**Работа с SQLite в Python**

### Введение

В этом уроке будут задействованы все основные аспекты использования SQLite, начиная с установки и создания базы данных, и заканчивая выполнением сложных запросов и управлением транзакциями.

#### Почему SQLite?

SQLite - это компактная и легкая встраиваемая база данных, которая позволяет хранить и управлять данными прямо внутри вашего приложения. Её простота в использовании и широкая поддержка делают её прекрасным выбором для различных проектов, включая веб-приложения, мобильные приложения и многое другое.

#### Преимущества использования SQLite

1. **Простота**: SQLite не требует отдельного сервера или настройки. Она работает как часть вашего приложения.
2. **Надежность**: Транзакции и ACID-свойства обеспечивают надежность и целостность данных.
3. **Кроссплатформенность**: SQLite поддерживается на множестве платформ, включая Windows, macOS и Linux.
4. **Эффективность**: Использует минимальные ресурсы системы, что делает её хорошим выбором для мобильных устройств.

#### Обзор плана туториала

Этот урок разделен на несколько логических разделов, чтобы обеспечить пошаговое понимание работы с SQLite в Python.

1. **Установка и настройка**: Узнаем, как установить библиотеку sqlite3, создать базу данных и подключиться к существующей. Рассмотрим также инструменты для удобной работы с SQLite.
2. **Создание и управление таблицами**: Углубимся в создание таблиц и их управление. Изучим различные типы данных SQLite.
3. **Вставка, обновление и удаление данных**: Узнаем, как добавлять, обновлять и удалять данные в таблицах. Рассмотрим примеры для каждой из операций.
4. **Выполнение запросов**: Изучим использование SQL-запросов для извлечения данных из таблиц. Познакомимся с операторами SELECT, FROM, WHERE, GROUP BY, HAVING и ORDER BY.
5. **Обработка результатов**: Узнаем, как получать результаты запросов и преобразовывать их в более удобные структуры данных. Рассмотрим работу с NULL-значениями.
6. **Транзакции и управление данными**: Изучим понятие транзакций, их важность и использование операторов BEGIN, COMMIT и ROLLBACK. Узнаем, как использовать контекстные менеджеры для автоматического управления транзакциями.
7. **Продвинутые концепции**: Рассмотрим подготовленные запросы для оптимизации производительности, создание представлений для упрощения сложных запросов, а также использование триггеров и индексов.
8. **Примеры использования**: Увидим, как SQLite может быть применена в различных сценариях, и реализуем простое приложение для управления задачами с использованием SQLite и Python.

### Установка и настройка

#### Установка библиотеки sqlite3 в Python

Чтобы начать работу с SQLite в Python, нам потребуется библиотека sqlite3. Эта библиотека входит в стандартную библиотеку Python (начиная с версии 2.5), поэтому нам не нужно устанавливать её отдельно. Мы можем начать использовать sqlite3 сразу после установки Python.

#### Создание новой базы данных и подключение к существующей

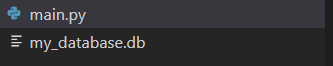
Для начала работы с SQLite, давайте создадим новую базу данных и подключимся к ней:

import sqlite3

# Создаем подключение к базе данных (файл my\_database.db будет создан)

connection = sqlite3.connect('my\_database.db')

connection.close()



#### Инструмент для работы с SQLite

Для удобной работы с базой данных SQLite, вы можете использовать инструменты, такие как DB Browser for SQLite (или SQLite Studio). Это графический интерфейс, позволяющий просматривать, редактировать и управлять данными в базе SQLite.

DB Browser for SQLite можно скачать с официального сайта: [sqlitebrowser.org](https://sqlitebrowser.org/).

### Создание и управление таблицами

#### Создание таблицы с определением структуры и типов данных

Давайте начнем с создания таблицы, в которой будем хранить информацию о пользователях id, username. email и age:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Создаем таблицу Users

cursor.execute('''

CREATE TABLE IF NOT EXISTS Users (

id INTEGER PRIMARY KEY,

username TEXT NOT NULL,

email TEXT NOT NULL,

age INTEGER

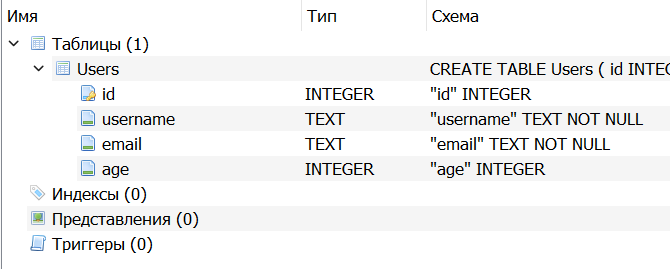
)

''')

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



cursor = connection.cursor() - это строка кода, которая создает объект "курсор" для выполнения SQL-запросов и операций с базой данных. В контексте работы с базами данных в Python, когда вы устанавливаете соединение с базой данных, вы создаете объект соединения (sqlite3.connect('my\_database.db') в данном случае), который представляет собой связь между вашим кодом и фактической базой данных.

Курсор же - это механизм, который позволяет вам отправлять SQL-запросы базе данных и получать результаты этих запросов. Курсор предоставляет методы для выполнения SQL-запросов (execute(), executemany()) и получения результатов (fetchone(), fetchall() и т.д.).

В нашем коде cursor = connection.cursor() создает объект курсора, связанный с соединением connection, и этот курсор будет использоваться для выполнения операций с базой данных, таких как создание таблицы, вставка данных, выборка данных и так далее. Когда вы вызываете методы курсора для выполнения SQL-запросов (cursor.execute()), изменения применяются к базе данных через это соединение.

Мы по сути говорим питону:

1. Подключись к базе данных 'my\_database.db'
2. Если таблица не существует (NOT EXISTS), то создай таблицу с названием "Users"
3. Если таблица существует (EXISTS), то пропускай создание таблиц.
4. Создай данные id как 'INTEGER', сделав первичным ключом.
5. Создай username, email и без NULL (чтобы не был пустым)
6. Сохрани изменение и закрой соединение.

#### Описание основных типов данных SQLite

SQLite поддерживает различные типы данных, которые могут использоваться при создании таблиц. Основные типы данных включают:

1. **INTEGER**: Целые числа.
2. **TEXT**: Текстовые данные.
3. **REAL**: Числа с плавающей запятой.
4. **BLOB**: Двоичные данные.

#### Индексы и их роль в оптимизации запросов

Индексы позволяют ускорить выполнение запросов, особенно при работе с большими объемами данных. Давайте создадим индекс для столбца email в таблице Users:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

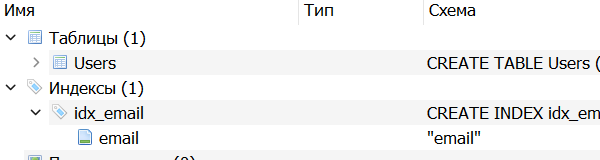
# Создаем индекс для столбца "email"

cursor.execute('CREATE INDEX idx\_email ON Users (email)')

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



### Вставка, обновление и удаление данных

#### Использование оператора INSERT для добавления данных

Чтобы добавить данные в таблицу, используйте оператор INSERT. Давайте добавим нового пользователя в таблицу "Users":

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

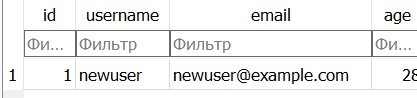
# Добавляем нового пользователя

cursor.execute('INSERT INTO Users (username, email, age) VALUES (?, ?, ?)', ('newuser', 'newuser@example.com', 28))

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



#### Обновление записей с использованием UPDATE

Используйте оператор UPDATE для обновления записей в таблице. Например, давайте изменим возраст пользователя с именем "newuser":

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

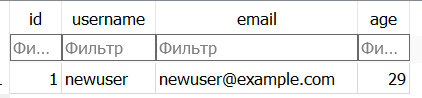
# Обновляем возраст пользователя "newuser"

cursor.execute('UPDATE Users SET age = ? WHERE username = ?', (29, 'newuser'))

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



#### Удаление данных с помощью оператора DELETE

Используйте оператор DELETE для удаления данных из таблицы. Например, давайте удалим пользователя с именем "newuser":

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

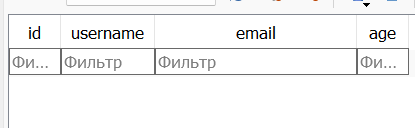
# Удаляем пользователя "newuser"

cursor.execute('DELETE FROM Users WHERE username = ?', ('newuser',))

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



### Выполнение запросов

#### Использование SQL-запросов для извлечения данных из таблиц

Для извлечения данных из таблицы используйте SQL-запросы. Давайте выполним запрос для выбора всех пользователей из таблицы "Users":

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем всех пользователей

cursor.execute('SELECT \* FROM Users')

users = cursor.fetchall()

# Выводим результаты

for user in users:

print(user)

# Закрываем соединение

connection.close()



#### Применение операторов SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

Одним из мощных инструментов SQL является возможность выполнения сложных запросов с использованием операторов SELECT, FROM, WHERE, GROUP BY, HAVING и ORDER BY. Эти операторы позволяют фильтровать, группировать и сортировать данные для получения нужных результатов.

### Примеры операторов SELECT, FROM, WHERE

Оператор SELECT позволяет выбрать определенные столбцы из таблицы:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем имена и возраст пользователей старше 25 лет

cursor.execute('SELECT username, age FROM Users WHERE age > ?', (25,))

results = cursor.fetchall()

for row in results:

print(row)

connection.close()



### Примеры операторов GROUP BY и HAVING

Оператор GROUP BY используется для группировки данных по определенным столбцам. Оператор HAVING применяется к агрегатным функциям, чтобы фильтровать результаты групп.

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Получаем средний возраст пользователей для каждого возраста

cursor.execute('SELECT age, AVG(age) FROM Users GROUP BY age')

results = cursor.fetchall()

for row in results:

print(row)

# Фильтруем группы по среднему возрасту больше 30

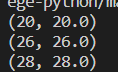
cursor.execute('SELECT age, AVG(age) FROM Users GROUP BY age HAVING AVG(age) > ?', (30,))

filtered\_results = cursor.fetchall()

for row in filtered\_results:

print(row)

connection.close()



### Примеры оператора ORDER BY

Оператор ORDER BY используется для сортировки результатов по указанным столбцам:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем и сортируем пользователей по возрасту по убыванию

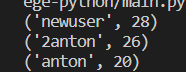
cursor.execute('SELECT username, age FROM Users ORDER BY age DESC')

results = cursor.fetchall()

for row in results:

print(row)

connection.close()



### Примеры комбинирования операторов

Можно комбинировать операторы для выполнения более сложных запросов. Например, выберем пользователей, у которых средний возраст в группе больше 30:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем и сортируем пользователей по возрасту по убыванию

cursor.execute('''

SELECT username, age, AVG(age)

FROM Users

GROUP BY age

HAVING AVG(age) > ?

ORDER BY age DESC

''', (30,))

results = cursor.fetchall()

for row in results:

print(row)

connection.close()



#### Использование агрегатных функций: COUNT, SUM, AVG, MIN, MAX

Агрегатные функции позволяют вычислять значения по группам данных или над всей таблицей. Давайте рассмотрим каждую из агрегатных функций на примерах.

### COUNT - подсчет количества записей

Функция COUNT используется для подсчета количества записей в столбце или таблице.

Пример подсчета всех пользователей в таблице:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Подсчет общего числа пользователей

cursor.execute('SELECT COUNT(\*) FROM Users')

total\_users = cursor.fetchone()[0]

print('Общее количество пользователей:', total\_users)

connection.close()



### SUM - суммирование числовых значений

Функция SUM вычисляет сумму числовых значений в столбце.

Пример вычисления суммы возрастов всех пользователей:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Вычисление суммы возрастов пользователей

cursor.execute('SELECT SUM(age) FROM Users')

total\_age = cursor.fetchone()[0]

print('Общая сумма возрастов пользователей:', total\_age)

connection.close()



### AVG - вычисление среднего значения

Функция AVG вычисляет среднее значение числовых данных в столбце.

Пример вычисления среднего возраста пользователей:

import sqlite3

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Вычисление среднего возраста пользователей

cursor.execute('SELECT AVG(age) FROM Users')

average\_age = cursor.fetchone()[0]

print('Средний возраст пользователей:', average\_age)

connection.close()



Для выполнения сложных запросов можно использовать объединение таблиц и подзапросы. Например, давайте найдем пользователей с наибольшим возрастом:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Находим пользователей с наибольшим возрастом

cursor.execute('''

SELECT username, age

FROM Users

WHERE age = (SELECT MAX(age) FROM Users)

''')

oldest\_users = cursor.fetchall()

# Выводим результаты

for user in oldest\_users:

print(user)

# Закрываем соединение

connection.close()



### Обработка результатов

#### Получение результатов запроса в виде списка кортежей

Результаты запросов обычно возвращаются в виде списка кортежей. Каждый кортеж представляет собой строку данных. Давайте выведем результаты запроса на выборку всех пользователей:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем всех пользователей

cursor.execute('SELECT \* FROM Users')

users = cursor.fetchall()

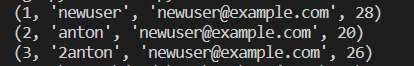
# Выводим результаты

for user in users:

print(user)

# Закрываем соединение

connection.close()



#### Использование методов fetchone(), fetchmany() и fetchall() для получения данных

Кроме того, вы можете использовать методы fetchone(), fetchmany() и fetchall() для получения данных по одной строке, нескольким строкам или всем строкам соответственно. Давайте рассмотрим примеры:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем первого пользователя

cursor.execute('SELECT \* FROM Users')

first\_user = cursor.fetchone()

print(first\_user)

# Выбираем первых 5 пользователей

cursor.execute('SELECT \* FROM Users')

first\_five\_users = cursor.fetchmany(5)

print(first\_five\_users)

# Выбираем всех пользователей

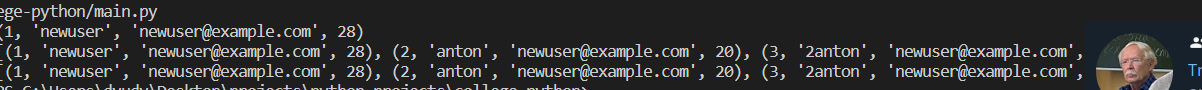
cursor.execute('SELECT \* FROM Users')

all\_users = cursor.fetchall()

print(all\_users)

# Закрываем соединение

connection.close()



#### Преобразование результатов в более удобные структуры данных (списки, словари)

Для удобства обработки данных, вы можете преобразовать результаты запросов в более удобные структуры данных, такие как списки или словари. Давайте преобразуем результаты запроса на выборку всех пользователей в список словарей:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Выбираем всех пользователей

cursor.execute('SELECT \* FROM Users')

users = cursor.fetchall()

# Преобразуем результаты в список словарей

users\_list = []

for user in users:

user\_dict = {

'id': user[0],

'username': user[1],

'email': user[2],

'age': user[3]

}

users\_list.append(user\_dict)

# Выводим результаты

for user in users\_list:

print(user)

# Закрываем соединение

connection.close()



### Транзакции и управление данными

#### Объяснение понятия транзакций и их важности

Транзакции - это группы операций, выполняемых как единое целое. Они обеспечивают надежность и целостность данных, гарантируя, что либо все операции будут выполнены успешно, либо ни одна из них не будет применена.

#### Использование операторов BEGIN, COMMIT и ROLLBACK

Операторы BEGIN, COMMIT и ROLLBACK позволяют управлять транзакциями в SQLite. Оператор BEGIN начинает транзакцию, COMMIT подтверждает изменения, а ROLLBACK отменяет транзакцию. Рассмотрим пример:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

try:

# Начинаем транзакцию

cursor.execute('BEGIN')

# Выполняем операции

cursor.execute('INSERT INTO Users (username, email) VALUES (?, ?)', ('user1', 'user1@example.com'))

cursor.execute('INSERT INTO Users (username, email) VALUES (?, ?)', ('user2', 'user2@example.com'))

# Подтверждаем изменения

cursor.execute('COMMIT')

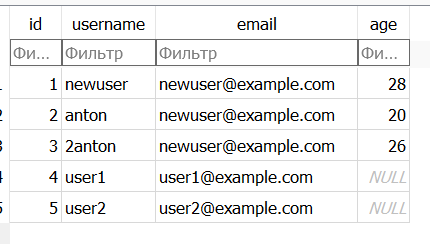
except:

# Отменяем транзакцию в случае ошибки

cursor.execute('ROLLBACK')

# Закрываем соединение

connection.close()



#### Автоматическое управление транзакциями с помощью контекстных менеджеров

Python предоставляет контекстные менеджеры, которые автоматически управляют транзакциями. Это обеспечивает более безопасное и читаемое управление данными. Давайте рассмотрим пример:

import sqlite3

# Устанавливаем соединение с базой данных

with sqlite3.connect('my\_database.db') as connection:

cursor = connection.cursor()

try:

# Начинаем транзакцию автоматически

with connection:

# Выполняем операции

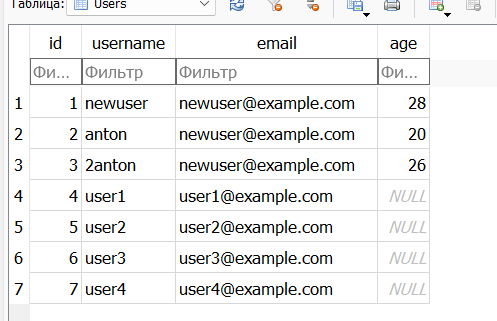
cursor.execute('INSERT INTO Users (username, email) VALUES (?, ?)', ('user3', 'user3@example.com'))

cursor.execute('INSERT INTO Users (username, email) VALUES (?, ?)', ('user4', 'user4@example.com'))

except:

# Ошибки будут приводить к автоматическому откату транзакции

Pass



#### Роли ACID (Atomicity, Consistency, Isolation, Durability) в транзакционных операциях

Транзакционные операции следуют принципам ACID:

* **Atomicity (Атомарность)**: Транзакция либо выполняется полностью, либо не выполняется совсем.
* **Consistency (Согласованность)**: Транзакция переводит базу данных из одного согласованного состояния в другое.
* **Isolation (Изолированность)**: Транзакции выполняются независимо друг от друга, как если бы они выполнялись последовательно.
* **Durability (Долговечность)**: После завершения транзакции изменения сохраняются даже при сбое системы.

### Продвинутые концепции

#### Использование подготовленных (prepared) запросов для повышения производительности

Подготовленные запросы позволяют многократно выполнять SQL-запросы с разными параметрами, что повышает производительность. Давайте рассмотрим пример:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Создаем подготовленный запрос

query = 'SELECT \* FROM Users WHERE age > ?'

cursor.execute(query, (25,))

users = cursor.fetchall()

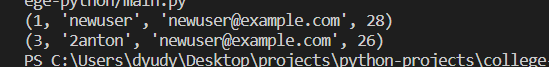
# Выводим результаты

for user in users:

print(user)

# Закрываем соединение

connection.close()



#### Работа с представлениями (views) для упрощения сложных запросов

Представления позволяют создавать виртуальные таблицы, которые являются результатом выполнения SQL-запроса. Это упрощает выполнение сложных запросов. Давайте создадим представление для выбора активных пользователей:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Создаем представление для активных пользователей

cursor.execute('CREATE VIEW ActiveUsers AS SELECT \* FROM Users WHERE is\_active = 1')

# Выбираем активных пользователей

cursor.execute('SELECT \* FROM ActiveUsers')

active\_users = cursor.fetchall()

# Выводим результаты

for user in active\_users:

print(user)

# Закрываем соединение

connection.close()



#### Создание триггеров для автоматизации операций при изменении данных

Триггеры - это специальные хранимые процедуры, которые автоматически вызываются при изменении данных в таблице. Давайте создадим триггер для автоматического обновления времени создания пользователя:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

# Создаем таблицу Users

cursor.execute('''

CREATE TABLE IF NOT EXISTS Users (

id INTEGER PRIMARY KEY,

username TEXT NOT NULL,

email TEXT NOT NULL,

age INTEGER,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

''')

# Создаем триггер для обновления времени создания при вставке новой записи

cursor.execute('''

CREATE TRIGGER IF NOT EXISTS update\_created\_at

AFTER INSERT ON Users

BEGIN

UPDATE Users SET created\_at = CURRENT\_TIMESTAMP WHERE id = NEW.id;

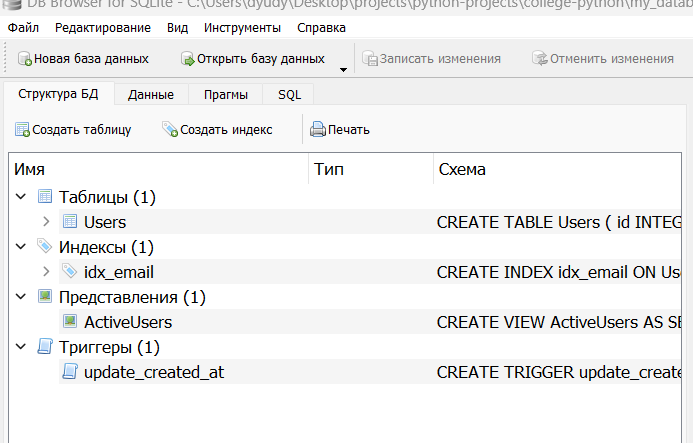
END;

''')

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



#### Работа с индексами для оптимизации запросов

Индексы позволяют ускорить выполнение запросов к базе данных. Давайте создадим индекс для ускорения поиска пользователей по имени:

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('my\_database.db')

cursor = connection.cursor()

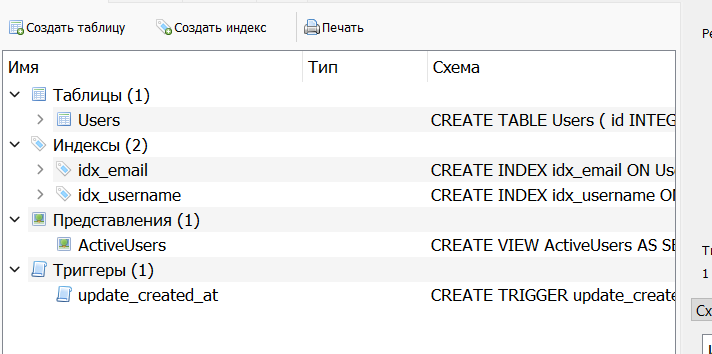
# Создаем индекс для столбца "username"

cursor.execute('CREATE INDEX idx\_username ON Users (username)')

# Сохраняем изменения и закрываем соединение

connection.commit()

connection.close()



### Примеры использования

#### Создание простого приложения для управления задачами с использованием SQLite и Python

Давайте рассмотрим пример создания простого приложения для управления задачами с использованием базы данных SQLite и языка программирования Python. В приложении мы будем хранить список задач с их статусами.

import sqlite3

# Устанавливаем соединение с базой данных

connection = sqlite3.connect('tasks.db')

cursor = connection.cursor()

# Создаем таблицу Tasks

cursor.execute('''

CREATE TABLE IF NOT EXISTS Tasks (

id INTEGER PRIMARY KEY,

title TEXT NOT NULL,

status TEXT DEFAULT 'Not Started'

)

''')

# Функция для добавления новой задачи

def add\_task(title):

cursor.execute('INSERT INTO Tasks (title) VALUES (?)', (title,))

connection.commit()

# Функция для обновления статуса задачи

def update\_task\_status(task\_id, status):

cursor.execute('UPDATE Tasks SET status = ? WHERE id = ?', (status, task\_id))

connection.commit()

# Функция для вывода списка задач

def list\_tasks():

cursor.execute('SELECT \* FROM Tasks')

tasks = cursor.fetchall()

for task in tasks:

print(task)

# Добавляем задачи

add\_task('Подготовить презентацию')

add\_task('Закончить отчет')

add\_task('Подготовить ужин')

# Обновляем статус задачи

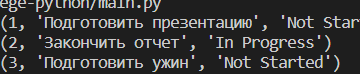
update\_task\_status(2, 'In Progress')

# Выводим список задач

list\_tasks()

# Закрываем соединение

connection.close()



### Заключение

В этом туториале мы рассмотрели основы работы с базой данных SQLite в языке программирования Python. Вы узнали, как создавать и управлять таблицами, выполнять запросы, использовать транзакции и применять продвинутые концепции для оптимизации и упрощения работы с данными. SQLite предоставляет мощные инструменты для управления данными внутри ваших приложений, делая их более эффективными и надежными.