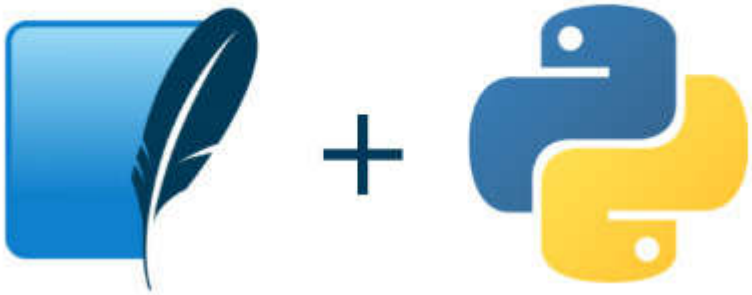
Автор: http://python-3.ru/

Руководство по SQLite Python



Основы SQLite - С чего начать?

В предыдущих статьях мы рассматривали работу с файлами и научились сохранять объекты с доступом по ключу с помощью модуля shelve. При сохранении объектов этот модуль использует возможности модуля pickle для сериализации объекта и модуль anybm для записи получившейся строки по ключу в файл. Если необходимо сохранять в файл просто строки, то можно сразу воспользоваться модулем anybm. Однако если объем сохраняемых данных велик и требуется удобный доступ к ним, то вместо этого модуля лучше использовать базы данных.

Начиная с версии 2.5, в состав стандартной библиотеки Python входит модуль sqlite3, позволяющий работать с базой данных SQLite. Для использования этой базы данных нет необходимости устанавливать сервер, ожидающий запросы на каком-либо порту, т.к. SQLite напрямую работает с файлом базы данных.

Необходимо заметить, что база данных SQLite не предназначена для проектов, предъявляющих требования к защите данных и разграничению прав доступа для нескольких пользователей. Тем не менее, для небольших проектов SQLite является хорошей заменой полноценной базы данных.

Так как SQLite входит в состав стандартной библиотеки Python, мы на некоторое время отвлечемся от изучения языка Python и рассмотрим особенности использования языка SQL (Structured Query Language - структурированный язык запросов) применительно к базе данных SQLite. Для выполнения SQL-запросов мы воспользуемся программой sqlite3.exe, позволяющей работать с SQLite из командной строки. Со страницы www.sqlite.org/download.html загружаем архив sqlite-3_6_23.zip (или тот кто на время прочтения статьи является актуальным), а затем распаковываем его в текущую папку. Далее копируем файл sqlite3.exe а каталог, с которым будем в дальнейшем работать.

Создание и открытие базы данных SQLite в Python

Для создания и открытия базы данных используется функция connect(). Функция имеет следующий формат:

```
connect(database[, timeout][, isolation_level][, detect_types][, factory])
```

В параметре database указывается абсолютный или относительный путь к базе данных. Если база данных не существует, то она просто открывается без удаления имеющихся данных. Вместо пути к базе данных можно указать значение :memory:, которое означает, что база данных будут удалены.

Все остальные параметры являются необязательными и могут быть указаны в произвольном порядке путем присвоения значения названию параметра. Необязательный параметр timeout задает время ожидания снятия блокировки с открываемой базы данных. По умолчанию значение параметра timeout равно пяти секундам. Предназначение остальных параметров мы рассмотрим немного позже.

Функция conntect() возвращает объект соединения, с помощью которого осуществляется вся дальнейшая работа с базой данных. Если открыть базу данных не удалось, то возбуждается исключение. Соединение закрывается, когда вызывается метод close() объекта соединения. В качестве примера откроем и сразу закроем базу данных testdb.db, расположенную в текущем рабочем каталоге.

```
>>> import sqlite3
>>> con = sqlite3.connect("testdb.db")
>>>
>>> con.close()
```

После покупку дома или квартире в так называемом "белом" варианте, вам еще предстоит много над ним поработать для создания отличного вида. Можно заказать электромонтажные работы по доступной цене и высокая качество работы. Не стоит самому браться за электромонтаж если у вас нет опыта в этом дела, стоит задуматься про ваше здоровье и здоровье ваших близких.

Доступ к базе данных SQLite из Python

Модуль sqlite3, входит в состав стандартной библиотеки Python, начиная с версии 2.5, и в дополнительной установке не нуждается. Если необходимо получить доступ к SQLite в предыдущих версиях Python, то следует воспользоваться модулем pysqlite. Этот модуль не входит в состав стандартной библиотеки, поэтому его придется устанавливать отдельно.

Для работы с базами данных в языке Python существует единый интерфейс доступа. Все разработчики модулей, осуществляющих связь базы данных с Python, должны придерживаться спецификации DB-API (DataBase Application Program Interface). Это спецификация более интересна для разработчиков модулей, чем для прикладных программистов, поэтому мы не будем ее подробно рассматривать.

Модуль sqlite3 поддерживает спецификацию DB-API 2.0, а так же предоставляет некоторые нестандартные возможности. Поэтому, изучив методы и атрибуты этого модуля, вы получите достаточно подробное представление о спецификации DB API 2.0 и сможете в дальнейшем работать с другой базой данных. Получить номер спецификации, поддерживаемой модулем, можно с помощью атрибута apilevel:

```
>>> import sqlite3
>>> sqlite3.apilevel
'2.0'
```

Получить номер версии используемого модуля sqlite3 можно с помощью атрибутов sqlite_version и sqlite_version_info. Атрибут sqlite_version возвращает номер версии в виде строки, а атрибут sqlite_version_info в виде кортежа из трех чисел. Пример:

```
>>> sqlite3.sqlite_version
'3.5.9'
>>> sqlite3.sqlite_version_info
(3, 5, 9)
```

Согласно спецификации DB-API 2.0 последовательность работы с базой данных выглядит следующим образом:

1. Производится подключение к базе данных с помощью функции connect(). Функция возвращает объект соединения, с помощью которого осуществляется дальнейшая работа с базой данных.

2. Создается объект-курсора.

3. Выполняется SQL-запросы и обрабатывается результаты. Перед выполнением первого запроса, который изменяет записи (INSERT, REPLACE, UPDATE и DELETE), автоматически запускается транзакция.

4. Завершается транзакция или отменяются все изменения в рамках транзакции.

5. Закрывается объекты курсор.

6. Закрывается соединение с базой данных.

Про любой модуль, язык программирования можно узнать подробней на всемирной энциклопедии Wikipedia. Данный ресурс уже подтвердил свою необходимость в повседневной жизни многих студентов, ученых и простых людей.

Выполнение запроса в SQLite3 через Python

Согласно спецификации DB-API 2.0 после создания объекта соединения необходимо создать объект-курсор. Все дальнейшие запросы должны производиться через этот объект. Создание объекта-курсора производится с помощью метода cursor(). Для выполнения запроса к базе данных предназначены следующие методы объекта-курсора:

close() - закрывает объект-курсор

executescript(<Запросы SQL через точку с запятой>) - выполняет несколько SQL запросов за один раз. Если в процессе выполнения запросов возникает ошибка, то метод возбуждается исключение. В качестве примера создадим базу данных и три таблицы в ней:

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

import sqlite3

con = sqlite3.connect("catalog.db")
cur = con.cursor()    # Создаем объект-курсор

sql = """\
CREATE TABLE user (
    id_user INTEGER PRIMARY KEY AUTOINCREMENT,
    email text,
    passw text
);

CREATE TABLE rubr (
    id_rubr INTEGER PRIMARY KEY AUTOINCREMENT,
    name_rubr TEXT
);
```

```
CREATE TABLE site (
    id_site INTEGER PRIMARY KEY AUTOINCREMENT,
    is_user INTEGER,
    id_rubr INTEGER,
    url TEXT,
    title TEXT,
    iq INTEGER
);
"""

try:    # обработка исключения
    cur.executescript(sql) # Выполняем SQL-запрос
except sqlite.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
cur.close()     # Закрываем объект-курсора
con.close()     # Закрываем соединение
raw_input()
```

Сохраняем код в файл, а затем запускаем его с помощью двойного щелка на значке файла. Обратите внимание на то, что мы работаем с кодировкой UTF-8. Это кодировка по умолчанию используется в SQLite.

execute(<Запрос-SQL[, <Значение>]>) - выполняет один SQL-запрос. Если в процессе выполнения запроса возникает ошибка, то метод возбуждает исключение. Добавим пользователя в таблицу user:

```
# -*- coding: utf-8 -*-
import sqlite3
con = sqlite3.connect("catalog.db")
cur = con.cursor()     # Создаем объект-курсор
sql = """\
INSERT INTO user (email, passw) VALUES ('admin@python-3.ru', 'TypA12ParoLi')
"""
try:    # обработка исключения
    cur.executescript(sql) # Выполняем SQL-запрос
except sqlite.DatabaseError, err:
    print u"Ошибка:", err
else:
    print u"Запрос успешно выполнен"
    con.commit()     # Завершаем транзикцию
cur.close()     # Закрываем объект-курсора
con.close()     # Закрываем соединение
raw_input()
```

Это руководство по программированию в **Python используя базы данных SQLite**. Оно покрывает основы программирования **SQLite** с помощью языка Python.

Необходимые условия

Чтобы работать с примерами их статьи, мы должны иметь установленный в системе язык Python, **базу данных SQLite**, привязку к языку pysqlite и инструмент командной строки sqlite3. Если мы имеем Python 2.5+, тогда нам необходимо только установить инструмент командной строки sqlite3. Как библиотека SQLite, так и привязка к языку pysqlite встроены в язык Python.

```
$ python
```

```
Python 2.7.3 (default, Jan  2 2013, 16:53:07)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> sqlite3.version
'2.6.0'
>>> sqlite3.sqlite_version
'3.7.13'
```

В оболочке, мы запускаем интерактивный интерпретатор Python. Мы можем проверить версию Python. В нашем случае, это Python 2.7.3. sqlite.version – это версия pysqlite (2.6.0), которая является привязкой языка Python к базе данных SQLite. Sqlite3.sqlite_version даёт нам версию библиотеки базы данных SQLite. В нашем случае, версией является 3.7.13.

Сейчас мы собираемся использовать инструмент командной строки **sqlite3**, чтобы создать новую базу данных.

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

Мы предоставляем параметр для sqlite3 tool; test.db – это имя базы данных. Это файл на нашем диске. Если он присутствует, то он открывается. Если нет, он создаётся.

```
sqlite> .tables
sqlite> .exit
$ ls
test.db
```

Команда .tables выдает список таблиц в базе данных test.db. В настоящее время нет таблиц. Команда .exit завершает интерактивную сессию инструмента командной строки sqlite3. Unix-команда ls показывает содержание текущей рабочей папки. Мы можем видеть файл test.db. Все данные будут хранить в этом единственном файле.

Версия

В первом примере кода, мы получим версию базы данных SQLite.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = None

try:
    con = lite.connect('test.db')
    cur = con.cursor()
    cur.execute('SELECT SQLITE_VERSION()')
    data = cur.fetchone()
    print "SQLite version: %s" % data

except lite.Error, e:
    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:
```

```
    if con:
        con.close()
```

В скрипте Python выше, мы подключаемся к предварительно созданной базе данных test.db. Мы выполняем SQL-запрос, который возвращает версию **базы данных SQLite**.

```
import sqlite3 as lite
```

Модуль **sqlite** используется для работы с базой данных SQLite.

```
con = None
```

Мы инициализируем переменную con как None. В случае, если бы мы не смогли произвести соединение с базой данных (к примеру, диск переполнен), мы не имели бы определенную переменную соединения. В конечном счёте, это приведёт к ошибке.

```
con = lite.connect('test.db')
```

Здесь мы соединяемся с базой данных test.db. Метод connect() возвращает объект соединения.

```
cur = con.cursor()
cur.execute('SELECT SQLITE_VERSION()')
```

Мы получаем объект указателя на соединения. Мы вызываем метод execute() и выполняем SQL запрос.

```
data = cur.fetchone()
```

Мы получаем данные. С этого момента мы извлекаем только одну запись и вызываем метод **fetchone**().

```
print "SQLite version: %s" % data
```

Мы выводим данные, которые мы получали в консоль.

```
except lite.Error, e:
    print "Error %s:" % e.args[0]
    sys.exit(1)
```

В случае исключения, мы выдаем сообщение об ошибке и выходим из скрипта с кодом ошибки 1.

```
finally:
    if con:
        con.close()
```

Финальным шагом, мы освобождаем ресурсы закрыв соединение.

Во втором примере, мы снова получаем версию базы данных SQLite. На этот раз мы будем использовать ключевое слово with.

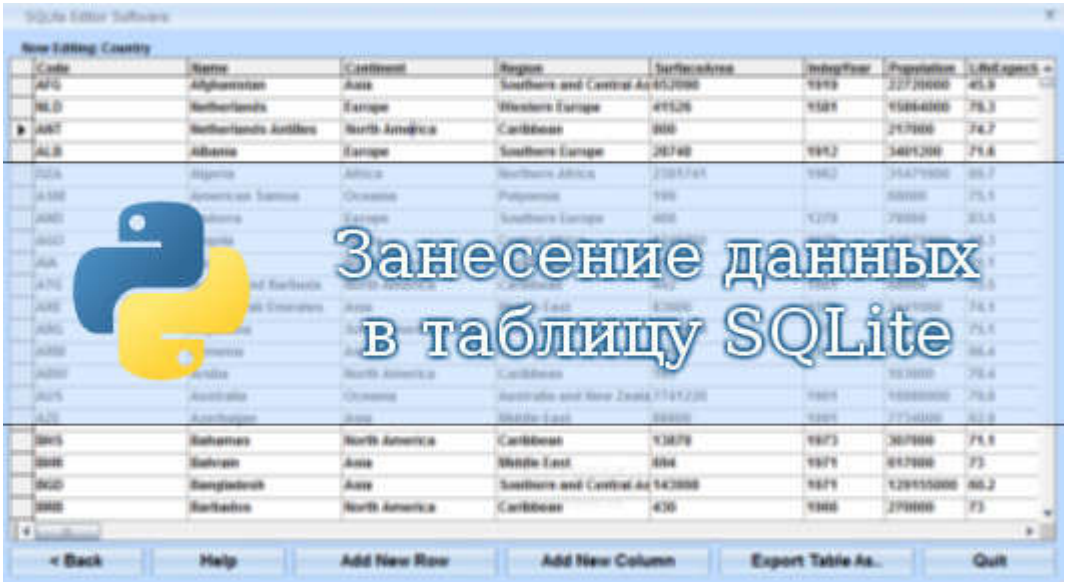
```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute('SELECT SQLITE_VERSION()')
    data = cur.fetchone()
    print "SQLite version: %s" % data
```

Скрипт возвращает текущую версию базы данных SQlite. С использованием ключевого слова with, код является более компактным.

```
with con:
```

С помощью ключевого слова with, интерпретатор Python автоматически освобождает ресурсы. К тому же, это обеспечивает лучшую обработку ошибок.

Занесение данных в таблицу SQLite [Часть 2]



Мы создадим таблицу Cars и внесем несколько строк данных в неё.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

# Подключаемся к базе данных
con = lite.connect('test.db')

with con:
    cur = con.cursor()
    # Создаем таблицу
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
    # Вносим данные
    cur.execute("INSERT INTO Cars VALUES(1, 'Audi', 52642)")
    cur.execute("INSERT INTO Cars VALUES(2, 'Mercedes', 57127)")
    cur.execute("INSERT INTO Cars VALUES(3, 'Skoda', 9000)")
    cur.execute("INSERT INTO Cars VALUES(4, 'Volvo', 29000)")
    cur.execute("INSERT INTO Cars VALUES(5, 'Bentley', 350000)")
    cur.execute("INSERT INTO Cars VALUES(6, 'Citroen', 21000)")
    cur.execute("INSERT INTO Cars VALUES(7, 'Hummer', 41400)")
    cur.execute("INSERT INTO Cars VALUES(8, 'Volkswagen', 21600)")
```

Данный скрипт создаёт таблицу Cars и вставляет 8 строк в таблицу.

```
cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
```

Этот SQL-запрос создает новую таблицу Cars. Таблица имеет три столбца.

```
cur.execute("INSERT INTO Cars VALUES(1,'Audi',52642)")
```

```
cur.execute("INSERT INTO Cars VALUES(2,'Mercedes',57127)")
```

Эти две строки добавляют в таблицу данные о двух машин. С использованием ключевого слова **with**, изменения фиксируются автоматически. В противном случае, мы должны были зафиксировать их вручную.

```
sqlite> .mode column
sqlite> .headers on
```

Мы проверяем записанные данные консольным инструментом sqlite3. В первую очередь, мы изменяем способ, которым данные отображаются в консоли. Мы используем режим столбцов и включаем заголовки.

```
sqlite> SELECT * FROM Cars;
Id          Name        Price
----------  ----------  ----------
1           Audi        52642
2           Mercedes    57127
3           Skoda       9000
4           Volvo       29000
5           Bentley     350000
6           Citroen     21000
7           Hummer      41400
8           Volkswagen  21600
```

Это данные, которые мы внесли в таблицу Cars.

Мы собираемся создать такую же таблицу, на этот раз используя удобный **метод executemany**().

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

cars = (
    (1, 'Audi', 52642),
    (2, 'Mercedes', 57127),
    (3, 'Skoda', 9000),
    (4, 'Volvo', 29000),
    (5, 'Bentley', 350000),
    (6, 'Hummer', 41400),
    (7, 'Volkswagen', 21600)
)

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Cars")
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
    cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
```

Скрипт удаляет таблицу Cars, если она существует и создает ее заново.

```
cur.execute("DROP TABLE IF EXISTS Cars")
cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
```

Первый SQL-запрос удаляет таблицу Cars, если она существует. Второй SQL запрос создает таблицу Cars.

```
cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
```

Мы вводим 8 строк в таблицу, используя метод **executemany()**. Первый параметр этого метода – это сам SQL запрос. Второй параметр – это данные в форме кортежа. Это более безопасный и уверенный способ внесения данных в базу.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

try:
    con = lite.connect('test.db')
    cur = con.cursor()
    cur.executescript("""
        DROP TABLE IF EXISTS Cars;
        CREATE TABLE Cars(Id INT, Name TEXT, Price INT);
        INSERT INTO Cars VALUES(1,'Audi',52642);
        INSERT INTO Cars VALUES(2,'Mercedes',57127);
        INSERT INTO Cars VALUES(3,'Skoda',9000);
        INSERT INTO Cars VALUES(4,'Volvo',29000);
        INSERT INTO Cars VALUES(5,'Bentley',350000);
        INSERT INTO Cars VALUES(6,'Citroen',21000);
        INSERT INTO Cars VALUES(7,'Hummer',41400);
        INSERT INTO Cars VALUES(8,'Volkswagen',21600);
        """)

    con.commit()

except lite.Error, e:
    if con:
        con.rollback()

    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:
    if con:
        con.close()
```

В вышеприведённом скрипте, мы пересоздаём таблицу Cars, используя метод **executescript()**.

```
cur.executescript("""
    DROP TABLE IF EXISTS Cars;
    CREATE TABLE Cars(Id INT, Name TEXT, Price INT);
    INSERT INTO Cars VALUES(1,'Audi',52642);
    INSERT INTO Cars VALUES(2,'Mercedes',57127);
...
```

Метод **executescript()** разрешает нам выполнять целый SQL-код в один шаг.

```
con.commit()
```

Без ключевого слова with, изменения должны вступить в силу благодаря использованию метода **commit()**.

```
except lite.Error, e:
```

```
    if con:
        con.rollback()

    print "Error %s:" % e.args[0]
    sys.exit(1)
```

В случае ошибки, изменения откатятся назад и сообщение об ошибке выведется в терминале.

Получить ID последнего внесения в базу

Иногда, нам необходимо определить id последней вставленной строки. В **Python SQLite**, мы используем атрибут **lastrowid**объекта указателя.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect(':memory:')

with con:
    cur = con.cursor()
    cur.execute("CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Tom');")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Rebecca');")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Jim');")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Robert');")

    lid = cur.lastrowid
    print "The last Id of the inserted row is %d" % lid
```

Мы создаём таблицу Friends. ID автоматически прибавился.

```
cur.execute("CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT);")
```

В **SQLite**, колонка **INTEGER PRIMARY KEY** прибавляется автоматически. Существует так же параметр **AUTOINCREMENT**. Для **INTEGER PRIMARY KEY AUTOINCREMENT** используются слегка разные алгоритмы для создания ID.

```
cur.execute("INSERT INTO Friends(Name) VALUES ('Tom');")
cur.execute("INSERT INTO Friends(Name) VALUES ('Rebecca');")
cur.execute("INSERT INTO Friends(Name) VALUES ('Jim');")
cur.execute("INSERT INTO Friends(Name) VALUES ('Robert');")
```

Когда используется авто-инкремент, мы должны явно изложить имена столбцов, пропуская тот, к которому происходит авто-добавление. Четыре оператора вводят четыре строки в таблицу Friends.

```
lid = cur.lastrowid
```

Используя **lastrowid**, мы получаем **ID последней вставленной строки**.

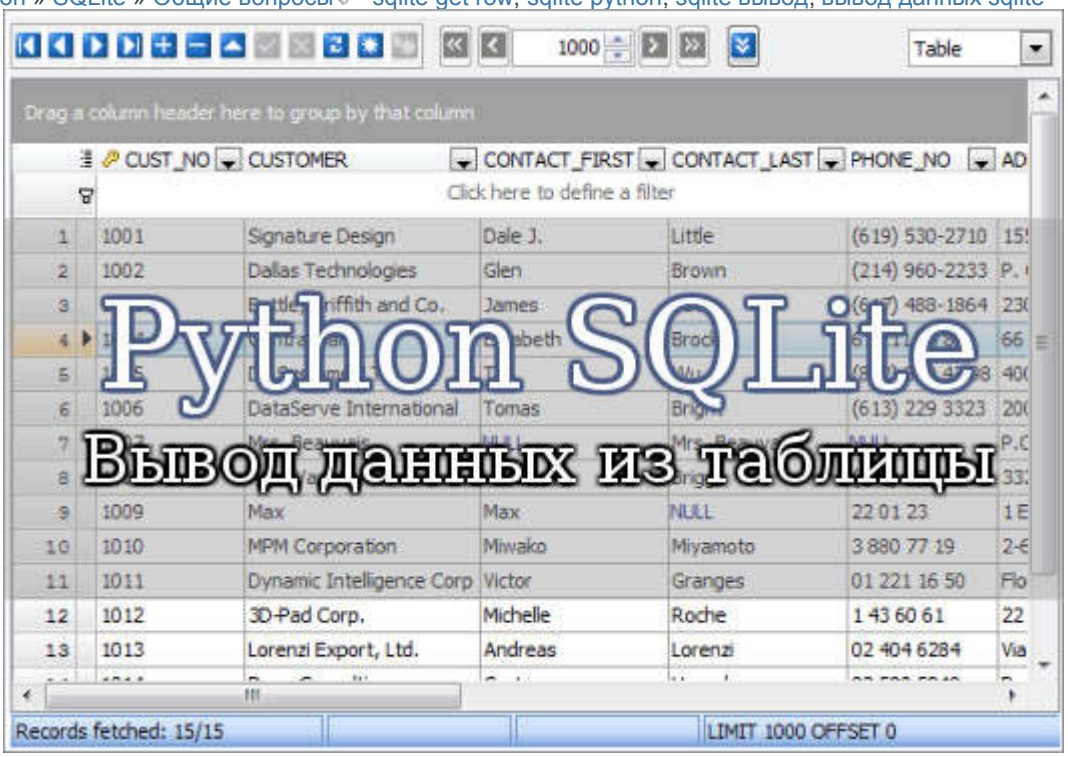
```
$ ./lastrow.py
The last Id of the inserted row is 4
```

После выполнения скрипта мы видим последний ID добавления.

Вывод данных из таблицы SQLite [Часть 3]

13 августа 2015 г. Archy Просмотров: 4553 RSS Обсудить

Примеры Python » SQLite » Общие вопросы sqlite get row, sqlite python, sqlite вывод, вывод данных sqlite



В предыдущей статье [Часть 2] мы рассмотрели моменты создания таблицы в базу данных и внесения данных в эту таблицу. В данной статье покажем пример как получить данные из таблицы.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sqlite3 as lite
import sys

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute("SELECT * FROM Cars")
    rows = cur.fetchall()

    for row in rows:
        print row
```

В этом примере, мы извлекаем все данные из таблицы Cars.

```
cur.execute("SELECT * FROM Cars")
```

Этот SQL запрос выбирает все данные из таблицы Cars.

```
rows = cur.fetchall()
```

Метод **fetchall()** получает все записи. Он возвращает результирующий набор. Технически, это кортеж. Каждый из внутренних кортежей представляет строку в таблице.

```
for row in rows:
    print row
```

Мы выводим данные в консоль, строка за строкой.

```
$ ./retrieveall.py
(1, u'Audi', 52642)
```

```
(2, u'Mercedes', 57127)
(3, u'Skoda', 9000)
(4, u'Volvo', 29000)
(5, u'Bentley', 350000)
(6, u'Citroen', 21000)
(7, u'Hummer', 41400)
(8, u'Volkswagen', 21600)
```

Это пример данных.

Получить все данные сразу нельзя, можно только в строковом виде.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect('test.db')
with con:
    cur = con.cursor()
    cur.execute("SELECT * FROM Cars")

    while True:
        row = cur.fetchone()

        if row == None:
            break

        print row[0], row[1], row[2]
```

В этом скрипте мы соединяемся с базой данных и получаем строки таблицы Cars одну за одной.

Данный скрипт можно запускать и у себя на сервере, будь то личный компьютер или VPS от host virtual server на mirohost.net которые знамениты своим качеством предоставления услуг и быстрой технической поддержкой.

```
while True:
```

Мы обращаемся к данным из цикла «While». Когда мы читаем последнюю строку, цикл завершается.

```
row = cur.fetchone()
if row == None:
    break
```

Метод **fetchone()** возвращает следующую строку из таблицы. Если данных там больше не осталось, он возвращает None. В этом случае, мы прерываем цикл.

```
print row[0], row[1], row[2]
```

Данные возвращаются в форме кортежа. Здесь мы выбираем записи из кортежа. Первая – это ID, вторая – название машины и третья – цена машины.

```
$ ./retrieveonebyone.py
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
```

```
7 Hummer 41400
8 Volkswagen 21600
```

Такой результат скрипта мы получили после его выполнения.

Загрузка изображения в базу данных SQLite

В этой статье, мы собираемся вставить данные изображения в базу данных SQLite. Следует отметить, что некоторые программисты выступают против помещения изображений в базы данных. Здесь мы только покажем, как делать это. Мы не станем останавливаться на технических проблемах того, сохранять ли изображения в базах данных или нет. Это применяется в зависимости от ситуации, когда эффективнее сохранять в базе чем в файлах.

```
sqlite> CREATE TABLE Images(Id INTEGER PRIMARY KEY, Data BLOB);
```

Для этого примера, мы создаём новую таблицу, называемую Images. Для изображений мы используем тип данных BLOB, который расшифровывается как Binary Large Objects.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# Подключаем библиотеки
import sqlite3 as lite
import sys
# Функция открытия изображения в бинарном режиме
def readImage(filename):
    try:
        fin = open(filename, "rb")
        img = fin.read()
        return img

    except IOError, e:
        # В случае ошибки, выводим ее текст
        print "Error %d: %s" % (e.args[0],e.args[1])
        sys.exit(1)

    finally:
        if fin:
            # Закрываем подключение с файлом
            fin.close()

try:
    # Открываем базу данных
    con = lite.connect('test.db')
    cur = con.cursor()
    # Получаем бинарные данные нашего файла
    data = readImage("woman.jpg")
    # Конвертируем данные
```

```
    binary = lite.Binary(data)
    # Готовим запрос в базу
    cur.execute("INSERT INTO Images(Data) VALUES (?)", (binary,) )
    # Выполняем запрос
    con.commit()

# В случаи ошибки выводим ее текст.
except lite.Error, e:
    if con:
        con.rollback()

    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:
    if con:
    # Закрываем подключение с базой данных
        con.close()
```

В этом скрипте, мы читаем изображение из текущей папки в бинарном режиме и записываем его в таблицу Images базы данных SQLite **test.db**.

```
try:
    fin = open("woman.jpg", "rb")
    img = fin.read()
    return img
```

Мы **читаем бинарные данные** из изображения. Мы имеем JPG-изображение, названное woman.jpg.

```
binary = lite.Binary(data)
```

Данные изображения конвертируются в данные объекта SQLite Binary.

```
cur.execute("INSERT INTO Images(Data) VALUES (?)", (binary,) )
```

Данный SQL запрос выполняется, чтобы вставлять изображение в базу данных.

Чтение изображения из SQLite

В этой статье, мы собираемся выполнять обратную операцию. Мы будем читать изображение из таблицы базы данных. В прошлой статье мы записали данные изображения в таблицу, теперь мы превратим эти данные обратно в изображение.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

def writeImage(data):
    try:
        fout = open('woman2.jpg','wb')
        fout.write(data)

    except IOError, e:
        print "Error %d: %s" % (e.args[0], e.args[1])
        sys.exit(1)
```

```
    finally:
        if fout:
            fout.close()

try:
    con = lite.connect('test.db')
    cur = con.cursor()
    cur.execute("SELECT Data FROM Images LIMIT 1")
    data = cur.fetchone()[0]
    writeImage(data)

except lite.Error, e:
    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:
    if con:
        con.close()
```

Мы читаем данные изображения из таблицы Images и пишем их в другой файл, который мы называем woman2.jpg. В базу данных можно сохранять **изображения для капч**, используя небольшой список русских слов (как это делает Яндекс) и выводить их для пользователя. Можно скрипт капчи скачать и использовать уже готовый для своих проектах.

```
try:
    fout = open('woman2.jpg','wb')
    fout.write(data)
```

Мы открываем двоичный файл в режиме записи. Данные из базы данных записываются в файл.

```
cur.execute("SELECT Data FROM Images LIMIT 1")
data = cur.fetchone()[0]
```

Эти две строки выбирают и получают данные из таблицы Images. Мы достаём двоичные данные из первой строки.

Метаданные в SQLite

Метаданные – это информация о данных в базе данных. Метаданные в SQLite содержат в себе информацию о таблицах и столбцах, в которых мы храним данные. Количество строк под воздействием оператора SQL – это метаданные. Количество строк и столбцов, возвращаемые в результирующий набор, также относятся к метаданным. Метаданные в SQLite могут быть получены с использованием команды PRAGMA. Объекты

SQLite могут иметь атрибуты, которые являются метаданными. Наконец, мы также можем достать определённые метаданные от запроса таблицы sqlite_master системы SQLite.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute('PRAGMA table_info(Cars)')
    data = cur.fetchall()

    for d in data:
        print d[0], d[1], d[2]
```

В этом примере, мы пускаем в обращение команду **PRAGMA table_info(tableName)**, чтобы получить некоторую метаинформацию о нашей таблице Cars.


```
cur.execute('PRAGMA table_info(Cars)')
```

Команда **PRAGMA table_info(tableName)** возвращает одну строку для каждого столбца в таблице Cars. Столбцы в результирующем наборе содержат в себе порядковый номер столбца, название столбца, тип данных, может ли столбец быть NULL или нет, а также значение по умолчанию для столбца.

```
for d in data:
    print d[0], d[1], d[2]
```

Из предоставленной информации, мы печатаем порядковый номер столбца, название столбца и тип данных столбца.

```
$ ./colnames1.py
0 Id INT
1 Name TEXT
2 Price INT
```

Выходная информация примера

Далее мы напечатаем все строки из таблицы Cars с именами столбцов.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute('SELECT * FROM Cars')

    col_names = [cn[0] for cn in cur.description]
```

```
    rows = cur.fetchall()
    print "%s %-10s %s" % (col_names[0], col_names[1], col_names[2])

    for row in rows:
        print "%2s %-10s %s" % row
```

Мы печатаем содержание таблицы Cars в консоль. Теперь, мы также включаем имена столбцов. Записи выравниваются с именами столбцов.

```
col_names = [cn[0] for cn in cur.description]
```

Мы получаем имена столбцов из свойства **description** объекта указателя.

```
print "%s %-10s %s" % (col_names[0], col_names[1], col_names[2])
```

Эта строка печатает три имени столбцов таблицы Cars.

```
for row in rows:
    print "%2s %-10s %s" % row
```

Мы печатаем строки, используя **цикл for**. Данные выравниваются с именами столбцов.

```
$ ./colnames2.py
Id Name       Price
 1 Audi       52642
 2 Mercedes   57127
 3 Skoda      9000
 4 Volvo      29000
 5 Bentley    350000
 6 Citroen    21000
 7 Hummer     41400
 8 Volkswagen 21600
```

Выходная информация

В нашем последнем примере, связанном с метаданными, мы перечислим все таблицы в **базе данных test.db**.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sqlite3 as lite
import sys

con = lite.connect('test.db')

with con:
    cur = con.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table'")

    rows = cur.fetchall()

    for row in rows:
        print row[0]
```

Пример кода печатает все доступные в текущей базе данных таблицы в терминал.

```
cur.execute("SELECT name FROM sqlite_master WHERE type='table'")
```

Имена таблиц хранятся внутри таблицы **sqlite_master** системы.

```
$ ./listtables.py
Images
sqlite_sequence
Salaries
Cars
```

Это были таблицы в моей системе.