Работа с БД в Python

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

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

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 мы будем использовать 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.

Слово SQL-сервер здесь не используем, потому что как таковой сервер там не нужен — весь функционал, который встраивается в SQL-сервер, реализован внутри библиотеки (и, соответственно, внутри программы, которая её использует).

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

SQLite CLI

Для начала, надо установить SQLite3 принятым в системе образом.
Для Debian: apt-get install sqlite3

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

Попробуем с помощью этой утилиты разобраться с базовыми командами SQL, которые нам понадобятся для работы с БД.

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

```
nata:$ sqlite3 testDB.db
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
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.

Рассмотрим несколько метакоманд:

- .help - команда, которая выводит подсказку со списком всех метакоманд
- .exit или .quit - выход из сессии sqlite3
- .databases - показывает присоединенные БД
- .tables - показывает доступные таблицы

```
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

Для начала, создадим таблицу switch, в которой будет хранится информация о коммутаторах:

```
sqlite> CREATE table switch (
   ...>     mac           text primary key,
   ...>     hostname      text,
   ...>     model         text,
   ...>     location      text
   ...> );
```

Аналогично можно было создать таблицу и таким образом:

```
sqlite> create table switch (mac text primary key, hostname text, model text,
location text);
```

В данном примере мы описали таблицу switch, используя язык DDL. Мы определили какие поля будут в таблице и значения какого типа будут в них находиться.

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

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

В нашем примере это вполне логично, так как MAC-адрес у коммутаторов должен быть уникальным.

Метакоманда .schema и DROP

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

```
sqlite> .schema switch
CREATE TABLE switch (
mac             text primary key,
hostname        text,
model           text,
location        text
);
```

Удалить таблицу можно так:

```
sqlite> DROP table switch
```

INSERT

Добавим записи в таблицу. Есть несколько вариантов добавления записей, в зависимости от того, все ли поля будут заполнены и будут ли они идти по порядку определения полей или нет.

Если мы будем указывать значения для всех полей, то добавить запись можно таким образом (порядок полей должен соблюдаться):

```
sqlite> INSERT into switch values ('0000.AAAA.CCCC', '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 ('0000.BBBB.CCCC', 'Cisco 3850', 'London, Green Str',
'sw5');
```

SELECT

Теперь в нашей таблице две записи. Просмотрим их:

```
sqlite> SELECT * from switch;
0000.AAAA.CCCC|sw1|Cisco 3750|London, Green Str
0000.BBBB.CCCC|sw5|Cisco 3850|London, Green Str
```

В данном случае мы отображаем все записи в таблице switch.

Метакоманды .headers, .mode

В отображении таблицы не хватает названия полей. Включить их отображение можно с помощью команды **.headers ON.**

```
sqlite> .headers ON
sqlite> SELECT * from switch;
mac|hostname|model|location
0000.AAAA.CCCC|sw1|Cisco 3750|London, Green Str
0000.BBBB.CCCC|sw5|Cisco 3850|London, Green Str
```

Заголовки отобразились, но в целом отображение не очень приятное. Хотелось бы, чтобы все выводилось в виде колонок. За форматирование вывода отвечает **команда .mode.**

Нам нужен режим **.mode column**:

```
sqlite> .mode column
sqlite> SELECT * from switch;
mac              hostname    model       location
---------------  ----------  ----------  ----------------
0000.AAAA.CCCC   sw1         Cisco 3750  London, Green Str
0000.BBBB.CCCC   sw5         Cisco 3850  London, Green Str
```

При желании, можно выставить и ширину колонок. Для этого используется команда .width. Например, попробуйте выставить .width 20.

ALTER

Теперь попробуем изменить таблицу. Добавим новые поля в таблицу switch.
Добавим в определение таблицы такие новые поля:

- mngmt_ip - IP-адрес коммутатора в менеджмент VLAN
- mngmt_vid - VLAN ID (номер VLAN) для менеджмент VLAN
- mngmt_vname - Имя VLAN, который используется для менеджмента

Добавление записей выполняется с помощью DDL, используя команду ALTER:

```
sqlite> ALTER table switch ADD COLUMN mngmt_ip text;
sqlite> ALTER table switch ADD COLUMN mngmt_vid varchar(10);
sqlite> ALTER table switch ADD COLUMN mngmt_vname  text;
```

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

```
sqlite> SELECT * from switch;
mac             hostname  model       location   mngmt_ip  mngmt_vid mngmt_vname
--------------  --------  ----------  ---------  --------  --------- -----------
0000.AAAA.CCCC  sw1       Cisco 3750  Green Str
0000.BBBB.CCCC  sw5       Cisco 3850  Green Str
```

UPDATE

Если информация о каком-то из коммутаторов изменилась, и необходимо изменить одно из полей, используется команда UPDATE.

Например, предположим, что sw1 был заменен с модели 3750 на модель 3850. Соответственно, изменилось не только поле модель, но и поле MAC-адрес.

Внесем изменения и проверим результат:

```
sqlite> UPDATE switch set model = '3850' where hostname = 'sw1';
sqlite> UPDATE switch set mac = '0000.DDDD.DDDD' where hostname = 'sw1';


sqlite> SELECT * from switch;
mac             hostname  model  location   mngmt_ip    mngmt_vid   mngmt_vname
--------------  --------  -----  ---------  ----------  ----------  -----------
0000.DDDD.DDDD  sw1       3850   Green Str  10.255.0.1  255         MNGMT
0000.BBBB.CCCC  sw5       3850   Green Str  10.255.0.5  255         MNGMT
```

WHERE

Оператор WHERE используется для уточнения запроса. С помощью этого оператора мы можем указывать определенные условия, по которым отбираются данные. Если условие выполнено, возвращается соответствующее значение из таблицы, если нет, не возвращается. Например, наша таблица с коммутаторами немного разрослась:

```
sqlite> SELECT * from switch;
mac              hostname  model location   mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  ----  ---------  ----------  ----------  -----------
0000.DDDD.DDDD   sw1       3850  Green Str  10.255.0.1  255         MNGMT
0000.BBBB.CCCC   sw5       3850  Green Str  10.255.0.5  255         MNGMT
0000.2222.CCCC   sw2       3750  Green Str  10.255.0.2  255         MNGMT
0000.3333.CCCC   sw3       3750  Green Str  10.255.0.3  255         MNGMT
0000.4444.CCCC   sw4       3650  Green Str  10.255.0.4  255         MNGMT
```

Воспользуется оператором WHERE и отфильтруем вывод. Отобразим информацию только о тех коммутаторах, модель которых 3750:

```
sqlite> SELECT * from switch WHERE model = 'Cisco 3750';
mac              hostname  model location   mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  ----  ---------  ----------  ----------  -----------
0000.2222.CCCC   sw2       3750  Green Str  10.255.0.2  255         MNGMT
0000.3333.CCCC   sw3       3750  Green Str  10.255.0.3  255         MNGMT
```

WHERE

Например, если в таблице поле model записано в разном формате:

```
sqlite> SELECT * from switch;
mac              hostname  model  location    mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  -----  ----------  ----------  ----------  -----------
0000.DDDD.DDDD   sw1        3850  Green Str   10.255.0.1  255         MNGMT
0000.BBBB.CCCC   sw5        3850  Green Str   10.255.0.5  255         MNGMT
0000.2222.CCCC   sw2       C3750  Green Str   10.255.0.2  255         MNGMT
0000.3333.CCCC   sw3        3750  Green Str   10.255.0.3  255         MNGMT
0000.4444.CCCC   sw4        3650  Green Str   10.255.0.4  255         MNGMT
```

В таком варианте предыдущий запрос с оператором WHERE нам не поможет:

```
sqlite> SELECT * from switch WHERE model = 'Cisco 3750';
mac              hostname  model  location    mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  -----  ----------  ----------  ----------  -----------
0000.3333.CCCC   sw3        3750  Green Str   10.255.0.3  255         MNGMT
```

LIKE

Но вместе с оператором WHERE мы можем использовать оператор LIKE:

```
sqlite> SELECT * from switch WHERE model LIKE '%3750';
mac              hostname  model location   mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  ----  ---------  ----------  ----------  -----------
0000.2222.CCCC   sw2       3750  Green Str  10.255.0.2  255         MNGMT
0000.3333.CCCC   sw3       3750  Green Str  10.255.0.3  255         MNGMT
```

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

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

ORDER BY

И еще один полезный оператор ORDER BY. Он используется для сортировки вывода по определенному полю, по возрастанию или убыванию.

Например, выведем все записи в таблице switch и отсортируем их по имени коммутаторов (по умолчанию выполняется сортировка по умолчанию, поэтому параметр ASC можно не указывать):

```
sqlite> SELECT * from switch ORDER BY hostname ASC;
mac              hostname  model location   mngmt_ip    mngmt_vid   mngmt_vname
---------------  --------  ----  ---------  ----------  ----------  -----------
0000.DDDD.DDDD   sw1        3850  Green Str  10.255.0.1  255         MNGMT
0000.2222.CCCC   sw2       C3750  Green Str  10.255.0.2  255         MNGMT
0000.3333.CCCC   sw3        3750  Green Str  10.255.0.3  255         MNGMT
0000.4444.CCCC   sw4        3650  Green Str  10.255.0.4  255         MNGMT
0000.BBBB.CCCC   sw5        3850  Green Str  10.255.0.5  255         MNGMT
```

Модуль sqlite3

Connection

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

Например, мы можем его создать так:

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

У объекта Connection есть несколько методов, с помощью которых, мы можем выполнять команды SQL:

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

Cursor

Для чтения данных из БД, используется объект Cursor - это основной способ работы с БД.

Создается курсор из соединения с БД:

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

На самом деле, при использовании методов:

- Connection.execute()
- Connection.executemany()
- Connection.executescript()

также создается курсор, но неявно, и фактически эти методы применяются к объекту Cursor.

Чтение данных из БД

При чтении данных, мы должны сначала выполнить выражение SQL, используя один из методов (описаны выше):

- execute()
- executemany()
- executescript()

А затем, с помощью метода fetch...(), мы обрабатываем полученные данные, и, в зависимости от метода, возвращаем их:

- fetchone() - возвращает одну строку данных
- fetchmany() - возвращает список строк данных.
  - С помощью параметра size, можно указывать какое количество строк возвращается
- fetchall() - возвращает все строки в результате запроса, в виде списка

Connection как менеджер контекста

После выполнения всех необходимых операций, изменения должны быть сохранены (надо выполнить commit()), а затем можно закрыть курсор, если он больше не нужен.

При работе с SQLite, мы можем использовать объект Connection, как менеджер контекста и, в таком случае, нам не надо явно делать commit и закрывать соединение:

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

Пример использования соединения с базой, как менеджера контекстов:

```
with sqlite3.connect('test.db') as conn
```

Пример использования SQLite

Задача

В разделе "Регулярые выражения" был пример разбора вывода команды show ip dhcp snooping binding. На выходе, мы получили информацию о параметрах подключенных устройств (interface, IP, MAC, VLAN).

Результат хороший, но в таком варианте мы можем посмотреть только все подключенные устройства к коммутатору. Если же нам нужно узнать на основании одного из параметров, другие, то в таком виде это не очень удобно.

Например, если нам нужно по IP-адресу получить информацию о том, к какому интерфейсу подключен компьютер, какой у него MAC-адрес и в каком он VLAN, то по выводу скрипта это сделать не очень просто и, главное, не очень удобно.

Попробуем записать в SQLite информацию полученную из вывода sh ip dhcp snooping binding.

Это позволит нам легко делать запрос по любому параметру и получать недостающие.

dhcp_snooping_schema.sql

Для этого примера нам достаточно одной таблицы, где будет храниться информация.

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

```
create table dhcp (
    mac           text primary key,
    ip            text,
    vlan          text,
    interface     text
);
```

Для всех полей определен тип данных "текст". И MAC-адрес является первичным ключом нашей таблицы. Что вполне логично, так как, MAC-адрес должен быть уникальным.

create_sqlite3_ver1.py

Теперь попробуем создать файл БД, подключиться к базе данных и создать таблицу.

Файл create_sqlite3_ver1.py:

```
import sqlite3

with sqlite3.connect('dhcp_snooping.db') as conn:
    print 'Creating schema...'
    with open('dhcp_snooping_schema.sql', 'rt') as f:
        schema = f.read()
        conn.executescript(schema)
    print "Done"
```

Комментарии к файлу:

- используем менеджер контекста with ... as
- при выполнении строки with sqlite3.connect('dhcp_snooping.db') as conn:
  - 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 несколько команд SQL, которые прописаны в файле (в данном случае команда одна)

create_sqlite3_ver1.py

Теперь нам нужно записать информацию, которую мы получили из вывода 'sh ip dhcp snooping binding' в таблицу.

Дополняем наш скрипт create_sqlite3_ver1.py таким образом, чтобы он и парсил команду (переносим сюда регулярные выражения) и добавлял записи из файла dhcp_snooping.txt в БД:

```
import sqlite3
import re

regex = re.compile('(.+?) +(.*?) +\d+ +[\w-]+ +(\d+) +(.*$)')
result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        if line[0].isdigit():
            result.append(regex.search(line).groups())

with sqlite3.connect('dhcp_snooping.db') as conn:
    print 'Creating schema...'
    with open('dhcp_snooping_schema.sql', 'rt') as f:
        schema = f.read()
        conn.executescript(schema)
    print "Done"

    print 'Inserting DHCP Snooping data'

    for val in result:
        query = """insert into dhcp (mac, ip, vlan, interface)
                values (?, ?, ?, ?)"""
        conn.execute(query, val)
```

create_sqlite3_ver1.py

Комментарии к скрипту:

- в регулярном выражении, которое проходится по выводу команды sh ip dhcp snooping binding, используются не именованные группы, как в примере раздела "Регулярные выражения"
  - группы созданы только для тех элементов, которые нас интересуют
- result - это список, в котором хранится результат обработки команды
  - но теперь тут не словари, а кортежи с результатами
  - это нужно для того, чтобы их можно было сразу передавать на запись в БД
- Перебираем в полученном списке кортежей, элементы
- В этом скрипте мы используем еще один вариант записи в БД
  - строка query описывает запрос. Но, вместо значений мы указываем знаки вопроса
  - затем методу execute, мы передаем строку запроса и кортеж val, где находятся значения

create_sqlite3_ver1.py

Проверим, что данные записались:

```
nata:~$ sqlite3 dhcp_snooping.db "select * from dhcp"
00:09:BB:3D:D6:58|10.1.10.2|10|FastEthernet0/1
00:04:A3:3E:5B:69|10.1.5.2|5|FastEthernet0/10
00:05:B3:7E:9B:60|10.1.5.4|5|FastEthernet0/9
00:07:BC:3F:A6:50|10.1.10.6|10|FastEthernet0/3
00:09:BC:3F:A6:50|192.168.100.100|1|FastEthernet0/7
```

Теперь попробуем запросить по определенному параметру:

```
sqlite3 dhcp_snooping.db "select * from dhcp where ip = '10.1.5.2'"
00:04:A3:3E:5B:69|10.1.5.2|5|FastEthernet0/10
```

То есть, теперь на основании одного параметра, мы можем получать остальные.

create_sqlite_db_ver2.py

Добавляем проверку: если файл БД есть, то не надо создавать таблицу, считаем, что она уже создана.

```
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('(.+?) +(.*?) +\d+ +[\w-]+ +(\d+) +(.*$)')

with open(data_filename) as data:
    result = [regex.search(line).groups() for line in data if line[0].isdigit()]

db_exists = os.path.exists(db_filename)

with sqlite3.connect(db_filename) as conn:
    if not db_exists:
        print 'Creating schema...'
        with open(schema_filename, 'rt') as f:
            schema = f.read()
        conn.executescript(schema)
        print 'Done'

        print 'Inserting DHCP Snooping data'
        for val in result:
            query = """insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)"""
            conn.execute(query, val)
    else:
        print 'Database exists, assume dhcp table does, too.'
```

create_sqlite_db_ver2.py

Теперь у нас есть проверка наличия файла БД, и файл БД будет создаваться только, если его нет.

Проверим. В случае если файл уже есть:

```
nata:$ python create_sqlite_db_ver2.py
Database exists, assume dhcp table does, too.
```

Если файла нет (предварительно его удаляем):

```
nata:$ rm dhcp_snooping.db
nata:$ python create_sqlite_db_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

create_sqlite_db_ver2.py

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

```
nata:$ python create_sqlite_db_ver2.py
Database exists, assume dhcp table does, too.
```

Если файла нет (предварительно его удаляем):

```
nata:$ rm dhcp_snooping.db
nata:$ python create_sqlite_db_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

get_data_ver1.py

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

db_filename = 'dhcp_snooping.db'

if len(sys.argv) == 1:
    print "\nВ таблице dhcp такие записи:"
    print '-' * 70
    with sqlite3.connect(db_filename) as conn:
        cursor = conn.cursor()
        cursor.execute('select * from dhcp')

        for row in cursor.fetchall():
            print '%-18s %-17s %-5s %-20s' % row

...
```

get_data_ver1.py (продолжение)

```
...
elif len(sys.argv) == 3:
    key, value = sys.argv[1:]
    keys = ['mac', 'ip', 'vlan', 'interface']
        #Проверка указанного ключа (параметра)
    if key in keys:
        keys.remove(key)
        with sqlite3.connect(db_filename) as conn:
            #Позволяет далее обращаться к данным в колонках, по имени колонки
            conn.row_factory = sqlite3.Row

            cursor = conn.cursor()

            cursor.execute("select * from dhcp where %s = ?" % key, (value,))

            print "\nDetailed information for host(s) with", key, value
            print '-' * 40
            for row in cursor.fetchmany(10):
                for k in keys:
                    print "%-12s: %s" % (k,row[k])
                print '-' * 40
    else:
        print "Данный параметр не поддерживается."
        print "Допустимые значения параметров: mac, ip, vlan, interface"
else:
    print "Введите, пожалуйста, два параметра"
```

get_data_ver1.py

Комментарии к скрипту:

- сначала проверяем количество параметров, которые ввел пользователь
  - если параметры не переданы, то отображаем все содержимое БД
    - в проверке длинна sys.argv равно 1, из-за того, что имя скрипта тоже находится в этом списке
    - затем обрабатываем все результаты с помощью метода fetchall() и выводим их таблицей
  - если было передано 2 параметра (3 вместе с именем скрипта):
    - проверяем правильное ли было введено имя ключа (параметра)
      - если правильно, то подключаемся к БД:
        - conn.row_factory = sqlite3.Row - позволяет далее обращаться к данным в колонках, по имени колонки
        - выбираем из БД те строки, в которых ключ равен указанному значению и выводим их
      - если имя параметра было указано неправильно, выводим сообщение об ошибке
  - если был передан только один параметр, выводим сообщение об ошибке

get_data_ver1.py

Проверим работу скрипта.

Сначала вызовем скрипт без параметров (должно быть показано содержимое БД):

nata:$ python get_data_ver1.py

```
В таблице dhcp такие записи:
-------------------------------------------------------------
00:09:BB:3D:D6:58  10.1.10.2          10    FastEthernet0/1
00:04:A3:3E:5B:69  10.1.5.2           5     FastEthernet0/10
00:05:B3:7E:9B:60  10.1.5.4           5     FastEthernet0/9
00:07:BC:3F:A6:50  10.1.10.6          10    FastEthernet0/3
00:09:BC:3F:A6:50  192.168.100.100    1     FastEthernet0/7
```

get_data_ver1.py

Показать параметры хоста с IP 10.1.10.2:

```
nata:$ 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:

```
nata:$ 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
----------------------------------------
```

get_data_ver1.py

Проверим скрипт на ошибки.
Сначала зададим неправильно название параметра:

```
nata:$ python get_data_ver1.py vln 10
Данный параметр не поддерживается.
Допустимые значения параметров: mac, ip, vlan, interface
```

Указываем имя параметра без значения параметра:

```
nata:$ python get_data_ver1.py vlan
Введите, пожалуйста, два параметра
```

Python для сетевых инженеров

Автор курса: Наташа Самойленко
nataliya.samoylenko@gmail.com