PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

РАБОТА С БАЗАМИ ДАННЫХ

РАБОТА С БАЗАМИ ДАННЫХ

База данных (БД) - это данные, которые хранятся в соответствии с определенной схемой. В этой схеме каким-то образом описаны соотношения между данными.

Язык БД (лингвистические средства) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Система управления базами данных (СУБД) - это программные средства, которые дают возможность управлять БД. СУБД должны поддерживать соответствующий язык (языки) для управления БД.

SQL

SQL (structured query language) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Язык SQL подразделяется на такие 4 категории:

- DDL (Data Definition Language) - язык описания данных
- DML (Data Manipulation Language) - язык манипулирования данными
- DCL (Data Control Language) - язык определения доступа к данным
- TCL (Transaction Control Language) - язык управления транзакциями

SQL

В каждой категории есть свои операторы (перечислены не все операторы):

- DDL
  - CREATE - создание новой таблицы, СУБД, схемы
  - ALTER - изменение существующей таблицы, колонки
  - DROP - удаление существующих объектов из СУБД
- DML
  - SELECT - выбор данных
  - INSERT - добавление новых данных
  - UPDATE - обновление существующих данных
  - DELETE - удаление данных

SQL

- DCL
  - GRANT - предоставление пользователям разрешения на чтение/запись определенных объектов в СУБД
  - REVOKE - отзывает ранее предоставленые разрешения
- TCL
  - COMMIT Transaction - применение транзакции
  - ROLLBACK Transaction - откат всех изменений сделанных в текущей транзакции

SQL И PYTHON

Для работы с реляционной СУБД в Python можно использовать два подхода:

- работать с библиотекой, которая соответствует конкретной СУБД и использовать для работы с БД язык SQL
  - Например, для работы с SQLite используется модуль sqlite3
- работать с ORM, которая использует объектно-ориентированный подход для работы с БД
  - Например, SQLAlchemy

SQLITE

SQLITE

SQLite — встраиваемая в процесс реализация SQL-машины.

На практике, SQLite часто используется как встроенная СУБД в приложениях.

SQLITE CLI

В комплекте поставки SQLite идёт также утилита для работы с SQLite в командной строке. Утилита представлена в виде исполняемого файла sqlite3 (sqlite3.exe для Windows) и с ее помощью можно вручную выполнять команды SQL.

В помощью этой утилиты очень удобно проверять правильность команд SQL, а также в целом знакомится с языком SQL.

SQLITE CLI

Для того чтобы создать БД (или открыть уже созданную) надо просто вызвать sqlite3 таким образом:

```
$ sqlite3 testDB.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite>
```

Внутри sqlite3 можно выполнять команды SQL или, так называемые, метакоманды (или dot-команды).

МЕТАКОМАНДЫ

К метакомандам относятся несколько специальных команд, для работы с SQLite. Они относятся только к утилите sqlite3, а не к SQL языку. В конце этих команд ; ставить не нужно.

Примеры метакоманд:

- .help - подсказка со списком всех метакоманд
- .exit или .quit - выход из сессии sqlite3
- .databases - показывает присоединенные БД
- .tables - показывает доступные таблицы

МЕТАКОМАНДЫ

Примеры выполнения:

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
...

sqlite> .databases
seq  name      file
---  --------  ----------------------------------
0    main      /home/nata/py_for_ne/db/db_article/testDB.db
```

ОСНОВЫ SQL (В SQLITE3 CLI)

CREATE

Оператор create позволяет создавать таблицы.

Создадим таблицу switch, в которой хранится информация о коммутаторах:

```
sqlite> CREATE table switch (
   ...>     mac          text not NULL primary key,
   ...>     hostname     text,
   ...>     model        text,
   ...>     location     text
   ...> );
```

CREATE

Аналогично можно было создать таблицу и таким образом:

```
sqlite> create table switch (mac text not NULL primary key, hostname text, model text, location text);
```

Поле mac является первичным ключом. Это автоматически значит, что:

- поле должно быть уникальным
- в нем не может находиться значение NULL

В этом примере это вполне логично, так как MAC-адрес должен быть уникальным.

CREATE

На данный момент записей в таблице нет, есть только ее определение. Просмотреть определение можно такой командой:

```
sqlite> .schema switch
CREATE TABLE switch (
mac           text not NULL primary key,
hostname      text,
model         text,
location      text
);
```

DROP

Оператор drop удаляет таблицу вместе со схемой и всеми данными.

Удалить таблицу можно так:

```
sqlite> DROP table switch;
```

INSERT

Оператор insert используется для добавления данных в таблицу.

Если указываются значения для всех полей, добавить запись можно таким образом (порядок полей должен соблюдаться):

```
sqlite> INSERT into switch values ('0010.A1AA.C1CC', 'sw1', 'Cisco 3750', 'London, Green Str');
```

Если нужно указать не все поля, или указать их в произвольном порядке, используется такая запись:

```
sqlite> INSERT into switch (mac, model, location, hostname)
   ...> values ('0020.A2AA.C2CC', 'Cisco 3850', 'London, Green Str', 'sw2');
```

SELECT

Оператор select позволяет запрашивать информацию в таблице.

Например:

```
sqlite> SELECT * from switch;
0010.A1AA.C1CC|sw1|Cisco 3750|London, Green Str
0020.A2AA.C2CC|sw2|Cisco 3850|London, Green Str
```

SELECT

Включить отображение названия полей можно с помощью команды .headers ON.

```
sqlite> .headers ON
sqlite> SELECT * from switch;
mac|hostname|model|location
0010.A1AA.C1CC|sw1|Cisco 3750|London, Green Str
0020.A2AA.C2CC|sw2|Cisco 3850|London, Green Str
```

SELECT

За форматирование вывода отвечает команда .mode.

Режим .mode column включает отображение в виде колонок:

```
sqlite> .mode column
sqlite> SELECT * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
```

МЕТАКОМАНДА .READ

В таблице switch всего две записи:

```
sqlite> SELECT * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
```

МЕТАКОМАНДА .READ

Метакоманда .read позволяет загружать команды SQL из файла.

Для добавления записей, заготовлен файл add_rows_to_testdb.txt:

```
INSERT into switch values ('0030.A3AA.C1CC', 'sw3', 'Cisco 3750', 'London, Green Str');
INSERT into switch values ('0040.A4AA.C2CC', 'sw4', 'Cisco 3850', 'London, Green Str');
INSERT into switch values ('0050.A5AA.C3CC', 'sw5', 'Cisco 3850', 'London, Green Str');
INSERT into switch values ('0060.A6AA.C4CC', 'sw6', 'C3750', 'London, Green Str');
INSERT into switch values ('0070.A7AA.C5CC', 'sw7', 'Cisco 3650', 'London, Green Str');
```

МЕТАКОМАНДА .READ

Для загрузки команд из файла, надо выполнить команду:

```
sqlite> .read add_rows_to_testdb.txt
```

МЕТАКОМАНДА .READ

Теперь таблица switch выглядит так:

```
sqlite> SELECT * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

WHERE

Оператор WHERE используется для уточнения запроса. С помощью этого оператора можно указывать определенные условия, по которым отбираются данные. Если условие выполнено, возвращается соответствующее значение из таблицы, если нет, не возвращается.

WHERE

Таблица switch выглядит так:

```
sqlite> SELECT * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

WHERE

Показать только те коммутаторы, модель которых 3850:

```
sqlite> SELECT * from switch WHERE model = 'Cisco 3850';
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
```

WHERE

Оператор WHERE позволяет указывать не только конкретное значение поля. Если добавить к нему оператор like, можно указывать шаблон поля.

LIKE с помощью символов _ и % указывает на что должно быть похоже значение:

- _ - обозначает один символ или число
- % - обозначает ноль, один или много символов

Например, если поле model записано в разном формате, с помощью предыдущего запроса, не получится вывести нужные коммутаторы.

WHERE

Например, у коммутатора sw6 поле model записано в таком формате C3750, а у коммутаторов sw1 и sw3 в таком Cisco 3750.

В таком варианте запрос с оператором WHERE не покажет sw6:

```
sqlite> SELECT * from switch WHERE model = 'Cisco 3750';
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
```

WHERE

Но, если вместе с оператором WHERE использовать оператор LIKE:

```
sqlite> SELECT * from switch WHERE model LIKE '%3750';
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
```

ALTER

Оператор ALTER позволяет менять существующую таблицу: добавлять новые колонки или переименовывать таблицу.

Добавим в таблицу новые поля:

- mngmt_ip - IP-адрес коммутатора в менеджмент VLAN
- mngmt_vid - VLAN ID (номер VLAN) для менеджмент VLAN

Добавление записей с помощью команды ALTER:

```
sqlite> ALTER table switch ADD COLUMN mngmt_ip text;
sqlite> ALTER table switch ADD COLUMN mngmt_vid integer;
```

ALTER

Теперь таблица выглядит так (новые поля установлены в значение NULL):

```
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

UPDATE

Оператор UPDATE используется для изменения существующей записи таблицы.

Обычно, UPDATE используется вместе с оператором where, чтобы уточнить какую именно запись необходимо изменить.

С помощью UPDATE, можно заполнить новые столбцы в таблице:

```
sqlite> UPDATE switch set mngmt_ip = '10.255.1.1' WHERE hostname = 'sw1';
```

UPDATE

Результат будет таким:

```
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str  10.255.1.1
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

UPDATE

Аналогичным образом можно изменить и номер VLAN:

```
sqlite> UPDATE switch set mngmt_vid = 255 WHERE hostname = 'sw1';
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

UPDATE

И можно изменить несколько полей за раз:

```
sqlite> UPDATE switch set mngmt_ip = '10.255.1.2', mngmt_vid = 255 WHERE hostname = 'sw2';
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str
0060.A6AA.C4CC  sw6         C3750       London, Green Str
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str
```

UPDATE

Чтобы не заполнять поля mngmt_ip и mngmt_vid вручную, заполним остальное из файла update_fields_in_testdb.txt:

```
UPDATE switch set mngmt_ip = '10.255.1.3', mngmt_vid = 255 WHERE hostname = 'sw3';
UPDATE switch set mngmt_ip = '10.255.1.4', mngmt_vid = 255 WHERE hostname = 'sw4';
UPDATE switch set mngmt_ip = '10.255.1.5', mngmt_vid = 255 WHERE hostname = 'sw5';
UPDATE switch set mngmt_ip = '10.255.1.6', mngmt_vid = 255 WHERE hostname = 'sw6';
UPDATE switch set mngmt_ip = '10.255.1.7', mngmt_vid = 255 WHERE hostname = 'sw7';
```

UPDATE

После загрузки команд, таблица выглядит так:

```
sqlite> .read update_fields_in_testdb.txt

sqlite> SELECT * from switch;
mac             hostname    model       location            mngmt_ip    mngmt_vid
--------------  ----------  ----------  ------------------  ----------  ----------
0010.A1AA.C1CC  sw1         Cisco 3750  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0030.A3AA.C1CC  sw3         Cisco 3750  London, Green Str  10.255.1.3  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
```

REPLACE

Оператор REPLACE используется для добавления или замены данных в таблице.

Когда возникает нарушение условия уникальности поля, выражение с оператором REPLACE:

- удаляет существующую строку, которая вызвала нарушение
- добавляет новую строку

REPLACE

У выражения REPLACE есть два вида:

```
sqlite> INSERT OR REPLACE INTO switch
   ...> VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London, Green Str', '10.255.1.3', 255);
```

Или более короткий вариант:

```
sqlite> REPLACE INTO switch
   ...> VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London, Green Str', '10.255.1.3', 255);
```

REPLACE

Результатом любой из этих команд, будет замена модели коммутатора sw3:

```
sqlite> SELECT * from switch;
mac             hostname    model       location            mngmt_ip    mngmt_vid
--------------  ----------  ----------  ------------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
```

REPLACE

При добавлении записи, для которой не возникает нарушения уникальности поля, replace работает как обычный insert:

```
sqlite> REPLACE INTO switch
   ...> VALUES ('0080.A8AA.C8CC', 'sw8', 'Cisco 3850', 'London, Green Str', '10.255.1.8', 255);

sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
0080.A8AA.C8CC  sw8         Cisco 3850  London, Green Str  10.255.1.8  255
```

DELETE

Оператор delete используется для удаления записей.

Как правило, он используется вместе с оператором where.

Например:

```
sqlite> DELETE from switch where hostname = 'sw8';
```

DELETE

Теперь в таблице нет строки с коммутатором sw:

```
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
```

ORDER BY

Оператор ORDER BY используется для сортировки вывода по определенному полю, по возрастанию или убыванию. Для этого он добавляется к оператору SELECT.

Если выполнить простой запрос SELECT, вывод будет таким:

```
sqlite> SELECT * from switch;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
```

ORDER BY

С помощью оператора ORDER BY можно вывести записи в таблице switch отсортированными их по имени коммутаторов:

```
sqlite> SELECT * from switch ORDER BY hostname ASC;
mac             hostname    model       location            mngmt_ip    mngmt_vid
--------------  ----------  ----------  ------------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
```

ORDER BY

По умолчанию сортировка выполняется по возрастанию, поэтому в запросе можно было не указывать параметр ASC:

```
sqlite> SELECT * from switch ORDER BY hostname;
mac             hostname    model       location           mngmt_ip    mngmt_vid
--------------  ----------  ----------  -----------------  ----------  ----------
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
```

ORDER BY

Сортировка по IP-адресу, по убыванию:

```
sqlite> SELECT * from switch ORDER BY mngmt_ip DESC;
mac             hostname    model       location            mngmt_ip    mngmt_vid
--------------  ----------  ----------  ------------------  ----------  ----------
0070.A7AA.C5CC  sw7         Cisco 3650  London, Green Str  10.255.1.7  255
0060.A6AA.C4CC  sw6         C3750       London, Green Str  10.255.1.6  255
0050.A5AA.C3CC  sw5         Cisco 3850  London, Green Str  10.255.1.5  255
0040.A4AA.C2CC  sw4         Cisco 3850  London, Green Str  10.255.1.4  255
0030.A3AA.C1CC  sw3         Cisco 3850  London, Green Str  10.255.1.3  255
0020.A2AA.C2CC  sw2         Cisco 3850  London, Green Str  10.255.1.2  255
0010.D1DD.E1EE  sw1         Cisco 3850  London, Green Str  10.255.1.1  255
```

МОДУЛЬ SQLITE3

МОДУЛЬ SQLITE3

Для работы с SQLite в Python используется модуль sqlite3.

Рассмотрим основные объекты и методы, которые модуль использует для работы с SQLite.

CONNECTION

Объект **Connection** - это подключение к конкретной БД. Можно сказать, что этот объект представляет БД.

```
import sqlite3

connection = sqlite3.connect('dhcp_snooping.db')
```

CURSOR

После создания соединения, надо создать объект Cursor - это основной способ работы с БД.

Создается курсор из соединения с БД:

```
import sqlite3

connection = sqlite3.connect('dhcp_snooping.db')
cursor = connection.cursor()
```

ВЫПОЛНЕНИЕ КОМАНД SQL

ВЫПОЛНЕНИЕ КОМАНД SQL

Для выполнения команд SQL в модуле есть несколько методов:

- **execute()** - метод для выполнения одного выражения SQL
- **executemany()** - метод позволяет выполнить одно выражение SQL для последовательности параметров (или для итератора)
- **executescript()** - метод позволяет выполнить несколько выражений SQL за один раз

МЕТОД EXECUTE

Метод execute позволяет выполнить одну команду SQL.

Сначала надо создать соединение и курсор:

```
In [1]: import sqlite3

In [2]: connection = sqlite3.connect('sw_inventory.db')

In [3]: cursor = connection.cursor()
```

МЕТОД EXECUTE

Создание таблицы switch с помощью метода execute:

```
In [4]: cursor.execute("create table switch (mac text not NULL primary key, hostname text, model text, locat
Out[4]: <sqlite3.Cursor at 0x1085be880>
```

МЕТОД EXECUTE

Выражения SQL могут быть параметризированы - вместо данных можно подставлять специальные значения. Засчет этого можно использовать одну и ту же команду SQL для передачи разных данных.

МЕТОД EXECUTE

Например, таблицу switch нужно заполнить данными из списка data:

```
In [5]: data = [
   ...: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
   ...: ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
   ...: ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

МЕТОД EXECUTE

Для этого можно использовать запрос вида:

```
In [6]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Знаки вопроса в команде используются для подстановки данных, которые будут передаваться методу execute.

МЕТОД EXECUTE

Теперь можно передать данные таким образом:

```
In [7]: for row in data:
   ...:     cursor.execute(query, row)
   ...:
```

Второй аргумент, который передается методу execute, должен быть кортежем. Если нужно передать кортеж с одним элементом, используется запись (value,).

МЕТОД EXECUTE

Чтобы изменения применились, нужно выполнить commit (обратите внимание, что метод commit вызывается у соединения):

```
In [8]: connection.commit()
```

МЕТОД EXECUTE

Теперь, при запросе из командной строки sqlite3, можно увидеть эти строки в таблице switch:

```
$ sqlite3 sw_inventory.db

sqlite> select * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0000.AAAA.CCCC  sw1         Cisco 3750  London, Green Str
0000.BBBB.CCCC  sw2         Cisco 3780  London, Green Str
0000.AAAA.DDDD  sw3         Cisco 2960  London, Green Str
0011.AAAA.CCCC  sw4         Cisco 3750  London, Green Str
```

МЕТОД EXECUTEMANY

Метод executemany позволяет выполнить одну команду SQL для последовательности параметров (или для итератора).

С помощью метода executemany, в таблицу switch можно добавить аналогичный список данных одной командой.

МЕТОД EXECUTEMANY

Например, в таблицу switch надо добавить данные из списка data2:

```
In [9]: data2 = [
   ...: ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

МЕТОД EXECUTEMANY

Для этого нужно использовать аналогичный запрос вида:

```
In [10]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Теперь можно передать данные методу executemany:

```
In [11]: cursor.executemany(query, data2)
Out[11]: <sqlite3.Cursor at 0x10ee5e810>

In [12]: connection.commit()
```

МЕТОД EXECUTEMANY

После выполнения commit, данные доступны в таблице:

```
sqlite> select * from switch;
mac             hostname    model       location
--------------  ----------  ----------  -----------------
0000.AAAA.CCCC  sw1         Cisco 3750  London, Green Str
0000.BBBB.CCCC  sw2         Cisco 3780  London, Green Str
0000.AAAA.DDDD  sw3         Cisco 2960  London, Green Str
0011.AAAA.CCCC  sw4         Cisco 3750  London, Green Str
0000.1111.0001  sw5         Cisco 3750  London, Green Str
0000.1111.0002  sw6         Cisco 3750  London, Green Str
0000.1111.0003  sw7         Cisco 3750  London, Green Str
0000.1111.0004  sw8         Cisco 3750  London, Green Str
```

МЕТОД EXECUTESCRIPT

Метод executescript позволяет выполнить несколько выражений SQL за один раз.

МЕТОД EXECUTESCRIPT

Особенно удобно использовать этот метод при создании таблиц:

```
In [14]: connection = sqlite3.connect('new_db.db')

In [15]: cursor = connection.cursor()

In [16]: cursor.executescript("""
    ...:     create table switches(
    ...:         hostname     text not NULL primary key,
    ...:         location     text
    ...:     );
    ...:
    ...:     create table dhcp(
    ...:         mac          text not NULL primary key,
    ...:         ip           text,
    ...:         vlan         text,
    ...:         interface    text,
    ...:         switch       text not null references switches(hostname)
    ...:     );
    ...: """)
Out[16]: <sqlite3.Cursor at 0x10efd67a0>
```

ПОЛУЧЕНИЕ РЕЗУЛЬТАТОВ ЗАПРОСА

ПОЛУЧЕНИЕ РЕЗУЛЬТАТОВ ЗАПРОСА

Для получения результатов запроса в sqlite3 есть несколько способов:

- использование методов fetch...() - в зависимости от метода возвращаются одна, несколько или все строки
- использование курсора как итератора - возвращается итератор

МЕТОД FETCHONE

Метод fetchone возвращает одну строку данных.

Пример получения информации из базы данных sw_inventory.db:

```
In [1]: import sqlite3

In [2]: connection = sqlite3.connect('sw_inventory.db')

In [3]: cursor = connection.cursor()

In [4]: cursor.execute('select * from switch')
Out[4]: <sqlite3.Cursor at 0x104eda810>

In [5]: cursor.fetchone()
Out[5]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
```

МЕТОД FETCHONE

Если повторно вызвать метод, он вернет следующую строку:

```
In [6]: print(cursor.fetchone())
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
```

МЕТОД FETCHONE

Аналогичным образом метод будет возвращать следующие строки. После обработки всех строк, метод начинает возвращать None.

МЕТОД FETCHONE

Засчет этого, метод можно использовать в цикле, например, так:

```
In [7]: cursor.execute('select * from switch')
Out[7]: <sqlite3.Cursor at 0x104eda810>

In [8]: while True:
   ...:     next_row = cursor.fetchone()
   ...:     if next_row:
   ...:         print(next_row)
   ...:     else:
   ...:         break
   ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
```

МЕТОД FETCHMANY

Метод fetchmany возвращает возвращает список строк данных.

Синтаксис метода:

```
cursor.fetchmany([size=cursor.arraysize])
```

МЕТОД FETCHMANY

С помощью параметра size, можно указывать какое количество строк возвращается. По умолчанию, параметр size равен значению cursor.arraysize:

```
In [9]: print(cursor.arraysize)
1
```

МЕТОД FETCHMANY

Например, таким образом можно возвращать по три строки из запроса:

```
In [25]: cursor.execute('select * from switch')
Out[25]: <sqlite3.Cursor at 0x104eda810>

In [26]: from pprint import pprint

In [27]: while True:
    ...:     three_rows = cursor.fetchmany(3)
    ...:     if three_rows:
    ...:         pprint(three_rows)
    ...:     else:
    ...:         break
    ...:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')]
[('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')]
[('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

МЕТОД FETCHALL

Метод fetchall возвращает все строки в виде списка:

```
In [12]: cursor.execute('select * from switch')
Out[12]: <sqlite3.Cursor at 0x104eda810>

In [13]: cursor.fetchall()
Out[13]:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

МЕТОД FETCHALL

Важный аспект работы метода - он возвращает все оставшиеся строки.

То есть, если до метода fetchall, использовался, например, метод fetchone, то метод fetchall вернет оставшиеся строки запроса:

```
In [30]: cursor.execute('select * from switch')
Out[30]: <sqlite3.Cursor at 0x104eda810>

In [31]: cursor.fetchone()
Out[31]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')

In [32]: cursor.fetchone()
Out[32]: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')

In [33]: cursor.fetchall()
Out[33]:
[('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

CURSOR КАК ИТЕРАТОР

CURSOR КАК ИТЕРАТОР

Если нужно построчно обрабатывать результирующие строки, лучше использовать курсор как итератор. При этом не нужно использовать методы fetch.

При использовании методов execute, возвращается курсор.

CURSOR КАК ИТЕРАТОР

```
In [34]: result = cursor.execute('select * from switch')

In [35]: for row in result:
    ...:     print(row)
    ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

ИСПОЛЬЗОВАНИЕ МОДУЛЯ SQLITE3 БЕЗ ЯВНОГО СОЗДАНИЯ КУРСОРА

ИСПОЛЬЗОВАНИЕ МОДУЛЯ SQLITE3 БЕЗ ЯВНОГО СОЗДАНИЯ КУРСОРА

Методы execute доступны и в объекте Connection. При их использовании курсор создается, но не явно. Однако, методы fetch в Connection недоступны.

Но, если использовать курсор, который возвращают методы execute, как итератор, методы fetch могут и не понадобиться.

ИСПОЛЬЗОВАНИЕ МОДУЛЯ SQLITE3 БЕЗ ЯВНОГО СОЗДАНИЯ КУРСОРА

Пример итогового скрипта (файл create_sw_inventory_ver1.py):

```
# -*- coding: utf-8 -*-
import sqlite3

data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]

con = sqlite3.connect('sw_inventory2.db')

con.execute("""create table not NULL switch
            (mac text primary key, hostname text, model text, location text)""")

query = "INSERT into switch values (?, ?, ?, ?)"
con.executemany(query, data)
con.commit()

for row in con.execute("select * from switch"):
    print(row)

con.close()
```

ИСПОЛЬЗОВАНИЕ МОДУЛЯ SQLITE3 БЕЗ ЯВНОГО СОЗДАНИЯ КУРСОРА

Результат выполнения будет таким:

```
$ python create_sw_inventory_ver1.py
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

ОБРАБОТКА ИСКЛЮЧЕНИЙ

В таблице switch поле mac должно быть уникальным. И, если попытаться записать пересекающийся MAC-адрес, возникнет ошибка:

```
In [37]: con = sqlite3.connect('sw_inventory2.db')

In [38]: query = "INSERT into switch values ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str')"

In [39]: con.execute(query)
---------------------------------------------------------------
IntegrityError              Traceback (most recent call last)
<ipython-input-56-ad34d83a8a84> in <module>()
----> 1 con.execute(query)

IntegrityError: UNIQUE constraint failed: switch.mac
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Соответственно, можно перехватить исключение:

```
In [40]: try:
    ...:     con.execute(query)
    ...: except sqlite3.IntegrityError as e:
    ...:     print("Error occured: ", e)
    ...:
Error occured:  UNIQUE constraint failed: switch.mac
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

После выполнения операций, изменения должны быть сохранены (надо выполнить commit()), а затем можно закрыть соединение, если оно больше не нужно.

Python позволяет использовать объект Connection, как менеджер контекста. В таком случае, не нужно явно делать commit и закрывать соединение. При этом:

- при возникновении исключения, транзакция автоматически откатывается
- если исключения не было, автоматически выполняется commit

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Пример использования соединения с базой, как менеджера контекстов (create_sw_inventory_ver2.py):

```
# -*- coding: utf-8 -*-
import sqlite3

data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


con = sqlite3.connect('sw_inventory3.db')
con.execute("""create table switch
               (mac text not NULL primary key, hostname text, model text, location text)""")

try:
    with con:
        query = "INSERT into switch values (?, ?, ?, ?)"
        con.executemany(query, data)

except sqlite3.IntegrityError as e:
    print("Error occured: ", e)

for row in con.execute("select * from switch"):
    print(row)

con.close()
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver2_functions.py:

```
# -*- coding: utf-8 -*-
from pprint import pprint
import sqlite3

data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


def create_connection(db_name):
    '''
    Функция создает соединение с БД db_name
    и возвращает его
    '''
    connection = sqlite3.connect(db_name)
    return connection
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver2_functions.py:

```
def write_data_to_db(connection, query, data):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

    Функция пытается записать все данные из списка data.
    Если данные удалось записать успешно, изменения сохраняются в БД
    и функция возвращает True.
    Если в процессе записи возникла ошибка, транзакция откатывается
    и функция возвращает False.
    '''
    try:
        with connection:
            connection.executemany(query, data)
    except sqlite3.IntegrityError as e:
        print("Error occured: ", e)
        return False
    else:
        print("Запись данных прошла успешно")
        return True
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver2_functions.py:

```
def get_all_from_db(connection, query):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить

    Функция возвращает данные полученные из БД.
    '''
    result = [row for row in connection.execute(query)]
    return result
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver2_functions.py:

```
if __name__ == '__main__':
    con = create_connection('sw_inventory3.db')

    print("Создание таблицы...")
    schema = """create table switch
                (mac text not NULL primary key, hostname text, model text, location text)"""
    con.execite(schema)

    query_insert = "INSERT into switch values (?, ?, ?, ?)"
    query_get_all = "SELECT * from switch"

    print("Запись данных в БД:")
    pprint(data)
    write_data_to_db(con, query_insert, data)
    print("\nПроверка содержимого БД")
    pprint(get_all_from_db(con, query_get_all))

    con.close()
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Результат выполнения скрипта выглядит так:

```
$ python create_sw_inventory_ver2_functions.py
Создание таблицы...
Запись данных в БД:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
Запись данных прошла успешно

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Теперь проверим как функция write_data_to_db отработает при наличии одинаковых MAC-адресов в данных.

```
# -*- coding: utf-8 -*-
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
         ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]

con = dbf.create_connection('sw_inventory3.db')

query_insert = "INSERT into switch values (?, ?, ?, ?)"
query_get_all = "SELECT * from switch"

print("\nПроверка текущего содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

print('-'*60)
print("Попытка записать данные с повторяющимся MAC-адресом:")
pprint(data2)
dbf.write_data_to_db(con, query_insert, data2)
print("\nПроверка содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Результат выполнения скрипта:

```
$ python create_sw_inventory_ver3.py

Проверка текущего содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
------------------------------------------------------------
Попытка записать данные с повторяющимся MAC-адресом:
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Error occured:  UNIQUE constraint failed: switch.mac

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Содержимое таблицы switch до и после добавления информации - одинаково. Это значит, что не записалась ни одна строка из списка data2.

Так получилось из-за того, что используется метод executemany и в пределах одной транзакции мы пытаемся записать все 4 строки. Если возникает ошибка с одной из них - откатываются все изменения.

Иногда, это именно то поведение, которое нужно. Если же надо чтобы игнорировались только строки с ошибками, надо использовать метод execute и записывать каждую строку отдельно.

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

В файле create_sw_inventory_ver4.py создана функция write_rows_to_db, которая уже по очереди пишет данные и, если возникла ошибка, то только изменения для конкретных данных откатываются:

```
# -*- coding: utf-8 -*-
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
         ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver4.py

```
def write_rows_to_db(connection, query, data, verbose=False):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

    Функция пытается записать по очереди кортежи из списка data.
    Если кортеж удалось записать успешно, изменения сохраняются в БД.
    Если в процессе записи кортежа возникла ошибка, транзакция откатывается.

    Флаг verbose контролирует то, будут ли выведены сообщения об удачной
    или неудачной записи кортежа.
    '''
    for row in data:
        try:
            with connection:
                connection.execute(query, row)
        except sqlite3.IntegrityError as e:
            if verbose:
                print('При записи данных "{}" возникла ошибка'.format(', '.join(row), e))
        else:
            if verbose:
                print('Запись данных "{}" прошла успешно'.format(', '.join(row)))
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Файл create_sw_inventory_ver4.py

```
con = dbf.create_connection('sw_inventory3.db')

query_insert = 'INSERT into switch values (?, ?, ?, ?)'
query_get_all = 'SELECT * from switch'

print('\nПроверка текущего содержимого БД')
pprint(dbf.get_all_from_db(con, query_get_all))

print('-'*60)
print('Попытка записать данные с повторяющимся MAC-адресом:')
pprint(data2)
write_rows_to_db(con, query_insert, data2, verbose=True)
print('\nПроверка содержимого БД')
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

CONNECTION КАК МЕНЕДЖЕР КОНТЕКСТА

Теперь результат выполнения будет таким (пропущен только sw7):

```
$ python create_sw_inventory_ver4.py

Проверка текущего содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
----------------------------------------------------------
Попытка записать данные с повторяющимся MAC-адресом:
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Запись данных "0055.AAAA.CCCC, sw5, Cisco 3750, London, Green Str" прошла успешно
Запись данных "0066.BBBB.CCCC, sw6, Cisco 3780, London, Green Str" прошла успешно
При записи данных "0000.AAAA.DDDD, sw7, Cisco 2960, London, Green Str" возникла ошибка
Запись данных "0088.AAAA.CCCC, sw8, Cisco 3750, London, Green Str" прошла успешно

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ SQLITE

ПРИМЕР ИСПОЛЬЗОВАНИЯ SQLITE

Запишем информацию полученную из вывода sh ip dhcp snooping binding в SQLite. Это позволит делать запросы по любому параметру и получать недостающие.

Для этого примера, достаточно создать одну таблицу, где будет храниться информация.

Определение таблицы прописано в отдельном файле dhcp_snooping_schema.sql и выглядит так:

```
create table if not exists dhcp (
    mac          text not NULL primary key,
    ip           text,
    vlan         text,
    interface    text
);
```

Теперь надо создать файл БД, подключиться к базе данных и создать таблицу (файл create_sqlite_ver1.py):

```
import sqlite3

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print("Done")

conn.close()
```

Комментарии к файлу:

- при выполнении строки conn = sqlite3.connect('dhcp_snooping.db'):
  - создается файл dhcp_snooping.db, если его нет
  - создается объект Connection
- в БД создается таблица, на основании команд, которые указаны в файле dhcp_snooping_schema.sql:
  - открываем файл dhcp_snooping_schema.sql
  - schema = f.read() - считываем весь файл как одну строку
  - conn.executescript(schema) - метод executescript позволяет выполнять команды SQL, которые прописаны в файле

Выполняем скрипт:

```
$ python create_sqlite_ver1.py
Creating schema...
Done
```

В результате должен быть создан файл БД и таблица dhcp.

Проверить, что таблица создалась, можно с помощью утилиты sqlite3, которая позволяет выполнять запросы прямо в командной строке.

Выведем список созданных таблиц (запрос такого вида позволяет проверить какие таблицы созданы в DB):

```
$ sqlite3 dhcp_snooping.db "SELECT name FROM sqlite_master WHERE type='table'"
dhcp
```

Теперь нужно записать информацию из вывода команды sh ip dhcp snooping binding в таблицу (файл dhcp_snooping.txt):

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  --------------------
00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0/1
00:04:A3:3E:5B:69   10.1.5.2         63951       dhcp-snooping   5     FastEthernet0/10
00:05:B3:7E:9B:60   10.1.5.4         63253       dhcp-snooping   5     FastEthernet0/9
00:09:BC:3F:A6:50   10.1.10.6        76260       dhcp-snooping   10    FastEthernet0/3
Total number of bindings: 4
```

Во второй версии скрипта, сначала вывод в файле dhcp_snooping.txt обрабатывается регулярными выражениями, а затем, добавляются записи в БД (файл create_sqlite3_ver2.py):

```
import sqlite3
import re

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print("Done")

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = """insert into dhcp (mac, ip, vlan, interface)
```

Комментарии к скрипту:

- В этом скрипте используется еще один вариант записи в БД
  - строка query описывает запрос. Но, вместо значений указываются знаки вопроса. Такой вариант записи запроса, позволяет динамически подставлять значение полей
  - затем, методу execute, передается строка запроса и кортеж row, где находятся значения

Выполняем скрипт:

```
$ python create_sqlite_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Проверим, что данные записались:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ---------------
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/1
00:05:B3:7E:9B:60  10.1.5.4    5           FastEthernet0/9
00:09:BC:3F:A6:50  10.1.10.6   10          FastEthernet0/3
```

Теперь попробуем запросить по определенному параметру:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp where ip = '10.1.5.2'"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ----------------
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/10
```

То есть, теперь на основании одного параметра, можно получать остальные.

Файл create_sqlite_ver3.py:

```
import os
import sqlite3
import re

data_filename = 'dhcp_snooping.txt'
db_filename = 'dhcp_snooping.db'
schema_filename = 'dhcp_snooping_schema.sql'

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

db_exists = os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if not db_exists:
    print('Creating schema...')
    with open(schema_filename, 'r') as f:
        schema = f.read()
    conn.executescript(schema)
    print('Done')
else:
```

Если файла нет (предварительно его удалить):

```
$ rm dhcp_snooping.db
$ python create_sqlite_ver3.py
Creating schema...
Done
Inserting DHCP Snooping data
```

В случае если файл уже есть, но данные не записаны:

```
$ rm dhcp_snooping.db

$ python create_sqlite_ver1.py
Creating schema...
Done
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
```

Если есть и БД и данные:

```
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
Error occured:  UNIQUE constraint failed: dhcp.mac
Error occured:  UNIQUE constraint failed: dhcp.mac
Error occured:  UNIQUE constraint failed: dhcp.mac
Error occured:  UNIQUE constraint failed: dhcp.mac
```

Теперь делаем отдельный скрипт, который занимается отправкой запросов в БД и выводом результатов. Он должен:

- ожидать от пользователя ввода параметров:
  - имя параметра
  - значение параметра
- делать нормальный вывод данных по запросу

Файл get_data_ver1.py:

```
# -*- coding: utf-8 -*-
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'


key, value = sys.argv[1:]
keys = ['mac', 'ip', 'vlan', 'interface']
keys.remove(key)

conn = sqlite3.connect(db_filename)

#Позволяет далее обращаться к данным в колонках, по имени колонки
conn.row_factory = sqlite3.Row

print("\nDetailed information for host(s) with", key, value)
print('-' * 40)

query = "select * from dhcp where {} = ?".format( key )
result = conn.execute(query, (value,))

for row in result:
    for k in keys:
        print("{:12}: {}".format(k, row[k]))
    print('-' * 40)
```

Показать параметры хоста с IP 10.1.10.2:

```
$ python get_data_ver1.py ip 10.1.10.2

Detailed information for host(s) with ip 10.1.10.2
----------------------------------------
mac         : 00:09:BB:3D:D6:58
vlan        : 10
interface   : FastEthernet0/1
----------------------------------------
```

Показать хосты в VLAN 10:

```
$ python get_data_ver1.py vlan 10

Detailed information for host(s) with vlan 10
---------------------------------------
mac         : 00:09:BB:3D:D6:58
ip          : 10.1.10.2
interface   : FastEthernet0/1
---------------------------------------
mac         : 00:07:BC:3F:A6:50
ip          : 10.1.10.6
interface   : FastEthernet0/3
---------------------------------------
```

Вторая версия скрипта для получения данных, с небольшими улучшениями:

- Вместо форматирования строк, используется словарь, в котором описаны запросы, соответствующие каждому ключу.
- Выполняется проверка ключа, который был выбран
- Для получения заголовков всех столбцов, который соответствуют запросу, используется метод keys()

Файл get_data_ver2.py:

```
# -*- coding: utf-8 -*-
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

query_dict = {'vlan': "select mac, ip, interface from dhcp where vlan = ?",
              'mac': "select vlan, ip, interface from dhcp where mac = ?",
              'ip': "select vlan, mac, interface from dhcp where ip = ?",
              'interface': "select vlan, mac, ip from dhcp where interface = ?"}


key, value = sys.argv[1:]
keys = query_dict.keys()

if not key in keys:
    print("Enter key from {}".format(', '.join(keys)))
else:
    conn = sqlite3.connect(db_filename)
    conn.row_factory = sqlite3.Row

    print("\nDetailed information for host(s) with", key, value)
    print('-' * 40)

    query = query_dict[key]
    result = conn.execute(query, (value,))

    for row in result:
        for row_name in row.keys():
```

В этом скрипте есть несколько недостатков:

- не проверяется количество аргументов, которые передаются скрипту
- хотелось бы собирать информацию с разных коммутаторов. А для этого надо добавить поле, которое указывает на каком коммутаторе была найдена запись

Кроме того, многое нужно доработать в скрипте, который создает БД и записывает данные.

Все доработки будут выполняться в заданиях этого раздела.