одна строка не удалена, будет возвращено 0.

## Список таблиц

Чтобы вывести список всех таблиц в базе данных SQLite3, нужно обратиться к таблице sqlite\_master, а затем использовать fetchall() для получения результатов из оператора SELECT.

Sqlite\_master - это главная таблица в SQLite3, в которой хранятся все таблицы.

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_fetch**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('SELECT name from sqlite\_master where type= "table"') |
|  |  |
|  | print(cursorObj.fetchall()) |
|  |  |
|  | sql\_fetch(con) |

## Проверка существования таблицы

При создании таблицы необходимо убедиться, что таблица еще не существует. Аналогично, при удалении таблицы она должна существовать.

Чтобы проверить, если таблица еще не существует, используем «if not exists» с оператором CREATE TABLE следующим образом:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_fetch**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('create table if not exists projects(id integer, name text)') |
|  |  |
|  | con.commit() |
|  |  |
|  | sql\_fetch(con) |

Точно так же, чтобы проверить, существует ли таблица при удалении, мы используем «if not exists» с инструкцией DROP TABLE следующим образом:

**cursorObj**.execute('drop table if exists projects')

Также проверим, существует ли таблица, к которой нужно получить доступ, выполнив следующий запрос:

|  |  |
| --- | --- |
|  | **cursorObj**.execute('SELECT name from sqlite\_master WHERE type = "table" AND name = "employees"') |
|  |  |
|  | **print**(cursorObj.fetchall()) |

Если указанное имя таблицы не существует, будет возвращен пустой массив.

## Удаление таблицы

Удаление таблицы выполняется с помощью оператора DROP. Синтаксис оператора DROP выглядит следующим образом:

**drop** **table** table\_name

Чтобы удалить таблицу, таблица должна существовать в БД. Поэтому рекомендуется использовать «if exists» с оператором DROP. Например, удалим таблицу employees:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | **def** **sql\_fetch**(con): |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('DROP table if exists employees') |
|  |  |
|  | con.commit() |
|  |  |
|  | sql\_fetch(con) |

## Исключения SQLite3

Исключением являются ошибки времени выполнения скрипта. При программировании на Python все исключения являются экземплярами класса производного от BaseException.

В SQLite3 у есть следующие основные исключения Python:

**DatabaseError**

Любая ошибка, связанная с базой данных, вызывает ошибку DatabaseError.

**IntegrityError**

IntegrityError является подклассом DatabaseError и возникает, когда возникает проблема целостности данных, например, когда внешние данные не обновляются во всех таблицах, что приводит к несогласованности данных.

**ProgrammingError**

Исключение ProgrammingError возникает, когда есть синтаксические ошибки или таблица не найдена или функция вызывается с неправильным количеством параметров / аргументов.

**OperationalError**

Это исключение возникает при сбое операций базы данных, например, при необычном отключении. Не по вине программиста.

**NotSupportedError**

При использовании некоторых методов, которые не определены или не поддерживаются базой данных, возникает исключение NotSupportedError.

## Массовая вставка строк в Sqlite

Для вставки нескольких строк одновременно использовать оператор executemany.

Рассмотрим следующий код:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('create table if not exists projects(id integer, name text)') |
|  |  |
|  | **data** = [(1, "Ridesharing"), (2, "Water Purifying"), (3, "Forensics"), (4, "Botany")] |
|  |  |
|  | cursorObj.executemany("INSERT INTO projects VALUES(?, ?)", **data**) |
|  |  |
|  | con.commit() |

Здесь создали таблицу с двумя столбцами, тогда у «данных» есть четыре значения для каждого столбца. Эта переменная передается методу executemany() вместе с запросом.

Обратите внимание, что использовался заполнитель для передачи значений.

## Закрытие соединения

Когда работа с БД завершена, рекомендуется закрыть соединение. Соединение может быть закрыто с помощью метода close().

Чтобы закрыть соединение, используйте объект соединения с вызовом метода close() следующим образом:

|  |  |
| --- | --- |
|  | **con** = sqlite3.connect('mydatabase.db') |
|  |  |
|  | #program statements |
|  |  |
|  | con.close() |

## SQLite3 datetime

В базе данных Python SQLite3 можно легко сохранять дату или время, импортируя [Python модуль datetime](https://digitology.tech/docs/python_3/library/datetime.html). Следующие форматы являются наиболее часто используемыми форматами для даты и времени:

|  |  |
| --- | --- |
|  | YYYY-MM-DD |
|  |  |
|  | YYYY-MM-DD HH:MM |
|  |  |
|  | YYYY-MM-DD HH:MM:SS |
|  |  |
|  | YYYY-MM-DD HH:MM:SS.SSS |
|  |  |
|  | **HH:MM** |
|  |  |
|  | **HH:MM:SS** |
|  |  |
|  | **HH:MM:SS.SSS** |
|  |  |
|  | now |

Рассмотрим следующий код:

|  |  |
| --- | --- |
|  | **import** sqlite3 |
|  |  |
|  | **import** datetime |
|  |  |
|  | con = sqlite3.connect('mydatabase.db') |
|  |  |
|  | cursorObj = con.cursor() |
|  |  |
|  | cursorObj.execute('create table if not exists assignments(id integer, name text, date date)') |
|  |  |
|  | **data** = [(1, "Ridesharing", datetime.date(2017, 1, 2)), (2, "Water Purifying", datetime.date(2018, 3, 4))] |
|  |  |
|  | cursorObj.executemany("INSERT INTO assignments VALUES(?, ?, ?)", **data**) |
|  |  |
|  | con.commit() |

В этом коде [модуль datetime](https://digitology.tech/docs/python_3/library/datetime.html) импортируется первым, далее создали таблицу с именем assignments с тремя столбцами.

Тип данных третьего столбца - дата. Чтобы вставить дату в столбец, воспользовались datetime.date. Точно так же можно использовать datetime.time для обработки времени.

## Вывод

SQLite можно использовать в своих разработках, но с учетом особенностей этой БД. SQLite прекрасно подойдет для проектов у которых мало операций записи, не нужна система прав доступа к БД и ограниченны ресурсы сервера.