13.13.3 Cursor Objects

A `Cursor` instance has the following attributes and methods:

`execute`(*sql*, [*parameters*])

Executes a SQL statement. The SQL statement may be parametrized (i. e. placeholders instead of SQL literals). The `sqlite3` module supports two kinds of placeholders: question marks (qmark style) and named placeholders (named style).

This example shows how to use parameters with qmark style:

```
import sqlite3

con = sqlite3.connect("mydb")

cur = con.cursor()

who = "Yeltsin"
age = 72

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

Download as text (original file name: sqlite3/execute_1.py).

This example shows how to use the named style:

```
import sqlite3

con = sqlite3.connect("mydb")

cur = con.cursor()

who = "Yeltsin"
age = 72

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

Download as text (original file name: sqlite3/execute_2.py).

`execute()`
will only execute a single SQL statement. If you try to execute more than one statement with it, it will raise a Warning. Use `executescript()` if you want to execute multiple SQL statements with one call.

`executemany`(*sql*, *seq_of_parameters*)

Executes a SQL command against all parameter sequences or mappings found in the sequence *sql*. The `sqlite3` module also allows using an iterator yielding parameters instead of a sequence.

```
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()
```

Download as text (original file name: sqlite3/executemany_1.py).

Here's a shorter example using a generator:

```
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()
```

Download as text (original file name: sqlite3/executemany_2.py).

executescript(*sql_script*)

This is a nonstandard convenience method for executing multiple SQL statements at once. It issues a COMMIT statement first, then executes the SQL script it gets as a parameter.

sql_script can be a bytestring or a Unicode string.

Example:

```
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
    );
    """)
```

Download as text (original file name: sqlite3/executescript.py).

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.

For `SELECT` statements, `rowcount` is always None because we cannot determine the number of rows a query produced until all rows were fetched.