PYTHON AND SQL

INTRODUCTION

The history of SQL goes back to the early 70th. SQL is a Structured Query Language, which is based on a relational model, as it was described in Edgar F. Codds's 1970 paper "A Relational Model of Data for Large Shared Data Banks. SQL is often pronounced like "sequel". SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987. As most people coming to this website are already familiar with mSQL, PostgresSQL, MySQL or other variants of SQL, we will not enlarge on SQL itself.

A database is an organized collection of data. The data are typically organized to model aspects of reality in a way that supports processes requiring this information. The term "database" can both refer to the data themselves or to the database management system. The Database management system is a software application for the interaction between users database itself. Users don't have to be human users. They can be other programs and applications as well. We will learn how Python or better a Python program can interact as a user of an SQL database.

This is an introduction into using SQLite and MySQL from Python. The Python standard for database interfaces is the Python DB-API, which is used by Python's database interfaces. The DB-API has been defined as a common interface, which can be used to access relational databases. In other words, the code in Python for communicating with a database should be the same, regardless of the database and the database module used. Even though we use lots of SQL examples, this is not an introduction into SQL but a tutorial on the Python interface. To learn SQL you have to consult a SQL tutorial.

SQLITE

SQLite is a simple relational database system, which saves its data in regular data files or even in the internal memory of the computer, i.e. the RAM. It was developped for embedded applications, like Mozilla-Firefox (Bookmarks), Symbian OS or Android. SQLITE is "quite" fast, even though it uses a simple file. It can be used for large databases as well. If you want to use SQLite, you have to import the module sqlite3. To use a database, you have to create first a Connection object. The connection object will represent the database. The argument of connection - in the following example "companys.db" - functions both as the name of the file, where the data will be stored, and as the name of the database. If a file with this name exists, it will be opened. It has to be a SQLite database file of course! In the following example, we will open a database called company. The file does not have to exist.:

```
>>> import sqlite3
>>> connection = sqlite3.connect("company.db")
```

We have now created a database with the name "company". It's like having sent the command "CREATE DATABASE company;" to a SQL server. If you call "sqlite3.connect('company.db')" again, it will open the previously created database.

After having created an empty database, you will most probably add one or more tables to this database. The SQL syntax for creating a table "employee" in the database "company" looks like this:

```
CREATE TABLE employee (
staff_number INT NOT NULL AUTO_INCREMENT,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE,
PRIMARY KEY (staff_number) );
```

This is the way, somebody might do it on a SQL command shell. Of course, we want to do this directly from Python. To be capable to send a command to "SQL", or SQLite, we need a cursor object. Usually, a cursor in SQL and databases is a control structure to traverse over the records in a database. So it's used for the fetching of the results. In SQLite (and other Python DB interfaces)it is more generally used. It's used for performing all SQL commands.

We get the cursor object by calling the cursor() method of connection. An arbitrary number of cursors can be created. The cursor is used to traverse the records from the result set. We can define a SQL command with a triple quoted string in Python:

```
sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""
```

Concerning the SQL syntax: You may have noticed that the AUTOINCREMENT field is missing in the SQL code within our Python program. We have defined the staff_number field as "INTEGER PRIMARY KEY" A column which is labelled like this will be automatically auto-incremented in SQLite3. To put it in other words: If a column of a table is declared to be an INTEGER PRIMARY KEY, then whenever a NULL will be used as an input for this column, the NULL will be automatically converted into an integer which will one larger than the highest value so far used in that column. If the table is empty, the value 1 will be used. If the largest existing value in this column has the 9223372036854775807, which is the largest possible INT in SQLite, an unused key value is chosen at random.

Now we have a database with a table but no data included. To populate the table we will have to send the "INSERT" command to SQLite. We will use again the execute method. The following example is a complete working example. To run the program you will either have to remove the file company.db or uncomment the "DROP TABLE" line in the SQL command:

```
import sqlite3
connection = sqlite3.connect("company.db")

cursor = connection.cursor()

# delete
#cursor.execute("""DROP TABLE employee;""")

sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""
```

```
cursor.execute(sql_command)

sql_command = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (NULL, "William", "Shakespeare", "m", "1961-10-25");"""
cursor.execute(sql_command)


sql_command = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (NULL, "Frank", "Schiller", "m", "1955-08-17");"""
cursor.execute(sql_command)

# never forget this, if you want the changes to be saved:
connection.commit()

connection.close()
```

Of course, in most cases, you will not literally insert data into a SQL table. You will rather have a lot of data inside of some Python data type e.g. a dictionary or a list, which has to be used as the input of the insert statement.

The following working example, assumes that you have already an existing database company.db and a table employee. We have a list with data of persons which will be used in the INSERT statement:

```
import sqlite3
connection = sqlite3.connect("company.db")

cursor = connection.cursor()

staff_data = [ ("William", "Shakespeare", "m", "1961-10-25"),
               ("Frank", "Schiller", "m", "1955-08-17"),
               ("Jane", "Wall", "f", "1989-03-14") ]

for p in staff_data:
    format_str = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (NULL, "{first}", "{last}", "{gender}", "{birthdate}");"""

    sql_command = format_str.format(first=p[0], last=p[1], gender=p[2], birthdate = p[3])
    cursor.execute(sql_command)
```

The time has come now to finally query our employee table:

```
import sqlite3
connection = sqlite3.connect("company.db")

cursor = connection.cursor()

cursor.execute("SELECT * FROM employee")
print("fetchall:")
result = cursor.fetchall()
for r in result:
    print(r)
cursor.execute("SELECT * FROM employee")
print("\nfetch one:")
res = cursor.fetchone()
print(res)
```

If we run this program, saved as "sql_company_query.py", we get the following result, depending on the actual data:

```
$ python3 sql_company_query.py
fetchall:
(1, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(2, 'Frank', 'Schiller', 'm', None, '1955-08-17')
```

```
(3, 'Bill', 'Windows', 'm', None, '1963-11-29')
(4, 'Esther', 'Wall', 'm', None, '1991-05-11')
(5, 'Jane', 'Thunder', 'f', None, '1989-03-14')

fetch one:
(1, 'William', 'Shakespeare', 'm', None, '1961-10-25')
```

MYSQL

If you work under a Python 2.x version, the module MySQLdb can be used. It has to be installed. This can be accomplished under Debian or Ubuntu like this:

```
sudo apt-get install python-MySQLdb
```

If you work with Python 3, you have to make sure that you write everything lowercase:

```
sudo apt-get install python3-mysqldb
```

Of course, you have also the possibility to install it via "pip install" inside a virtualenv:

```
pip install mysqlclient
```

- import the MySQLdb modul
- Open a connection to the SQL server
- Sending and receiving commands
- Closing the connection to SQL

Importing and connecting looks like this:

```
import MySQLdb

connection = MySQLdb.connect (host = "localhost",
                              user = "testuser",
                              passwd = "testpass",
                              db = "company")

cursor = connection.cursor()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone()
print("server version:", row[0])
cursor.close()
connection.close()
```

For the following examples, we assume that you have created a user "pytester". You can do this e.g. on the command line with the following commands: First we start a mysql session with:

```
mysql -u root -p
```

On the mysql shell we continue with:

```
mysql> CREATE USER 'pytester'@'localhost' IDENTIFIED BY 'monty';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON *.* TO 'pytester'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql>
```

Let's check the MySQL server version by using the previously created connection. To do this, we have to create a cursor object first:

```
import mysql.connector as mc

connection = mc.connect (host = "localhost",
                         user = "pytester",
                         passwd = "monty",
                         db = "company")
cursor = connection.cursor()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone()
print("server version:", row[0])
cursor.close()
connection.close()
```

The output may look like this:

```
server version: 5.5.52-0ubuntu0.14.04.1
```

Like in our example for sqlite3 in the beginning of this chapter, we will create a table employee and fill it with some data. The program works only under Python 3:

```
import sys
import mysql.connector as mc

try:
    connection = mc.connect (host = "localhost",
                             user = "pytester",
                             passwd = "monty",
                             db = "company")
except mc.Error as e:
    print("Error %d: %s" % (e.args[0], e.args[1]))
    sys.exit(1)

cursor = connection.cursor()

cursor.execute ("DROP TABLE IF EXISTS employee")

# delete
#cursor.execute("""DROP TABLE employee;""")

sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""

cursor.execute(sql_command)

staff_data = [ ("William", "Shakespeare", "m", "1961-10-25"),
               ("Frank", "Schiller", "m", "1955-08-17"),
               ("Jane", "Wall", "f", "1989-03-14"),
              ]

for staff, p in enumerate(staff_data):
    format_str = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES ({staff_no}, '{first}', '{last}', '{gender}', '{birthdate}');"""

    sql_command = format_str.format(staff_no=staff, first=p[0], last=p[1], gender=p[2],
```

```
birthdate = p[3])
    print(sql_command)
    cursor.execute(sql_command)

connection.commit()


cursor.close()
connection.close()
```

This program returns the following output which corresponds to the insertions into the table 'employee':

```
INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (0, 'William', 'Shakespeare', 'm', '1961-10-25');
INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (1, 'Frank', 'Schiller', 'm', '1955-08-17');
INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (2, 'Jane', 'Wall', 'f', '1989-03-14');
```

After this, we want to query our database again:

```
import sys
import mysql.connector as mc

try:
    connection = mc.connect (host = "localhost",
                             user = "pytester",
                             passwd = "monty",
                             db = "company")
except mc.Error as e:
    print("Error %d: %s" % (e.args[0], e.args[1]))
    sys.exit(1)

cursor = connection.cursor()

cursor.execute("SELECT * FROM employee")
print('''Result of "SELECT * FROM employee":''')
result = cursor.fetchall()
for r in result:
    print(r)

cursor.close()
connection.close()
```

It generates the following output:

```
Result of "SELECT * FROM employee":
(0, 'William', 'Shakespeare', 'm', None, datetime.date(1961, 10, 25))
(1, 'Frank', 'Schiller', 'm', None, datetime.date(1955, 8, 17))
(2, 'Jane', 'Wall', 'f', None, datetime.date(1989, 3, 14))
```