Notes on sqlite3

The module doc for sqlite3 at Python.org is a bit below par for clarity as far as even the generally poor docs for modules in the standard library go. It assumes you already know how to use SQL from previous experience, but to be fair its not their job to teach us SQL. Keep in mind that SQL is **not** an application, it is a **language** to be learned in and of itself. **sqlite3** (which is also referred to as **SQLite**) is an implementation of a database supporting an SQL interface in the Python environment – one which reasonalby adheres to the ANSI standard established for SQL.

The sqlite doc provides a link to a pretty good SQL Tutorial at w3schools.com:
https://www.w3schools.com/sql/
and another very similar tutorial/reference web site (slightly more complete) can be founds at:
http://www.1keydata.com/sql/sql.html

Looking at the big picture, let's consider the three primary components of SQL and add the Python environment support:
***DDL** (Data Definition Language) - the commands or methods you use to create a database.
***DML** (Data Manipulation Language) – the commands used to maintain the data in the database.
***DCL** (Data Control Language) – the security components of SQL.
The sqlite3 doc contextually mentions DDL and DML one time each, and DCL nada.
***Overriding** these 3 SQL components are the sqlite3 Python components that implement and expose the SQL-like database interface.

The reader deeply experienced in SQL is encouraged to abandon this summary and go to:
https://docs.python.org/3/library/sqlite3.html#module-sqlite3
where he or she can delve into the non-standard aspects of sqlite3.

What follows is a summary of **basic** functions, methods, and attributes in sqlite3 that is, in some ways, abbreviated; but in other ways substantially extended. **This summary is completely different in organization, orientation, and structure from the Python Library doc. It is optimized for someone who is modestly conversant with SQL.**

Note the following shortcomings of this summary: there are many methods and attributes from sqlite3 not mentioned here and little or no attempt is made in this document to address threading, bytestring conversion, detail of the "sqlite3.complete statement", aggregate functions, interrupts, access authorization, or adapters. **We have tried to maintain the language of the module documentation where it makes sense, change it where it doesn't, and in both cases place it in a context clearer for the non-SQL expert.**

sqlite3 Data Types and Their Relationship to Python Data Types

Let's get data types out of way for clarification as we go through the rest of the process. SQL defines around 20 data types but only five are available in Python and sqlite3.

Python type	SQLite type
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

sqlite3 Program Implementation Commands

The Process of Using sqlite3

A simplified, high level abstraction of creating and using a sqlite3 database in Python :

1. Import the module
2. Create a **connection object** to a database on disk or in memory
3. Make desired modifications to the database environment at the module level
4. Use the connection object to create a **cursor object** (which will expose methods and attributes necessary to use the database) – a curson object is essentially an active instance of the database.
5. Implement DDL commands to create and/or alter the database
6. Retrieve sqlite3.row information, if desired
7. Implement SQL DML commands to store and retrieve data
8. Use a module level command to destroy the database, if desired

Import the Module

import sqlite3 ***[as shortcutname]*** - just like any other module

Module Level Functions and Constants

sqlite3.connect – see below, Connecting to a Database
sqlite3.sqlite_version - The version number of the run-time SQLite library, as a string.
sqlite3.PARSE_DECLTYPES - This constant is meant to be used with the detect_types parameter of the **connect()** function. Setting it makes the sqlite3 module parse the declared type for each column it returns. It will parse out the first word of the declared type, i. e. for "integer primary key", it will parse out "integer", or for "number(10)" it will parse out "number". Then for that column, it will look into the converters dictionary and use the converter function registered there for that type.
sqlite3.PARSE_COLNAMES - For use with the detect_types parameter of the connect() function. Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that 'mytype' is the type of the column. It will try to find an entry of 'mytype' in the converters dictionary and then use the converter function found there to return the value. The column name found in Cursor.description is only the first word of the column name, i. e. if you use something like 'as "x [datetime]"' in your SQL, then it will parse out everything until the first blank for the column name: the column name would simply be "x".
sqlite3.complete_statement(sql) - Returns True if the string sql contains one or more complete SQL statements terminated by semicolons. It does not verify that the SQL is syntactically correct, only that there are no unclosed string literals and the statement is terminated by a semicolon. **This statement is provided to allow construction of a shell for a user interface – see library doc for an example.**

Connect to a Database

sq3con = **sqlite3.connect** ('some-db-including-path-as-apprpriate'[,detect_types]) – *see module doc for all options*
Example (in Windows): **sq3con** = **sqlite3.connect("D:\\Temp\\MyData.db")**

*Alternatively, the special name **:memory:** can be used to CREATE a database in RAM; general form**:** **sq3con=sqlite3.connect(":memory:") .** Of course, this will make the data impermanent, but will provide lightening fast access.*

Connection Objects/Methods

Having created a connection object (in our example we named it "**sq3con**") the following methods are available to it:
.cursor(factory=Cursor) – see **Establish a Cursor Object** below
.commit() - **sq3con**.commit() - conpletes a transaction; save changes and make them visible
Note: Any command that changes the database (basically, any SQL command other than SELECT) will automatically start a transaction if one is not already in effect. Automatically started transactions are committed when the last query finishes.

.rollback() - **sq3con.r**ollback – reverses any changes to the database since the last **commit()**
.close() - **sq3con**.close() – closes the database connection – **does NOT call commit() before closing**.
*There are numerous non-standard shortcuts not listed here – why develop bad habits?
.create_function(name, num_params, func) - Creates a user-defined function that you can later use from within SQL statements under the function name **name**. num_params is the number of parameters the function accepts (if num_params is -1, the function may take any number of arguments), and func is a Python callable that is called as the SQL function, i.e., def your-function-name.
.create_aggregate(name, num_params, aggregate_class)
.row_factory
.total_changes - Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.
.iterdump() - Returns an itera tor to dump the database in an SQL text format. Useful when saving an in-memory database to disk for later restoration. This function provides the same capabilities as the .dump command in the sqlite3 shell.

*[Note: **.execute** and **.executemany** as connection objects: We hate to mention this, but to avoid confusion we need to warn you that sqlite3 includes non-standard connection objects as well as cursor objects **both** using the .execute and .executemany keywords.]*

Establish a Cursor Object

With connection established, the next step is to create a "cursor object". A cursor object is a class of variable that tells Python everything it must know to access your database. General form: **CurObj = sq3con.cursor()**
[Note: the actual description is ".cursor(factory=Cursor)" where factory is an instance of cursor. The "factory" method/concept is well beyond the scope of this article, not to mention the knowledge of the authors. Technically, a factory is a static method that returns some callable object passed by class instead of instance. (ZOOOooomm!??) In this case it is probably a static implementation which would centralize modification of cursor objects.]

Cursor Object Methods and Attributes

CurObj [remember: "CurObj" is our example variable name for **squilte3.connect .cursor()**] has the following objects:
Execution commands .execute and .executemany – see **Python sqlite3 SQL Command Execution Methods** below
below.
.fetchone() - Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.
.fetchmany(size=cursor.arraysize) - Fetches the next set of rows of a query result, returning a list. An empty list is returned when no more rows are available.
.fetchall() - Fetches all (remaining) rows of a query result, returning a list.
.close() - Close the cursor now (rather than whenever __del__ is called).
.rowcount - Although the Cursor class of the sqlite3 module implements this attribute, the database engine's own support for the determination of "rows affected"/"rows selected" is quirky.
.lastrowid - This read-only attribute provides the rowid of the last modified row. It is only set if you issued an INSERT or a REPLACE statement using the **execute()** method.

Python sqlite3 SQL Command Execution Methods

.execute("sql [,parameters]") – after a connection is established and a cursor object is created, this is the Python method called to perform commands. It can be thought of as a "wrapper" for SQL commands.
.executemany.("sql [,parameters]") - like .execute, but for multiple records at a time.

A couple of notes on SQL syntax. Statements are case insensitive including SQL keywords. Multiple statements are separated by semicolons (;). SQL ignores white space. Parameters are separated by commas but a comma after the last parameter causes an error.

DDL sqlite3 Define, Modify and Destroy Database Structure

Python Implementation of SQL DDL Commands in sqlite3

One quick preliminary note: white space is ignored by SQL so it may be used for visual clarity but it is not part of the required syntax.

CurObj**.execute ("sql [,parameters]") -** this is the method called to execute SQL commands; *general form*:
CurObj**.execute** ("CREATE TABLE table name (col name data type , …)")
The three most common DDL commands called by **execute** are:

- CREATE TABLE – see example above. For each column created the column name and the type of data must be specified after the command. If you plan to have multiple tables that will need to be related, your first column should contain a key field. Your key field is a piece of data that is absolutely unique for each record (think "row"). In addition, every table should have at least one column in common with at least one other table in the database. Hint: Prior Planning Prevents Piss Poor Performance. *(The library doc shows the command bracketed by 3 single quotes but in our testing one double quote works just as well.)*

- ALTER TABLE – used to add, change or delete columns in your table. Various versions of SQL databases allow or do not allow various types of changes. SQL does not let you specify or change **key fields** so plan those before initial creation. Also, as a general rule it is a bad idea to change a table that already holds data. The most common commands used with ALTER TABLE are ADD and DROP. Generl form:
  CurObj**.execute** ('''ALTER TABLE table name (ADD col name data type , …)''')

- DROP TABLE - Deleting a table is as simple as:
  CurObj**.execute("DROP TABLE *tablename*")**

DML sqlite3 Create, Load, Retrieve, Alter and Destroy Data

ROW OBJECTS:

sqlite3.Row - A Row instance serves as a highly optimized row_factory for Connection objects. It mimics a tuple in most of its features.

.keys() - This method returns a list of column names. Immediately after a query, it is the first member of each tuple in *Cursor.description.*

[Note: This seems to us a strangely anomalous but useful object that doesn't really "fit", but here it is.]

SQL BASIC COMMANDS IMPLEMENTED BY AN ESTABLISHED CONNECTION AND CURSOR OBJECT USING EXECUTE OR EXECUTEMANY, I.E.:

CurObj.EXECUTE("SQL [,PARAMETERS]")

- ("INSERT INTO table name VALUES ('col values', .,.,.,)") list-name-of-tuples-containing-line-values which the "?" will sub into)
  Example: CurObj**.execute** ("INSERT INTO FlagColors VALUES ('United States', 'red', 'white', 'blue', 1960)")

The SQL statement may be parameterized (i. e. placeholders instead of SQL literals). The sqlite3 module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style). Extending the above example:

```
Import sqlite3 as sq
sq3con=sq.connect
sq3con.execute('''CREATE TABLE FlagColors (nation text, color1 text, color2 text, color3 text, yradopted integer)''')
TupleList=[('United States','red','white','blue',1960), ('United Kingdom', 'red', 'blue', 'white', 1801),
('Denmark','red','white',,1219)]
 CurObj.executemany ("INSERT INTO FlagColors VALUES (?,?,?,?,?)",TupleList)
```

- ("REPLACE string_to_searc, substring_to_find, string_replacement")
  General form: [SELECT] REPLACE('searchstring','findstring','replacestring') [columnLabel] FROM table_name

- ("UPDATE table_name SET column_name = new_value WHERE limiting_condition") – if you fail to include a WHERE clause you will update every field in your database!

- ("DELETE FROM column_name WHERE limiting_condition") removes entire rows, not columns. Without a WHERE clause, DELETE may (probably will) irretrieveable remove all data from your table.

- ('SELECT column name FROM table name WHERE the-val-in-the-col-named colname=?', variable name of value to match)

After defining the data desired with a SELECT command [i.e., CurObj**.execute** (SELECT nation, yradopted FROM FlagColors)] there are three options for retrieving it:

(1) fetchone(), (2)fetchmany() or (3) use the select statement as an iterator

Example of SELECT as an iterator:

```
for i in row CurObj.execute("SELECT * FROM table_name [WHERE limiting_condition]"):
        print(row)
```

12.6.5. EXCEPTIONS

exception sqlite3.Warning - A subclass of Exception.
exception sqlite3.Error - The base class of the other exceptions in this module. **exception sqlite3.DatabaseError** - Exception raised for errors that are related to the database.
exception sqlite3.IntegrityError - Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.
exception sqlite3.ProgrammingError - Exception raised for programming errors.

AN OVERVIEW OF SQL IMPLEMENTED COMMANDS AND FUNCTIONS IN SQLITE3

ALTER TABLE				
	RENAME TO	new table name		
	ADD	column datatype	[NULL / NOT NULL]	[constraints]
	~~DROP~~	~~column datatype~~	~~[NULL / NOT NULL]~~	~~[constraints]~~

CREATE				
		column		
	TABLE	datatype	[NULL / NOT NULL]	[constraints]
	INDEX	name of index	ON tablename (indexed column	[,…])
			WHERE	criteria
	UNIQUE INDEX	(same as above)		
				SELECT
	VIEW	view name	AS	statement

SELECT	[DISTINCT]	column names	
		[*] *wildcard for all*	
		[, || for concatenation]	
			RTRIM (column name)
		calculated field	
		AS	calc field label
	[AS]	aliasname	
	FROM	tablename	
		[UNION]	
	[WHERE]	criteria	
	[GROUP BY]		
	[HAVING]		
	[ORDER BY]	column name	[DESC]

		[,column name]…	[DESC]	
	[LIMIT]	integer value		
	[OFFSET]	integer value		

UPDATE	tablename	SET	column name =	expression
			WHERE	criteria

INSERT	INTO	tablename	[(columns)]	
			VALUES	(values)
			DEFAULT	(values)
REPLACE	INTO	"	"	"
INSERT OR REPLACE	INTO	"	"	"
INSERT OR ROLLBACK	INTO	"	"	"
INSERT OR ABORT	INTO	"	"	"
INSERT OR FAIL	INTO	"	"	"
INSERT OR IGNORE	INTO	"	"	"

DROP			
	[TABLE]	object name	
	[INDEX]	object name	
	[VIEW]	object name	
	[PROCUDURE]	object name	

BEGIN	[TRANSACTION]	-	-	-
COMMIT	[TRANSACTION]			
alias: END				
ROLLBACK TO	TRANSACTION			
	TO SAVEPOINT	save point name		
SAVEPOINT	savepoint name			
RELEASE SAVEPOINT	savepoint name			

			SELECT
WITH	tablename	AS	statement

DELETE FROM	tablename		
	WHERE	criteria	
	ORDER BY	column name	[DESC]
	LIMIT	integer value	

WHERE STATEMENT OPERATORS AND OPERATOR PRESCEDENCE

WHERE	=,<>,!=,<,>,<=,>=,!>,BETWEEN, IS NULL	
	AND	
	OR	
	IN()	(value,value,value…)
	NOT	
	LIKE	? wildcard, _ 1 cr, [set]

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
+   -
<<   >>   &   |
<   <=   >   >=
=   ==   !=   <>   IS   IS NOT   IN   LIKE   GLOB   MATCH   REGEXP
AND
OR
```

AGGREGATE FUNCTIONS SUPPORTED

AVG()
COUNT(*)
COUNT(DISTINCT X)
GROUP_CONCAT(X,Y)
MAX()
MIN()
SUM()
TOTAL()

SELECTED FUNCTIONS SUPPORTED

ABS()
COALESE(X,Y,….)
LEFT()
LTRIM(X)
RTRIM(X)

TRIM(X)
SQRT()
INSTR(X,Y)
LAST_INSERT_ROWID()
LENGTH(X)
QUOTE(X)
RANDOM()
REPLACE(X,Y,Z)
ROUND(X,Y)

Comments

--
/* */

SQLite3 Supported but Not Addressed in This Summary

	DROP
CREATE PROCEDURE	TRIGGER
CREATE VIEW	EXPLAIN
ANALYZE	INDEXED BY
ATTACH DATABASE	ON CONFLICT
DETATCH DATABASE	PRAGMA
CREATE TRIGGER	REINDEX
CREATE VIRTUAL TABLE	VACUUM