Материалы к занятию

На прошлом занятии мы с вами выгрузили нашего бота на сервис Heroku.
На этом можно было бы остановиться, эхо-бот готов, но в реальном проекте нам понадобится сохранять различные данные из приложения. Для этого нужна база данных.Мы можем воспользоваться стандартной sqlite, но так как мы используем асинхронную библиотеку, то и запросы в бд должны быть асинхронными. Поэтому устанавливаем библиотеку databases для sqlite:
pip install databases[sqlite].

Пока что просто пишем не в даваясь в подробности sqlite. Ее мы детально разберем на следующих занятиях
.
Разобьём код по модулям и подключимся к базе данных: создаём файл config.py и выносим туда все переменные (WEBHOOK_HOST, WEBHOOK_PATH и т.д.).

И ещё один модуль “db.py”, в котором пишем следующий код:

```
from databases import Database
database = Database('sqlite:///bot.db')
```

Создадим таблицу:

```
CREATE TABLE messages (
id INTEGER PRIMARY KEY AUTOINCREMENT,
telegram_id INTEGER NOT NULL,
text text NOT NULL
);
```

И дополняем модуль основной файл:

```
import logging

from aiogram import Bot, types
from aiogram.contrib.middlewares.logging import LoggingMiddleware
from aiogram.dispatcher import Dispatcher
from aiogram.dispatcher.webhook import SendMessage
from aiogram.utils.executor import start_webhook
from db import database
from config import *
```

```
logging.basicConfig(level=logging.INFO)

bot = Bot(token=API_TOKEN)
dp = Dispatcher(bot)
dp.middleware.setup(LoggingMiddleware())


async def save(user_id, text):
   await database.execute(f"INSERT INTO messages(telegram_id, text) "
                          f"VALUES (:telegram_id, :text)",
values={'telegram_id': user_id, 'text': text})


async def read(user_id):
   messages = await database.fetch_all('SELECT text '
                                       'FROM messages '
                                       'WHERE telegram_id = :telegram_id ',
                                       values={'telegram_id': user_id})
   return messages


@dp.message_handler()
async def echo(message: types.Message):
   # Regular request
   # await bot.send_message(message.chat.id, message.text)

   # or reply INTO webhook
   return SendMessage(message.chat.id, message.text)


async def on_startup(dp):
   await bot.set_webhook(WEBHOOK_URL)
   # insert code here to run it after start


async def on_shutdown(dp):
   logging.warning('Shutting down..')

   # insert code here to run it before shutdown

   # Remove webhook (not acceptable in some cases)
   await bot.delete_webhook()

   # Close DB connection (if used)
   await dp.storage.close()
   await dp.storage.wait_closed()

   logging.warning('Bye!')


if __name__ == '__main__':
   start_webhook(
       dispatcher=dp,
       webhook_path=WEBHOOK_PATH,
       on_startup=on_startup,
       on_shutdown=on_shutdown,
       skip_updates=True,
       host=WEBAPP_HOST,
```

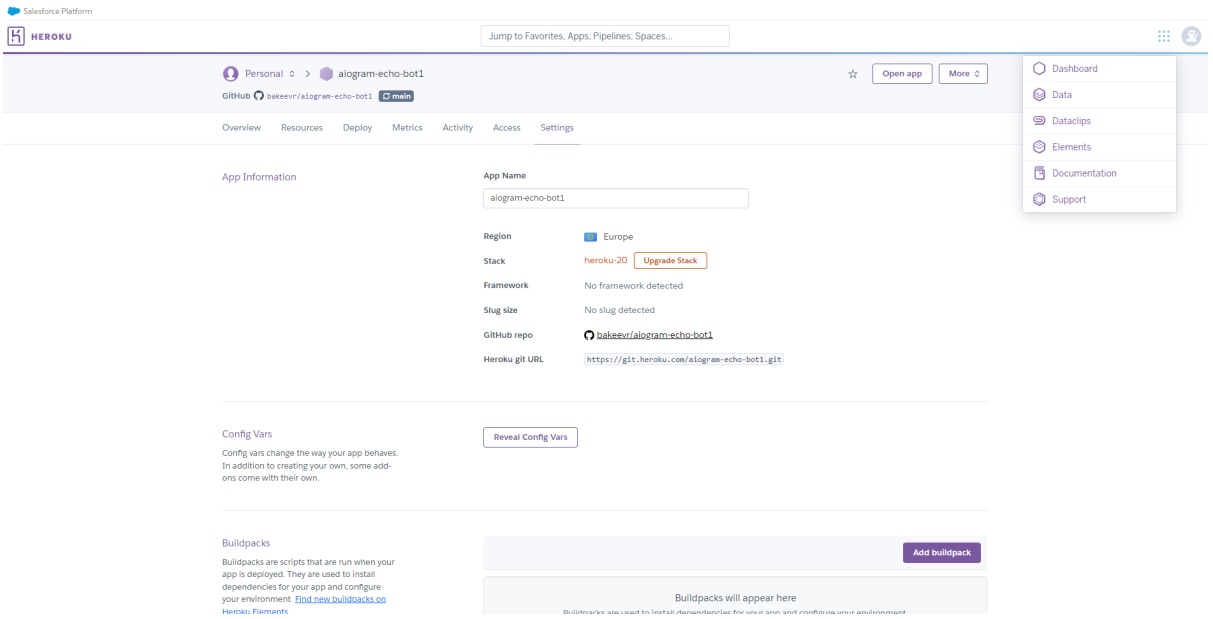
```
        port=WEBAPP_PORT,
    )
```

Здесь мы после получения сообщения сохраняем его в базу данных, и затем просто возвращаем все сообщения, полученные от этого пользователя.
Не забываем обновить requirements.txt

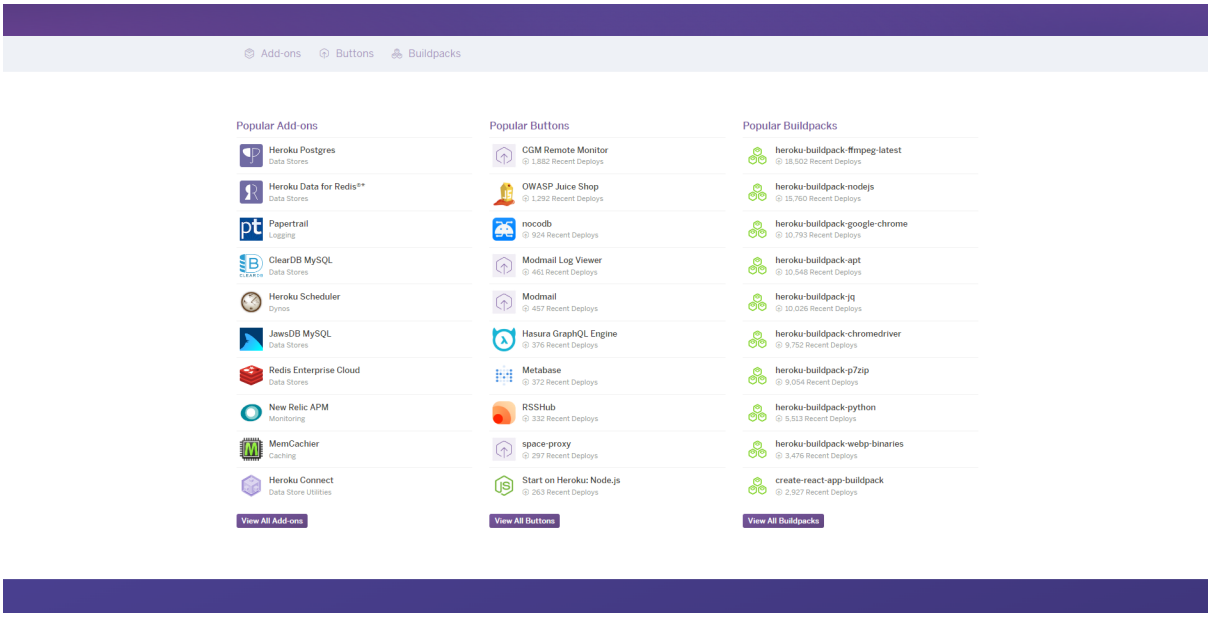
pip freeze > requirements.txt

Проверяем в боте: отправляем пару сообщений, бот возвращает нам список сохранённых в базу.
Казалось бы, всё хорошо, но вдруг произошла непредвиденная ошибка и приложение необходимо перезапустить: зайдём на вкладку «Resources»
Нажимаем на карандаш, жмём переключатель, для выключения приложения и подтверждаем «confirm».

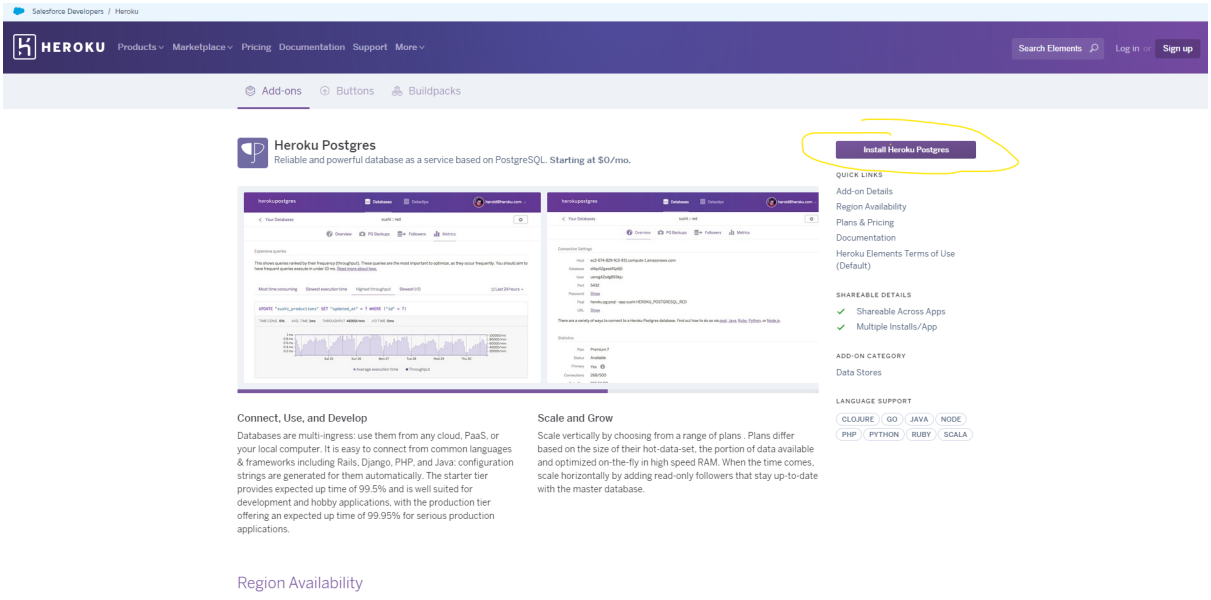
Вновь включаем таким же способом и пробуем отправить боту сообщения. Мы потеряли все данные! Но почему, ведь они хранятся в базе данных? Это происходит потому, что деплой происходит в изолированных контейнерах и при каждом новом запуске создаётся новый контейнер, а как мы помним исходный файл с бд у нас был пустым.
В нашем случае данные нужны будут и после выключения, поэтому нам нужна изолированная от приложения база данных. К счастью на Heroku, помимо множества приложений, можно бесплатно развернуть и базу данных, например postgres.
Переходим в «elements»



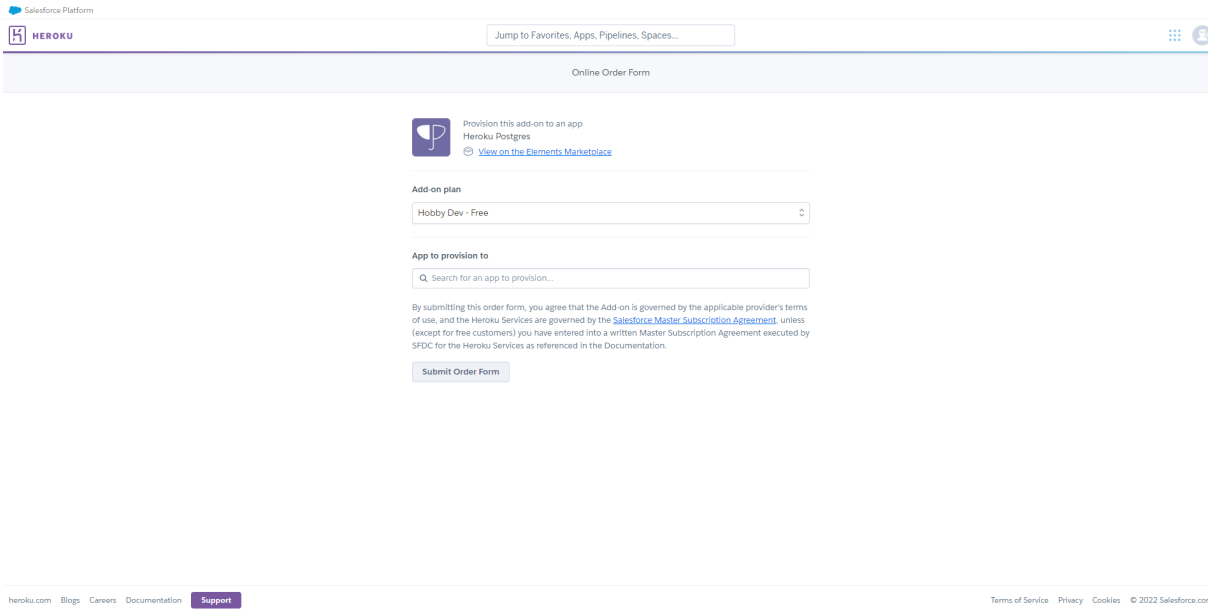
Выбираем «Heroku Postgres»



И устанавливаем



Выбираем бесплатный план и вводим имя приложения, для которого подключаем бд, для того чтобы потом мы могли считывать строку подключения с переменных среды:



Переходим в переменные среды нашего приложения и видим, что там появился ключ «DATABASE_URL», который мы и будем использовать для подключения.

Для подключения к бд postgresql, установим пакет databases[postgresql]: pip install databases[postgresql]. Создаём исходные таблицы, но синтаксис создания таблицы немного поменяется:

```
CREATE TABLE messages (
id SERIAL PRIMARY KEY,
telegram_id INTEGER NOT NULL,
text text NOT NULL
);
```

```
async def read(user_id):
    results = await database.fetch_all('SELECT text '
                                       'FROM messages '
                                       'WHERE telegram_id = :telegram_id ',
                                       values={'telegram_id': user_id})
    return [next(result.values()) for result in results]
```

Вновь обновляем requirements.txt и отправляем на гит.
Дожидаемся окончания деплоя, если приложение не запущено, то запускаем его и отправляем проверочные сообщения боту.
Получаем сообщения, всё отлично! Если же произойдет непредвиденная ошибка, то все данные сохраняться в БД

База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. База данных обычно управляется системой управления базами данных (СУБД). Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных, или, для краткости, просто базой данных.
Данные в наиболее распространенных типах современных баз данных обычно хранятся в виде строк и столбцов формирующих таблицу. Этими данными можно легко управлять, изменять, обновлять, контролировать и упорядочивать. В большинстве баз данных для записи и запросов данных используется язык структурированных запросов (SQL).
SQL — это язык программирования, используемый в большинстве реляционных баз данных для запросов, обработки и определения данных, а также контроля доступа. SQL был разработан в IBM в 1970-х годах. Со временем у стандарта SQL ANSI появились многочисленные расширения, разработанные такими компаниями как IBM, Oracle и Microsoft. Хотя в настоящее время SQL все еще широко используется, начали появляться новые языки программирования запросов.

Базы данных и электронные таблицы (в частности, Microsoft Excel) предоставляют удобные способы хранения информации. Основные различия между ними заключаются в следующем.

- Способ хранения и обработки данных
- Полномочия доступа к данным
- Объем хранения данных

Электронные таблицы изначально разрабатывались для одного пользователя, и их свойства отражают это. Они отлично подходят для одного пользователя или небольшого числа пользователей, которым не нужно производить сложные операции с данными. С другой стороны, базы данных предназначены для хранения гораздо больших наборов упорядоченной информации—иногда огромных объемов. Базы данных дают возможность множеству пользователей в одно и то же время быстро и безопасно получать доступ к данным и запрашивать их, используя развитую логику и язык запросов.

Существует множество различных типов баз данных. Выбор наилучшей базы данных для конкретной компании зависит от того, как она намеревается использовать данные.

- Реляционные базы данных Реляционные базы данных стали преобладать в 1980-х годах. Данные в реляционной базе организованы в виде таблиц, состоящих из столбцов и строк. Реляционная СУБД обеспечивает быстрый и эффективный доступ к структурированной информации.
- Объектно-ориентированные базы данных Информация в объектно-ориентированной базе данных представлена в форме объекта, как в объектно-ориентированном программировании.
- Распределенные базы данных Распределенная база данных состоит из двух или более частей, расположенных на разных серверах. Такая база данных может храниться на нескольких компьютерах.

- Хранилища данных Будучи централизованным репозиторием для данных, хранилище данных представляет собой тип базы данных, специально предназначенной для быстрого выполнения запросов и анализа.
- Базы данных NoSQL База данных NoSQL, или нереляционная база данных, дает возможность хранить и обрабатывать неструктурированные или слабоструктурированные данные (в отличие от реляционной базы данных, задающей структуру содержащихся в ней данных). Популярность баз данных NoSQL растет по мере распространения и усложнения веб-приложений.
- Графовые базы данных Графовая база данных хранит данные в контексте сущностей и связей между сущностями.
- Базы данных OLTP. База данных OLTP — это база данных предназначенная для выполнения бизнес-транзакций, выполняемых множеством пользователей.

Это лишь некоторые из десятков типов баз данных, используемых в настоящее время. Другие, менее распространенные базы данных, предназначены для очень специфических научных, финансовых и иных задач.

В данном случае мы будем знакомиться с SQL базами данных, а непосредственно SQLite. Данная база данных не требует установки на компьютер и взаимодействует с Python с помощью модуля sqlite3.

Модуль sqlite3 реализует интерфейс доступа к внутрипроцессной базе данных SQLite. База данных SQLite спроектирована таким образом, чтобы ее можно было встраивать в приложения, а не использовать отдельную серверную программу, такую как MySQL, PostgreSQL или Oracle.
SQLite — быстрая, тщательно протестированная и гибкая база данных, что делает ее весьма удобной для прототипирования и производственного развертывания в случае некоторых приложений.

База данных SQLite хранится в файловой системе в виде одиночного файла. Библиотека управляет доступом к этому файлу, включая его блокирование с целью предотвращения повреждения данных, когда одновременно несколько программ пытаются записать данные в файл. База данных создается при первой попытке доступа к файлу, но ответственность за создание таблицы определений, или схемы базы данных, возлагается на приложение.
В этом примере программа сначала осуществляет поиск файла базы данных, прежде чем открыть его с помощью функции connect (), поэтому ей известно, в каких случаях следует создавать схему для новых баз данных.

```
import os
import sqlite3
db_filename = 'todo.db'
db_is_new = not os.path.exists(db_filename)
conn = sqlite3.connect(db_filename)
if db_is_new:
```

```
   print('Создана новая база')
else:
   print('База данных уже существует.')
conn.close()
```

Выполнение этого кода два раза подряд показывает, что в том случае, когда файл не существует, создается пустой файл.

После создания нового файла базы данных следующим шагом является создание схемы для определения таблиц в базе данных.

```
import os
import sqlite3
db_filename = 'todo.db'
schema_filename = 'todo_schema.sql'
db_is_new = not os.path.exists(db_filename)
with sqlite3.connect(db_filename) as conn:
   if db_is_new:
       print('Creating schema')
       with open(schema_filename, 'rt') as f:
           schema = f.read()
       conn.executescript(schema)
       print('Inserting initial data')
       conn.executescript("""
       insert into project (name, description, deadline)
       values ('pymotw', 'Python Module of the Week',
       '2022-11-01');
       insert into task (details, status, deadline, project)
       values ('write about select', 'done', '2022-04-25',
       'pymotw');
       insert into task (details, status, deadline, project)
       values ('write about random', 'waiting', '2022—08—22',
       'pymotw');
       insert into task (details, status, deadline, project)
       values ('write about sqlite3', 'active', '2021—07—31',
       'pymotw');
       """)
   else:
       print('Database exists, assume schema does, too.')
```

Вслед за созданием таблиц выполняются инструкции вставки, с помощью которых создается пробный проект и относящиеся к нему задачи.

SQLite для Python предлагает меньше типов данных, чем есть в других реализациях SQL. С одной стороны, это накладывает ограничения, но, с другой стороны, в SQLite многое сделано проще. Вот основные типы:

- NULL — значение NULL
- INTEGER — целое число
- REAL — число с плавающей точкой
- TEXT — текст

- BLOB — бинарное представление крупных объектов, хранящееся в точности с тем, как его ввели

Разберем создание базы данных и таблицы подробнее. Есть несколько способов создания базы данных в Python с помощью SQLite. Для этого используется объект Connection, который и представляет собой базу. Он создается с помощью функции connect().
Создадим файл .db, поскольку это стандартный способ управления базой SQLite. Файл будет называться test.db. За соединение будет отвечать переменная conn.

```
conn = sqlite3.connect('test.db')
```

Эта строка создает объект connection, а также новый файл orders.db в рабочей директории. Если нужно использовать другую, ее нужно обозначить явно:

```
conn = sqlite3.connect('ПУТЬ-К-ПАПКИ/test.db')
```

Если файл уже существует, то функция connect осуществит подключение к нему. Функция connect создает соединение с базой данных SQLite и возвращает объект, представляющий ее.

Еще один способ создания баз данных с помощью SQLite в Python — создание их в памяти. Это отличный вариант для тестирования, ведь такие базы существуют только в оперативной памяти.

```
conn = sqlite3.connect(:memory:)
```

После создания объекта соединения с базой данных нужно создать объект cursor. Он позволяет делать SQL-запросы к базе. Используем переменную cur для хранения объекта:

```
cur = conn.cursor()
```

Теперь выполнять запросы можно следующим образом:

```
cur.execute("ВАШ-SQL-ЗАПРОС-ЗДЕСЬ;")
```

Обратите внимание на то, что сами запросы должны быть помещены в кавычки — это важно. Это могут быть одинарные, двойные или тройные кавычки. Последние используются в случае особенно длинных запросов, которые часто пишутся на нескольких строках.

Давайте разберем как создать таблицу, в которой у нас есть имя фамилия пол, ну и порядковый номер. Для этого необходимо создать запрос

```
cur.execute("""CREATE TABLE IF NOT EXISTS users(
   userid INT PRIMARY KEY,
   fname TEXT,
   lname TEXT,
   gender TEXT);
""")
conn.commit()
```

В коде выполняются следующие операции:

1. Функция execute отвечает за SQL-запрос
2. SQL генерирует таблицу users
3. IF NOT EXISTS поможет при попытке повторного подключения к базе данных. Запрос проверит, существует ли таблица. Если да — проверит, ничего ли не поменялось.
4. Создаем первые четыре колонки: userid, fname, lname и gender. Userid — это основной ключ.
5. Сохраняем изменения с помощью функции commit для объекта соединения.

Создадим еще одну таблицу

```
cur.execute("""CREATE TABLE IF NOT EXISTS orders(
   orderid INT PRIMARY KEY,
   date TEXT,
   userid TEXT,
   total TEXT);
""")
conn.commit()
```

Полный код

```
import os
import sqlite3


conn = sqlite3.connect('todo.db')
cur = conn.cursor()
cur.execute("""CREATE TABLE IF NOT EXISTS users(
  userid INT PRIMARY KEY,
  fname TEXT,
  lname TEXT,
  gender TEXT);
""")
cur.execute("""CREATE TABLE IF NOT EXISTS orders(
  orderid INT PRIMARY KEY,
  date TEXT,
  userid TEXT,
  total TEXT);
""")
```

```
conn.commit()
```

После исполнения этих двух скриптов база данных будет включать две таблицы. Теперь можно добавлять данные.

Давайте создадим новую базу данных и поработаем с ней

```
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.
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 создается с использованием объекта подключения
3. Используя объект курсора, вызывается метод execute с запросом create table в качестве параметра.

Давайте создадим таблицу Employees со следующими колонками:

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

```
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.
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 кортеж entities содержат значения для заполнения одной строки в таблице:
entity = (2, 'Andrew', 800, 'IT', 'Tech', '2018-02-06')

```
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)
```

Учитель: 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.

```
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 Эндрю на Роджерс.
Оператор SELECT используется для выборки данных из одной или более таблиц. Если нужно выбрать все столбцы данных из таблицы, можете использовать звездочку (*). 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.

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.

```
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 100:

```
import sqlite3

con = sqlite3.connect('mydatabase.db')


def sql_fetch(con):
   cursorObj = con.cursor()

   cursorObj.execute('SELECT id, name FROM employees WHERE salary > 100.0')
```

```
   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.

Счетчик 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.

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)
```

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 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)
```