блог JOHN_16

Здесь я планирую размещать небольшие статьи на интересующие меня темы.

Популярные сообщения

Python: модуль sqlite3. Перевод документации с примерами.

Python: модуль Queue. Перевод документации. Примеры работы с очередями

Python: модуль subprocess. Перевод документации с комментариями и примерами. Особенности применения.

Python: модуль signal. Перевод документации. Примеры.

Python: модуль timeit - измерение времени выполнения маленьких фрагментов кода. Примеры использования.

Python: модуль optparse. Частичный перевод документации с примером.

Python: Модуль logging. Примеры конфигураций.

Python: модули fnmatch и glob. Перевод документации с комментариями и примерами

Python: модуль SymPy. Пример символьных вычислений.

Как маленькая деталь "убила" мой смартфон

четверг, 24 марта 2011 г.

Python: модуль sqlite3. Перевод документации с примерами.

sqlite3 — DB-API 2.0 интерфейс для баз данных SQLite

Новое в версии 2.5

SQLite это библиотека написанная на языке C, которая предоставляет легковесную, находящуюся на диске базу данных, которой не требуется отдельный серверный процесс и которая позволяет получать доступ используя не стандартные диалекты языка SQL. Некоторые приложения могут использовать SQLite для внутреннего хранения данных. Также возможно сделать прототип приложения используя SQLite и затем перенести код для больших баз данных, таких как PostgreSQL или Oracle.

sqlite3 была написана Gerhard Häring и предоставляет SQL интерфейс совместимый с DB-API 2.0 спецификациями описанными в PEP 249.

Для использования модуля, сначала нужно создать Connection объект олицетворяет базу данных. В данном примере данные будут храниться в файле /tmp/example:

```
conn=sqlite3.connect('/tmp/example')
```

Также можно использовать специальное имя :memory: для создания базы данных в ОЗУ

Создав ранее объект Connection можно создать объект Cursor и вызвать его метод execute() для выполнения SQL команд:

```
c = conn.cursor()
# Создание таблицы
c.execute('''create table stocks (date text, trans text, symbol text, qty real, price real)''')
# Вставка ряда данных
c.execute("""insert into stocks values ('2006-01-05','BUY','RHAT',100,35.14)""")
# Сохранение (commit) изменений
conn.commit()
# Закрытие курсора, в случае если он больше не нужен
c.close()
```

Обычно для выполнения SQL операций необходимо использовать значения из переменных Python. Не рекомендуется собирать запрос используя строковые операции Питона потому что это может быть не безопасным, это делает программу уязвимой к инъекционным SQL атакам.

Вместо этого используйте DB-API подстановку параметров. Вставьте символ ? в качестве заполнителя в месте где вы хотите использовать значение и предоставьте кортеж значений в качестве второго аргумента метода курсора execute(). (Другие модули баз данных могут использовать другие заполнители, такие как %s или :1). Пример:

```
# Никогда не делайте так - это не безопасно!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)
# Вместо этого делайте так
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)
# Еще пример
for t in [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00)]:
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

Что бы получить данные, после выполнения оператора SELECT, можно обработать курсор как итератор, вызвать метод fetchone() что бы получить одиночный ряд, или вызвать fetchall() что бы получить список соответствующих строк. Например:

```
print '1st example'
c.execute('select * from stocks order by price')
```

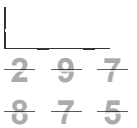

Поиск по этому блогу

Поиск

Теги

python (19) Программирование (16) subprocess (2) vscode (2) Музыка (2) факап (2) BOSS (1) Boggart30 (1) Django (1) FL Studio (1) GT-8 (1) Grinders (1) ODROID (1) Queue (1) SQLite (1) Wine (1) Yerasov (1) aiohttp (1) android (1) cx_freeze (1) docker (1) mapswithme (1) miniseed (1) mseed (1) obspy (1) optparse (1) psutil (1) pyserial (1) pysteim (1) pytest (1) python3 (1) pywinauto (1) quiz (1) signal (1) sqlite3 (1) steim (1) timeit (1) разное (1) смартфон (1) схемотехника (1)

Общее количество просмотров страницы



Архив блога

- ► 2018 (3)
- ► 2017 (2)
- ► 2015 (4)
- ► 2014 (5)
- ► 2013 (4)
- ► 2012 (5)
- ▼ 2011 (6)
  - ► августа (1)
  - ► июля (2)
  - ▼ марта (1)
    - Python: модуль sqlite3. Перевод документации с при...
  - ► февраля (2)
- ► 2010 (1)

Постоянные читатели

```
    for row in c:
        print row

    print '2nd example'
    c.execute('select * from stocks order by price')
    while True:
        tmp=c.fetchone()
        if tmp:
            print tmp
        else:
            break

    print '3rd example'
    c.execute('select * from stocks order by price')
    print c.fetchall()
```

Результат:

```
    1st example
    (u'2006-03-28', u'BUY', u'IBM', 1000.0, 45.0)
    (u'2006-04-06', u'SELL', u'IBM', 500.0, 53.0)
    (u'2006-04-05', u'BUY', u'MSOFT', 1000.0, 72.0)
    2nd example
    (u'2006-03-28', u'BUY', u'IBM', 1000.0, 45.0)
    (u'2006-04-06', u'SELL', u'IBM', 500.0, 53.0)
    (u'2006-04-05', u'BUY', u'MSOFT', 1000.0, 72.0)
    3rd example
    [(u'2006-03-28', u'BUY', u'IBM', 1000.0, 45.0), (u'2006-04-06', u'SELL', u'IBM',
    500.0, 53.0), (u'2006-04-05', u'BUY', u'MSOFT', 1000.0, 72.0)]
```

Функции и константы модуля.

sqlite3.**PARSE_DECLTYPES**

Эта константа предназначена для использования с параметром **detect_types** функции connect().
Установка ее позволяет модулю sqlite3 разбирать заявленный тип для каждой возвращаемой колонки. Разбор первого слова объявленного типа, такого как «целочисленное значение первичного ключа» будет воспринято как целочисленное число или для «number(10)» будет воспринято как «число». Далее для выбранной колонки будет просмотрен словарь конвертеров и будет использована та функция конвертера что зарегистрирована для данного типа.

sqlite3.**PARSE_COLNAMES**

Эта константа предназначена для использования с параметром **detect_types** функции connect().
Установка этого параметра позволяет интерфейсу SQLite проанализировать имя каждого возвращаемого столбца. Он будет искать строки сформированные [mytype], и затем решит, что 'mytype' - тип столбца. Он попытается найти запись 'mytype' в словаре преобразователей и затем использует функцию преобразователя что бы возвратить значение. Имя столбца, найденное в Cursor.description, является только первым словом имени столбца, то есть если Вы используете что-то наподобие 'as "x [datetime]"' в SQL-запросе, тогда будет проанализировано все до первого пробела для имени столбца: имя столбца просто было бы "x".

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*,*factory*, *cached_statements*])

Создает соединение с файлом database базы данных SQLite. Возможно использование ":memory:" для открытия соединения с базой данных находящейся в оперативной памяти вместо жесткого диска.

Когда к базе данных получают доступ многократные соединения, и один из процессов изменяет базу данных, база данных SQLite блокируется, до тех пор пока та транзакция не зафиксируется (commit). Параметр timeout определяет, сколько времени соединение должно ждать блокировки, чтобы уйти до возбуждения исключения. Значение по умолчанию для параметра тайм-аута 5.0 (пять секунд).

Описание параметра *isolation_level* приведено в разделе Connection объекта.

SQLite нативно поддерживает только следующие типы данных: TEXT, INTEGER, FLOAT, BLOB, NULL. Если имеется необходимость использовать другие типы, то реализация их поддержки является самостоятельной задачей. Использование параметра *detect_types* и специальных конверторов, зарегистрированных с помощью функции register_convertor(), позволит легко сделать это.

detect_types по умолчанию равен 0 (т.е., обнаружение типов выключено), возможные значения представляют собой любые комбинации PARSE_DECLTYPES и PARSE_COLNAMES.

По умолчанию модуль sqlite3 использует класс Connection для соединений. Тем не менее, возможно разделить на подклассы класс Connection и сделать

connect() используя свой класс, указав его в параметре *factory*.

sqlite3 модуль использует внутренний кэш операторов что бы избежать издержек при парсинге SQL запросов. Что бы явно задать число операторов для кеширования, необходимо задать параметр *cashed_statements*. По умолчанию он равен значению 100.

sqlite3.**register_convertor**(*typename, callable*)

Регистрирует вызываемый объект для преобразования строки байтов из базы данных в специальный тип данных Питона. Вызываемый объект будет вызываться для всех значений базы данных которые являются типа *typename*. Присвойте параметр *detect_type* функции connect() что бы посмотреть как работает детектирование типов. Помните что в случае *typename* и имя типа должны совпадать в вашем запросе.

sqlite3.**register_adapter**(*type, callable*)

Регистрирует вызываемый объект для преобразования специального Питоновского типа данных *type* в один из типов поддерживаемых SQLite. Вызываемый объект *callable* принимает в качестве одиночного параметра значение Питона и должно возвращать значение следующих типов: int, long, float, str (в кодировке UTF-8), unicode, buffer.

sqlite3.**complete_statement**(*sql*)

Возвращает True если строка *sql* содержит одно или более SQL операторов оканчивающихся точкой с запятой. Это не проверяет корректность синтаксиса SQL запроса, только то что не содержится не закрытых строковых символов и выражение заканчивается точкой с запятой.

sqlite3.**enable_callback_tracebacks**(*flag*)

По умолчанию вы не будете получать traceback информацию в функциях определяемых пользователем преобразователях и т.д. Если вы хотите отладить их, то нужно вызвать эту функцию с параметром *flag* равным True. После этого вы будете получать traceback информацию на стандартное устройство вывода ошибок sys.stderr. Использование значения параметра *flag* равным False отключит данную функцию.

Объекты Connection

Class sqlite3.**Connection**

Класс соединения базы данных SQLite имеет следующие атрибуты и методы:

Connection.**isolation_level**

Возвращает или устанавливает текущий уровень изоляции. Принимает значение None для режима автоматического принятия изменений (autocommit) или один из «DEFFERED», «IMMEDIATE», «EXCLUSIVE».

Connection.**cursor**([*cursorClass*])

Параметр *cursorClass* должен быть частным классом курсора который бы расширил sqlite3.Cursor.

Connection.**commit**()

Этот метод фиксирует текущую транзакцию. Если не вызвать этот метод, то все изменения сделанные с момента прошлого вызова commit() не будут видимы другим соединениям.

Connection.**rollback**()

Этот метод отменяет все изменения сделанные прошлым вызовом commit().

Connection.**close**()

Закрытие соединения с базой данных. **Внимание!**Этот метод не вызывает автоматически commit(), поэтмоу для сохранения всех изменений следует сперва вызывать commit().

Connection.**execute**(*sql*[,*parameters*])
Connection.**executemany**(*sql*[,*parameters*])
Connection.**executescript**(*sql_script*)

Это не стандартный ярлык, который создает промежуточный объект курсора вызывая метод курсора. Затем вызывая метод курсора execute/executemany/executescript с заданными параметрами.

Connection.**create_function**(*name, num_params, func*)

Создает пользовательскую функцию с именем name которую потом можно использовать внутри SQL выражения. Параметр *num_params* задает число параметров принимаемой функцией, *func* это вызываемый объект Питона.

Функция может возвращать любой тип данных поддерживаемый SQLite: unicode, str, int, long, float,buffer,None.

Пример:

```
import sqlite3
import md5
def md5sum(t):
return md5.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

Результат:

```
cbd18db4cc2f85cedef654fccc4a4d8
```

Connection.**create_aggregate**(*name, num_params, aggregate_class*)

Создать пользовательскую агрегированную (совокупную) функцию.

Агрегированный класс должен предоставлять метод step, который принимает число параметров *num_params*, и метод finalize, который будет возвращать окончательный результат.

Метод finalize может возвращать любой тип данных поддерживаемых SQLite.

Пример:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (10)")
cur.execute("insert into test(i) values (15)")

cur.execute("select mysum(i) from test")
print cur.fetchall()
```

Результат:

```
[(26,)]
```

Connection.**create_collation**(*name, callable*)

Создает функцию сравнения, параметр *name* – им функции, *callable* – вызываемый объект, которому будут передаваться два строковых параметра. Он возвращать -1 если первое значение меньше второго, 0 если равны и 1 если больше. Помните, что это управляется сортировкой (оператором ORDER BY в SQL запросе), поэтому ваше сравнение может не иметь силы для других SQL операций.

Помните, что все вызываемые объекты принимают параметры в качестве строки кодированной в UTF-8 кодировке. Пример:

```
import sqlite3

alphabet=[chr(i) for i in xrange(97,123)]

def collate_reverse(string1, string2):
    return -cmp(string1, string2)

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)
cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", alphabet)

cur.execute("select x from test order by x")
print 'print alphabet'
print cur.fetchall()

cur.execute("select x from test order by x collate reverse")
print 'print reverse alphabet'
print cur.fetchall()
```

```
con.close()
```

Результат:

```
print alphabet
[(u'a',), (u'b',), (u'c',), (u'd',), (u'e',), (u'f',), (u'g',), (u'h',), (u'i',),
(u'j',), (u'k',), (u'l',), (u'm',), (u'n',), (u'o',), (u'p',), (u'q',), (u'r',),
(u's',), (u't',), (u'u',), (u'v',), (u'w',), (u'x',), (u'y',), (u'z',)]
print reverse alphabet
[(u'z',), (u'y',), (u'x',), (u'w',), (u'v',), (u'u',), (u't',), (u's',), (u'r',),
(u'q',), (u'p',), (u'o',), (u'n',), (u'm',), (u'l',), (u'k',), (u'j',), (u'i',),
(u'h',), (u'g',), (u'f',), (u'e',), (u'd',), (u'c',), (u'b',), (u'a',)]
```

Что бы удалить действие сравнения необходимо вызвать create_collation со значением параметра *callable* равным None:

```
con.create_collation("reverse",None)
```

Connection.**interrupt**()

Вызов этого метода из различного потока прервет любые запросы, которые могут выполняться на соединении. Запрос будет отменен и вызывающая сторона получит исключение.

Connection.**set_authorizer**(*authorizer_callback*)

Эта подпрограмма регистрирует функцию обратного вызова(callback). Обратный вызов вызывается для каждой попытки получить доступ к столбцу таблицы в базе данных. Обратный вызов должен возвратить SQLITE_OK, если доступ предоставлен, SQLITE_DENY, если весь SQL-оператор должен быть прерван с ошибкой и SQLITE_IGNORE, если столбец должен быть обработан как значение NULL. Эти константы доступны в sqlite3 модуле.

Первый параметр, передаваемый в функцию обратного вызова, означает, какого рода операция должна быть авторизированна. Второй и третий параметры должны быть аргументы или None в зависимости от первого аргумента. Четвертый аргумент это название базы данных ("main","temp" и т.д.) если это применимо. Пятый аргумент это имя внутреннего триггера or view that is responsible for the access attempt или None если это попытка доступа из самого SQL запроса.

Connection.**set_progress_handler**(*handler,n*)

Новое в версии 2.6

Эта подпрограмма регистрирует функцию обратного вызова(callback). Обратный вызов вызывается каждые *n* инструкций виртуальной машины SQLite. Это полезно если вы хотите быть вызванными из SQLite во время продолжительных операций, например для обновления GUI. Присвоение параметру значения None отключит работу данного метода.

Connection.**enable_load_extension**(*enabled*)

Новое в версии 2.7

Этот метод позволяет или не позволяет движку SQLite загружать библиотеки расширений, которые могут определять новые функции, агрегаты или реализации новых виртуальных таблиц. Хорошо известно расширение полнотекстового поиска поставляемое с SQLite.
Пример смотри ниже.

Connection.**load_extension**(*path*)

Новое в версии 2.7

Загрузка библиотеки расширений SQLite. Перед использованием необходимо включить поддержку расширений с помощью enable_load_extension(). Пример:

```
# пример работает если установлено дополнение полнотектсового поиска
import sqlite3
con = sqlite3.connect(":memory:")
# enable extension loading
con.enable_load_extension(True)
# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")
# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")
# disable extension laoding again
con.enable_load_extension(False)
# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
        insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli
peppers cheese tomatoes');
        insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin
onions garlic celery');
        insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli
```

```
cheese onions flour');
        insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin
sugar flour butter');
        """)
for row in con.execute("select rowid, name, ingredients from recipe where name
match 'pie'"):
    print row
```

Connection.**row_factory**

Вы можете изменить этот атрибут на вызываемые объект, который принимает курсор и исходную строку как кортеж и возвращает реальную строку результата. Таким образом, Вы можете реализовать усовершенствованные способы возврата результатов, такие как возврат объекта, который может также получить доступ к столбцам по имени. Пример:

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

workers=(('John',3,10000),('Nik',5,15000))

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("create table test(name text, experience integer, screw integer)")
cur.executemany('insert into test(name,experience,screw) values (?,?,?)',workers)
cur.execute('select * from test')

for item in cur.fetchall():
    print 'worker %s have %i years of experience and %i $ of screw'%
(item['name'],item['experience'],item['screw'])
```

Результат:

```
worker John have 3 years of experience and 10000 $ of screw
worker Nik have 5 years of experience and 15000 $ of screw
```

Если возвращаемого кортежа недостаточно, и вы хотите основанный на имени доступ к столбцам, то вы должны рассматривать настройку **row_factory** как высоко-оптимизированную sqlite3.Row. Тип Row предоставляет доступ к столбцам, как на основе индексов, так и на основе чувствительным к регистру именам почти без издержек памяти. Это, возможно, даже лучшее решение чем собственный словарь или даже решение основанное на *db_row*.

Connection.**text_factory**

Используя этот атрибут возможно контролировать какие объекты будут возвращены для типа данных TEXT. По умолчанию этот атрибут установлен в unicode и sqlite3 модуль возвратит объекты Unicode для типа данных TEXT. Если необходимо возвратить строки байтов вместо этого, то нужно установить значение str.

По причинам эффективности существует также другой способ возвращать объекты Unicode только для не-ASCII данных и строк байтов. Для его активации установите этот атрибут в значение sqlite3.OptimizedUnicode.

Также возможно присвоение ему любого вызываемого объекта принимающего на входе одиночную строку и возвращающую результирующий объект.

Пример ниже демонстрирует варианты применения параметра. Для наглядности работы функции mystr изменяющей выходную кодировку строки использовалась консоль ОС Windows XP:

```
# -*- encoding: utf-8 -*-
import sqlite3,sys

def mystr(input):
    # здесь можно сделать все что угодно
    return unicode(input,'u8').encode('cp866')
# подготавливаем данные
data=['Россия'.decode('u8'),'England']

# создаем таблицу, вносим данные
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table country(name text)")
for i in data:
    cur.execute('insert into country values (?)',(i,))
```

```
print u'По умолчанию все объекты возвращаются типом Unicode'
cur.execute('select * from country')
for i in cur.fetchall():
    print i[0],type(i[0])

print '-'*10
print u'Выходные данный типа str '
con.text_factory=str
cur.execute('select * from country')
for i in cur.fetchall():
    print i[0],type(i[0])

print '-'*10
print u'Использование преобразующей функции mystr'
con.text_factory=mystr
cur.execute('select * from country')
for i in cur.fetchall():
    print i[0],type(i[0])

print '-'*10
print u'Использование OptimizedUnicode '
con.text_factory=sqlite3.OptimizedUnicode
cur.execute('select * from country')
for i in cur.fetchall():
    print i[0],type(i[0])
```

Результат:

```
По умолчанию все объекты возвращаются типом Unicode
Россия <type 'unicode'>
England <type 'unicode'>
----------
Выходные данный типа str
¦a¦-ТБТБ¦¬ТП <type 'str'>
England <type 'str'>
----------
Использование преобразующей функции mystr
Россия <type 'str'>
England <type 'str'>
----------
Использование OptimizedUnicode
Россия <type 'unicode'>
England <type 'str'>
```

Connection.**total_changes**

Возвращает общее число рядов которые были модифицированы, вставлены, или удалены с тех пор как было открыто соединение с базой данных.

Connection.**iterdump**

Новое в версии 2.6

Возвращает итератор к дампу базы данных в SQL текстовом формате. Полезна для сохранения базы данных в памяти для последующего восстановления. Эта функция обеспечивает те же самые возможности как .dump команда в оболочке sqlite3. Пример ниже демонстрирует весь цикл работы:

```
import sqlite3, sys
# Подготавливаем данные БД
alphabet=[chr(i) for i in xrange(97,123)]
# Создаем первую БД и заполняем ее данными
con=sqlite3.connect(':memory:')
cur=con.cursor()
cur.execute('create table test(s)')
cur.executemany('insert into test values (?)',alphabet)
con.commit()
# делаем файл дамп данных первой базы
with open('/tmp/dump.sql', 'w') as f:
for line in con.iterdump():
    f.write('%s\n' % line)
con.close()
# создаем вторую БД
con=sqlite3.connect(':memory:')
cur=con.cursor()
# восстанавливаем дамп первой базы во второй
dump=open('/tmp/dump.sql')
for i in dump:
    try:
        cur.execute(i)
    except:
        print sys.exc_info()
dump.close()
# проверяем наличия данных
cur.execute('select * from test')
print cur.fetchall()
```

Объект Cursor

Class sqlite3.**Cursor**

Экземпляр класса Cursor имеет следующие атрибуты и методы.

Cursor.**execute**(*sql[, parameters]*)

Выполнить SQL выражение, которое может иметь параметры (заполнители вместо SQL литералов). Модуль sqlite3 поддерживает 2 вида заполнителей: вопросительный знак (стиль qmark) и именованные заполнители.

Пример показывает как использовать параметры обоими стилями:

```
import sqlite3
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("CREATE TABLE people(name_last TEXT, age INTEGER)")
cur.executemany("INSERT INTO people VALUES(?,?)",[('Smart',32),('Tesla',87),
('Yeltsin',72)])

who = "Yeltsin"
age = 72
cur.execute("select name_last, age from people where name_last=? and age=?",
(who, age))
print cur.fetchone()

cur.execute("select name_last, age from people where name_last=:who and
age=:age",{"who": who, "age": age})
print cur.fetchone()
```

Результаты:

```
(u'Yeltsin', 72)
(u'Yeltsin', 72)
```

execute() выполняет одиночное SQL выражение. Если попытаться выполнить более чем одно выражение то будет возбуждено Предупреждение. Используйте executescript() если необходимо выполнить многократные SQL выражения за один вызов.

Cursor.**executemany**(*sql, seq_of_parameters*)

Выполнить команду SQL против всех последовательностей параметра или отображения, найденные в последовательности sql. sqlite3 модуль также позволяет использовать итератор вместо последовательности.

Пример с использованием итератора:

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')
    def __iter__(self):
        return self
    def next(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
cur.execute("select c from characters")
print cur.fetchall()
```

Пример с использованием генератора:

```
import sqlite3
def char_generator():
    import string
    for c in string.letters[:26]:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
cur.executemany("insert into characters(c) values (?)", char_generator())
cur.execute("select c from characters")
print cur.fetchall()
```

Результат в обоих случаях идентичный:

```
[(u'a',), (u'b',), (u'c',), (u'd',), (u'e',), (u'f',), (u'g',), (u'h',), (u'i',),
(u'j',), (u'k',), (u'l',), (u'm',), (u'n',), (u'o',), (u'p',), (u'q',), (u'r',),
(u's',), (u't',), (u'u',), (u'v',), (u'w',), (u'x',), (u'y',), (u'z',)]
```

Cursor.**executescript**(*sql_script*)

Это не стандартный удобный метод позволяющий выполнить многократные SQL выражения за один раз. Сначала выполняется COMMIT объявление, затем выполняется SQL скрипт заданный как параметр. *sql_script* может быть как строка так и Unicode. Пример:

```
import sqlite3
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
        );
    create table book(
        title,
        author,
        published
        );
        insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
        );
    """)
```

Cursor.**fetchone()**

Получить следующий ряд набора результатов запроса, возвращает одиночную последовательность или None если более не доступно данных.

Cursor.**fetchmany**(*[size=cursor.arraysize]*)

Получить следующий набор рядов результатов запроса, возвращает список. Пустой список возвращается когда более нет доступных рядов.

Количество получаемых за раз рядов указывается параметром *size*. Если он не задан, то количество получаемых рядов определяется параметров arraysize курсора. Метод постарается получить настолько много рядов сколько указано параметром. Если это не возможно, то будет возвращено меньшее число рядов.

Примечание. Существуют некоторые соображения производительности касательно параметра size. Для оптимальной производительности обычно лучше использовать атрибут *arraysize*. Если используется параметр *size*, то лучше для этого сохранить тоже самое значение от одного вызова fetchmany() до другого.

Cursor.**fetchall()**

Получить все ряды результата запроса в виде списка. Примечательно что атрибут курсора *arraysize* может иметь воздействие на производительность данной операции. Пустой лист возвращается в случае отсутствия доступных рядов.

Cursor.**rowcount**

Хотя класс Cursor модуля sqlite3 реализует этот атрибут, собственная поддержка его движком базы данных определения «строки на которые влияют/ выбранные строки» является изворотливой.

Для выражений DELETE SQLite сообщает rowcont как 0 если сделать DELETE FROM table без каких либо условий.

Для выражения executemany() число модификаций суммируется вплоть до rowcount. Как требуется Спецификацией API DB Python, атрибут rowcount “-1 в случае, если никакой executeXX () не был выполнен на курсоре, или rowcount последней работы не определим интерфейсом”.

Так же это включает в себя оператор SELECT, потому что не возможно определить число рядов в результате запроса пока все строки не были выбраны.

Cursor.**lastrowid**

Этот атрибут, доступный только для чтения, предоставляет id последнего модифицированного ряда. Он устанавливается только если был использован оператор INSERT в методе execute().Для других операций отличных от INSERT или когда вызван executemany() атрибут принимает значение None.

Cursor.**description**

Этот атрибут доступный только для чтения, предоставляет колонку имен последнего запроса. Для соблюдения совместимости с Python DB API он

возвращает кортеж из семи значений, 6 последних значений которого равны None.

Также это установлено для оператора SELECT без любых строк соответствия.

Объект Row

class sqlite3.**Row**

Экземпляр Row служит высоко-оптимизированным row_factory для объектов Connection. Он пытается подражать кортежу в большинстве своих функций. Поддерживает отображающийся доступ по имени столбца, индексу, итерации, представлению, равному тестированию и len().

Если два объекта Row имеют точно такие же столбцы и их элементы равны, то они сравниваются как равные.

Изменения в версии 2.6: Добавлены итерации и равенство (подобие хэша)

keys()

Этот метод возвращает кортеж с именами столбцов. Непосртедственно после запроса это первый член каждого кортежа в Cursor.description.

Пример:

```
# -*- coding: utf-8 -*-
import sqlite3
# создаем базу и заполянем ее одним значением
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks(date text, trans text, symbol text, qty real,
price real)''')
c.execute("""insert into stocks values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()

# Демонстрируем возможности Row
conn.row_factory = sqlite3.Row
c = conn.cursor()
c.execute('select * from stocks')
r = c.fetchone()
print 'type(r) = ',type(r)
print 'r = ',r
print 'len(r) = ',len(r)
print 'r[2] = ',r[2]
print 'r.keys() = ',r.keys()
print 'r[qry] = ',r['qty']
for member in r:
    print member
```

Результат:

```
type(r) = <type 'sqlite3.Row'>
r = (u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.140000000000001)
len(r) = 5
r[2] = RHAT
r.keys() = ['date', 'trans', 'symbol', 'qty', 'price']
r[qry] = 100.0
2006-01-05
BUY
RHAT
100.0
35.14
```

SQLite и Python типы данных.

Введение

SQLite нативно поддерживает следующие типы: NULL, INTEGER, REAL, TEXT, BLOB. Нижеперечисленные типы Python могут быть посланы SQLite безо всяких проблем:

Типы Python	Типы SQLite
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str(UTF-8 encoded)	TEXT
uncode	TEXT

buffer	BLOB

SQLite типы конвертируются в Python по умолчанию следующим образом:

Типы SQLite	Типы Python
NULL	None
INTEGER	int или long, в зависимости от размера
REAL	float
TEXT	зависит от text_factory, по умолчанию unicode
BLOB	buffer

Система типов модуля sqlite3 позволяет расширяться двумя способами: возможно сохранение дополнительных типов Python в базе данных SQLite через адаптацию объектов или через преобразователи.

Использование адаптеров для хранения дополнительных типов Python в базах данных SQLite

Как говорилось ранее,SQLite поддерживает только ограниченные типы данных нативно. Для использования других типов данных Python в SQLite базах данных необходимо адаптировать их к оному из поддерживаемых модулем sqlite3 типу: None, int, long, float, str,Unicode, buffer.

Модуль sqlite3 использует адаптацию объектов Python, описанную в PEP 246. Используется протокол PrepareProtocol.

Существует два способа настроить модуль sqlite3 для адаптации специальных типов Python.

Объект адаптирующий самого себя.

Это хороший подход, если вы пишете собственный класс. Для того что бы поместить экземпляр класса в колонку базы данных, в первую очередь нужно выбрать один из поддерживаемых типов данных который будет использован для переназначения. Затем нужно назначить классу метод __conform__(self,protocol) который будет возвращать конвертированное значение.

В примере ниже в качестве конвертируемого типа используется str, параметр *protocol* равен PrepareProtocol:

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

Результат:

```
4.000000;-3.200000
```

Регистрация функции адаптера.

Другой возможностью является создание функции которая конвертирует тип в строковое представление и регистрирует функцию с помощью register_adapter().

> Памятка: Тип/класс что бы адаптироваться должен быть классом нового стиля, т.е. должен иметь object в качестве одного из своих оснований.

Предыдущий пример примет следующий вид:

```
import sqlite3

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)
```

```
con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

Реузльтат:

```
4.000000;-3.200000
```

Модуль sqlite3 имеет по умолчанию 2 адаптера для Python типов datetime.date и datetime.datetime.
Пример в котором объект datetime.datetime хранится не в ISO представлении, а как штамп времени UNIX:

```
import sqlite3
import datetime, time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

Результат:

```
1299207380.0
```

Конвертация SQLite значений в специальные типы Python.

Функции конверторов принимают в качестве параметра строку и возвращают преобразованный Python тип.

> Памятка: Функции конверторов **всегда** вызываются со строкой, не важно какой тип данных был послан из SQLite.

Интерпретация получаемых данных как типов Python модулем sqlite3 осуществляется либо неявно через объявление типов либо явно через имя столбца (использование констант PARSE_DECTYPES и PARSE_COKNAMES соответственно).
Пример ниже раскрывает оба этих случая:

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

class Point(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

def convert_point(s):
    x, y = map(float, s.split(";"))
    return Point(x, y)

# Регистрируем функуию адаптер
sqlite3.register_adapter(Point, adapt_point)

# Регистрируем функцию конвертор
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

########################
# 1) Используем объявление типа
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")
cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
result=cur.fetchone()[0]
print "with declared types:", result,type(result)
cur.close()
con.close()

######################
# 2) Используем имена столбцов
```

```
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")
cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
result=cur.fetchone()[0]
print "with column names:",result,type(result)
cur.close()
con.close()
```

Результат:

```
with declared types: (4.000000;-3.200000) <class '__main__.Point'>
with column names: (4.000000;-3.200000) <class '__main__.Point'>
```

Адаптеры и конверторы по умолчанию.

Для типов `date` и `datetime` в модуле `datetime` существуют базовые адаптеры, которые будут посланы в SQLite как ISO даты / ISO временные метки.

Базовые конверторы зарегистрированы под именем “date” для типа `datetime.date` и под именем “timestamp” для типа `datetime.datetime`.

Таким образом, можно использовать даты и временные метки из Python без каких либо дополнительных махинаций в большинстве случаев. Формат адаптеров также совместим с экспериментальными функциями даты и времени SQLite.

Пример ниже демонстрирует это:

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:",
detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts
[timestamp]"')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])
```

Результат:

```
2011-03-05 => 2011-03-05 <type 'datetime.date'>
2011-03-05 12:31:20.712000 => 2011-03-05 12:31:20.712000 <type 'datetime.datetime'>
current_date 2011-03-05 <type 'datetime.date'>
current_timestamp 2011-03-05 01:31:20 <type 'datetime.datetime'>
```

Контроль транзакций.

По умолчанию модуль sqlite3 открывает транзакции неявно перед оператором языка модификации данных (DML) таких как INSERT,UPDATE,DELETE,REPLACE и фиксирует транзакции неявно перед выражениями, не относящимися к DML или запросам (т.е. что-либо другое кроме SELECT или вышеупомянутого).

Так, если Вы в пределах транзакции и даете команду как CREATE TABLE..., VACUUM, PRAGMA, sqlite3 модуль будет фиксировать изменения неявно прежде, чем выполнить ту команду. Есть две причины для этого. Прежде всего, некоторые из этих команд не работают в пределах транзакций. Другая причина состоит в том, что sqlite3 должен отследить состояние транзакции (если транзакция является активной или нет).

Вы можете контролировать какого рода выражения BEGIN выполняются (или не выполняются вовсе) неявно через параметр *isolation_level* при вызове `connect()` или через свойство соединений isolation_level.

Если вы хотите **автоматический режим** фиксации изменений, то необходимо установить isolation_level в значение `None`.

Иначе оставьте этот параметр по умолчанию, результатом которого будет простой оператор BEGIN или установите его в один из поддерживаемых уровней изоляций: DEFERRED, IMMEDIATE, EXCLUSIVE.

Эффективное использование sqlite3

Использование методов ярлыка

Используя нестандартные методы execute(), executemany() и executescript() объекта Connection ваш код будет более кратким, потому что не будет явно создан объект Cursor, вместо этого он будет создан неявно и эти методы ярлыка возвращают объект курсора. Таким образом, вы можете выполнить оператор SELECT и выполнить итерации по нему используя только один вызов к объекту Connection.

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")]

con = sqlite3.connect(":memory:")
con.execute("create table person(firstname, lastname)")
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)
for row in con.execute("select firstname, lastname from person"):
    print row

print "I just deleted", con.execute("delete from person where 1=1").rowcount, "rows"
```

Результат:

```
(u'Hugo', u'Boss')
(u'Calvin', u'Klein')
I just deleted 2 rows
```

Доступ к столбцам по имени вместо индекса

Одна из полезных возможностей модуля sqlite3 является класс sqlite3.Row разработанный что бы быть использованным как фабрика рядов.

К рядам, обернутыми в этот класс, можно получить доступ как по индексам (подобно кортежам) так и, без влияния регистра, по имени.

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

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("CREATE TABLE people(name_last,age)")
cur.execute("INSERT INTO people values (?,?)",[(u'Степанов'),25])
cur.execute("select name_last, age from people")
for row in cur:
    assert row[0] == row["name_last"]
    assert row["name_last"] == row["nAmE_lAsT"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
print row[0],row[1]
```

Результатом будет отсутствие ошибок и результирующий вывод:

```
Степанов 25
```

Использование соединения как менеджера контекста.

Новое в версии 2.6

Объекты соединения могут использоваться в качестве менеджеров по контексту, которые автоматически фиксируют или откатывают транзакции. В случае исключения откатывается транзакция; иначе, транзакция фиксируется:

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

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Будет выполненно успешно, con.commit() будет вызван автоматически
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() будет вызван после блока with окончившегося исключением
# которое должно быть перехвачено и корректно обработано
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print "couldn't add Joe twice"
```

Результат:

```
couldn't add Joe twice
```

Мультипоточность.

У более старых версий SQLite были проблемы с совместным использованием соединений между потоками. Именно поэтому модуль Python отвергает совместное использование соединений и курсоров между потоками. Если Вы все еще попытаетесь сделать так, то вы получите исключение во время выполнения.

Единственное исключение это вызов метода `interrupt()` который имеет смысл вызвать только из другого потока.

Автор: Евгений на 01:48

Ярлыки: Программирование, python, SQLite, sqlite3

16 комментариев:

Павел 21 мая 2011 г. в 13:03

отличная статья - огромное спасибо!!!

Ответить

Анонимный 6 марта 2012 г. в 20:53

Спасибо.

Добавлю в закладки, будет в качестве руководства.

Ответить

Анонимный 10 марта 2012 г. в 09:52

отличная статья, сборник рецептов +1 в закладки

Ответить

Евпаториец 1 апреля 2012 г. в 17:04

Большое спасибо за проделанную работу!!!

Ответить

Анонимный 15 марта 2013 г. в 12:24

Очень хорошая работа. Все разложено по-полочкам. Есть качественные примеры. Спасибо.

Ответить

Анонимный 22 апреля 2013 г. в 19:07

Огромное спасибо!

Ответить

Анонимный 23 мая 2013 г. в 22:07

Лучшее руководство по этой теме найденное мною в инете... СПАСИБО !!!

Ответить

Анонимный 18 октября 2013 г. в 14:17

Предлагаю посмотреть бесплатный инструмент - Valentina Studio. Супер вещь!!! Очень рекомендую - best manager for SQLite!!! http://www.valentina-db.com/en/valentina-studio-overview

Ответить

Анонимный 4 октября 2014 г. в 19:21

Присоединяюсь к господам выше. Большое спасибо!

Ответить

Анонимный 24 июля 2015 г. в 14:08

Спасибо, всё подробно

Ответить

Unknown 19 октября 2015 г. в 14:26

Спасибо, в закладки!

Ответить

Unknown 22 марта 2016 г. в 11:26

Спасибо! Очень полезный перевод!

Ответить

Анонимный 28 января 2018 г. в 17:34

спасибо большое!

Ответить

gonzito 10 сентября 2018 г. в 21:16

Спасибо, просто супер побольше б таких статей!!!

Ответить

Питон 5 мая 2020 г. в 23:19

круто, еще на заметку про сохранение https://pythononline.ru/question/hranenie-funktsii-python-v-tablitse-sqlite

Ответить

Alex 15 июня 2020 г. в 12:47

Очень полезная статья.

Ответить

Введите текст комментария

Следующее Главная страница Предыдущее

Подписаться на: Комментарии к сообщению (Atom)