

12.6. `sqlite3` — DB-API 2.0 interface for SQLite databases

Source code: [Lib/sqlite3/](#)

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The `sqlite3` module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by [PEP 249](#).

To use the module, you must first create a [Connection](#) object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a [Connection](#), you can create a [Cursor](#) object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

The data you've saved is persistent and is available in subsequent sessions:

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack (see <https://xkcd.com/327/> for humorous example of what can go wrong).

Instead, use the DB-API's parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
              ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
              ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
              ]
c.executemany('INSERT INTO stocks VALUES (?, ?, ?, ?, ?)', purchases)
```

To retrieve data after executing a `SELECT` statement, you can either treat the cursor as an `iterator`, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

This example uses the iterator form:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

See also:

<https://github.com/ghaering/pysqlite>

The pysqlite web page – sqlite3 is developed externally under the name “pysqlite”.

<https://www.sqlite.org>

The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

<http://www.w3schools.com/sql/>

Tutorial, reference and examples for learning SQL syntax.

PEP 249 - Database API Specification 2.0

PEP written by Marc-André Lemburg.

12.6.1. Module functions and constants

`sqlite3.version`

The version number of this module, as a string. This is not the version of the SQLite library.

`sqlite3.version_info`

The version number of this module, as a tuple of integers. This is not the version of the SQLite library.

`sqlite3.sqlite_version`

The version number of the run-time SQLite library, as a string.

`sqlite3.sqlite_version_info`

The version number of the run-time SQLite library, as a tuple of integers.

`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 for that type there.

`sqlite3.PARSE_COLNAMES`

This constant is meant to be used 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 we will parse out everything until the first blank for the column name: the column name would simply be “x”.

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

Opens a connection to the SQLite database file *database*. You can use “:memory:” to open a database connection to a database that resides in RAM instead of on disk.

When a database is accessed by multiple connections, and one of the processes modifies the database, the SQLite database is locked until that transaction is committed. The *timeout* parameter specifies how long the connection should wait for the lock to go away until raising an exception. The default for the timeout parameter is 5.0 (five seconds).

For the *isolation_level* parameter, please see the [isolation_level](#) property of [Connection](#) objects.

SQLite natively supports only the types TEXT, INTEGER, REAL, BLOB and NULL. If you want to use other types you must add support for them yourself. The *detect_types* parameter and the using custom **converters** registered with the module-level [register_converter\(\)](#) function allow you to easily do that.

detect_types defaults to 0 (i. e. off, no type detection), you can set it to any combination of [PARSE_DECLTYPES](#) and [PARSE_COLNAMES](#) to turn type detection on.

By default, *check_same_thread* is [True](#) and only the creating thread may use the connection. If set [False](#), the returned connection may be shared across multiple threads. When using multiple threads with the same connection writing operations should be serialized by the user to avoid data corruption.

By default, the [sqlite3](#) module uses its [Connection](#) class for the connect call. You can, however, subclass the [Connection](#) class and make [connect\(\)](#) use your class instead by providing your class for the *factory* parameter.

Consult the section [SQLite and Python types](#) of this manual for details.

The [sqlite3](#) module internally uses a statement cache to avoid SQL parsing overhead. If you want to explicitly set the number of statements that are cached for the connection, you can set the *cached_statements* parameter. The currently implemented default is to cache 100 statements.

If *uri* is true, *database* is interpreted as a URI. This allows you to specify options. For example, to open a database in read-only mode you can use:

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

More information about this feature, including a list of recognized options, can be found in the [SQLite URI documentation](#).

Changed in version 3.4: Added the *uri* parameter.

`sqlite3.register_converter(typename, callable)`

Registers a callable to convert a bytestring from the database into a custom Python type. The callable will be invoked for all database values that are of the type *typename*. Confer the parameter *detect_types* of the [connect\(\)](#) function for how the type detection works. Note that the case of *typename* and the name of the type in your query must match!

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

Registers a callable to convert the custom Python type *type* into one of SQLite's supported types. The callable *callable* accepts as single parameter the Python value, and must return a value of the following types: int, float, str or bytes.

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 can be used to build a shell for SQLite, as in the following example:

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.enable_callback_tracebacks(*flag*)

By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to **True**. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use **False** to disable the feature again.

12.6.2. Connection Objects

`class sqlite3.Connection`

A SQLite database connection has the following attributes and methods:

`isolation_level`

Get or set the current isolation level. `None` for autocommit mode or one of “DEFERRED”, “IMMEDIATE” or “EXCLUSIVE”. See section [Controlling Transactions](#) for a more detailed explanation.

`in_transaction`

`True` if a transaction is active (there are uncommitted changes), `False` otherwise. Read-only attribute.

New in version 3.2.

`cursor(factory=Cursor)`

The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of `Cursor` or its subclasses.

`commit()`

This method commits the current transaction. If you don’t call this method, anything you did since the last call to `commit()` is not visible from other database connections. If you wonder why you don’t see the data you’ve written to the database, please check you didn’t forget to call this method.

`rollback()`

This method rolls back any changes to the database since the last call to `commit()`.

`close()`

This closes the database connection. Note that this does not automatically call `commit()`. If you just close your database connection without calling `commit()` first, your changes will be lost!

`execute(sql[, parameters])`

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor’s `execute()` method with the *parameters* given, and returns the cursor.

`executemany(sql[, parameters])`

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor’s `executemany()` method with the *parameters* given, and returns the cursor.

`executescript(sql_script)`

This is a nonstandard shortcut that creates a cursor object by calling the `cursor()` method, calls the cursor's `executescript()` method with the given `sql_script`, and returns the cursor.

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.

The function can return any of the types supported by SQLite: bytes, str, int, float and `None`.

Example:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

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

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

Creates a user-defined aggregate function.

The aggregate class must implement a `step` method, which accepts the number of parameters *num_params* (if *num_params* is -1, the function may take any number of arguments), and a `finalize` method which will return the final result of the aggregate.

The `finalize` method can return any of the types supported by SQLite: bytes, str, int, float and `None`.

Example:

```
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 (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])

```

create_collation(*name*, *callable*)

Creates a collation with the specified *name* and *callable*. The callable will be passed two string arguments. It should return -1 if the first is ordered lower than the second, 0 if they are ordered equal and 1 if the first is ordered higher than the second. Note that this controls sorting (ORDER BY in SQL) so your comparisons don't affect other SQL operations.

Note that the callable will get its parameters as Python bytestrings, which will normally be encoded in UTF-8.

The following example shows a custom collation that sorts “the wrong way”:

```

import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

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 (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()

```

To remove a collation, call `create_collation` with `None` as callable:

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

interrupt()

You can call this method from a different thread to abort any queries that might be executing on the connection. The query will then abort and the caller will get an exception.

set_authorizer(*authorizer_callback*)

This routine registers a callback. The callback is invoked for each attempt to access a column of a table in the database. The callback should return `SQLITE_OK` if access is allowed, `SQLITE_DENY` if the entire SQL statement should be aborted with an error and `SQLITE_IGNORE` if the column should be treated as a NULL value. These constants are available in the [sqlite3](#) module.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database (“main”, “temp”, etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the [sqlite3](#) module.

set_progress_handler(*handler*, *n*)

This routine registers a callback. The callback is invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *handler*.

set_trace_callback(*trace_callback*)

Registers *trace_callback* to be called for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as string) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the [Cursor.execute\(\)](#) methods. Other sources include the transaction management of the Python module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

New in version 3.3.

enable_load_extension(*enabled*)

This routine allows/disallows the SQLite engine to load SQLite extensions from shared libraries. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Loadable extensions are disabled by default. See [1].

New in version 3.2.

```
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 loading 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', 'brocc
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpki
    insert into recipe (name, ingredients) values ('broccoli pie', 'brocco
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin
    """)
for row in con.execute("select rowid, name, ingredients from recipe where
print(row)
```

load_extension(path)

This routine loads a SQLite extension from a shared library. You have to enable extension loading with `enable_load_extension()` before you can use this routine.

Loadable extensions are disabled by default. See [1].

New in version 3.2.

row_factory

You can change this attribute to a callable that accepts the cursor and the original row as a tuple and will return the real result row. This way, you can implement more advanced ways of returning results, such as returning an object that can also access columns by name.

Example:

```
import sqlite3

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

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

If returning a tuple doesn't suffice and you want name-based access to columns, you should consider setting `row_factory` to the highly-optimized `sqlite3.Row` type. `Row` provides both index-based and case-insensitive name-based access to columns with almost no memory overhead. It will probably be better than your own custom dictionary-based approach or even a `db_row` based solution.

text_factory

Using this attribute you can control what objects are returned for the `TEXT` data type. By default, this attribute is set to `str` and the `sqlite3` module will return Unicode objects for `TEXT`. If you want to return bytestrings instead, you can set it to `bytes`.

You can also set it to any other callable that accepts a single bytestring parameter and returns the resulting object.

See the following example code for illustration:

```
import sqlite3

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

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in
# database ...
```

```
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

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 iterator to dump the database in an SQL text format. Useful when saving an in-memory database for later restoration. This function provides the same capabilities as the `.dump` command in the **sqlite3** shell.

Example:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

12.6.3. Cursor Objects

`class sqlite3.Cursor`

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

`execute(sql[, parameters])`

Executes an SQL statement. 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).

Here's an example of both styles:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
```

```
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"w": who, "a": age})

print(cur.fetchone())
```

`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 an SQL command against all parameter sequences or mappings found in the sequence `seq_of_parameters`. 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())
```

Here's a shorter example using a [generator](#):

```
import sqlite3
import string

def char_generator():
```

```

    for c in string.ascii_lowercase:
        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())

```

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 an instance of `str`.

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
    );
""")

```

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.

The number of rows to fetch per call is specified by the *size* parameter. If it is not given, the cursor's `arraysize` determines the number of rows to be fetched. The method should try to fetch as many rows as indicated by the *size* parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the `arraysize` attribute. If the *size* parameter is used, then it is best for it to retain the same value from one `fetchmany()` call to the next.

fetchall()

Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's `arraysize` attribute can affect the performance of this operation. An empty list is returned when no rows are available.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward; a `ProgrammingError` exception will be raised if any operation is attempted with the cursor.

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 `executemany()` statements, the number of modifications are summed up into `rowcount`.

As required by the Python DB API Spec, the `rowcount` attribute "is -1 in case no `executeXX()` has been performed on the cursor or the rowcount of the last operation is not determinable by the interface". This includes `SELECT` statements because we cannot determine the number of rows a query produced until all rows were fetched.

With SQLite versions before 3.6.5, `rowcount` is set to 0 if you make a `DELETE FROM table` without any condition.

lastrowid

This read-only attribute provides the rowid of the last modified row. It is only set if you issued an `INSERT` statement using the `execute()` method. For operations other than `INSERT` or when `executemany()` is called, `lastrowid` is set to `None`.

arraysize

Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

description

This read-only attribute provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

connection

This read-only attribute provides the SQLite database `Connection` used by the `Cursor` object. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to `con`:

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

>>>

12.6.4. Row Objects

`class sqlite3.Row`

A `Row` instance serves as a highly optimized `row_factory` for `Connection` objects. It tries to mimic a tuple in most of its features.

It supports mapping access by column name and index, iteration, representation, equality testing and `len()`.

If two `Row` objects have exactly the same columns and their members are equal, they compare equal.

`keys()`

This method returns a list of column names. Immediately after a query, it is the first member of each tuple in `Cursor.description`.

Changed in version 3.5: Added support of slicing.

Let's assume we initialize a table as in the example given above:

```
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()
```

Now we plug [Row](#) in:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5. Exceptions

exception **sqlite3.Warning**

A subclass of [Exception](#).

exception **sqlite3.Error**

The base class of the other exceptions in this module. It is a subclass of [Exception](#).

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. It is a subclass of [DatabaseError](#).

exception **sqlite3.ProgrammingError**

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. It is a subclass of [DatabaseError](#).

12.6.6. SQLite and Python types

12.6.6.1. Introduction

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
<code>None</code>	<code>NULL</code>
<code>int</code>	<code>INTEGER</code>
<code>float</code>	<code>REAL</code>
<code>str</code>	<code>TEXT</code>
<code>bytes</code>	<code>BLOB</code>

This is how SQLite types are converted to Python types by default:

SQLite type	Python type
<code>NULL</code>	<code>None</code>
<code>INTEGER</code>	<code>int</code>
<code>REAL</code>	<code>float</code>
<code>TEXT</code>	depends on <code>text_factory</code> , <code>str</code> by default
<code>BLOB</code>	<code>bytes</code>

The type system of the `sqlite3` module is extensible in two ways: you can store additional Python types in a SQLite database via object adaptation, and you can let the `sqlite3` module convert SQLite types to different Python types via converters.

12.6.6.2. Using adapters to store additional Python types in SQLite databases

As described before, SQLite supports only a limited set of types natively. To use other Python types with SQLite, you must **adapt** them to one of the `sqlite3` module's supported types for SQLite: one of `NoneType`, `int`, `float`, `str`, `bytes`.

There are two ways to enable the `sqlite3` module to adapt a custom Python type to one of the supported ones.

12.6.6.2.1. Letting your object adapt itself

This is a good approach if you write the class yourself. Let's suppose you have a class like this:

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

Now you want to store the point in a single SQLite column. First you'll have to choose one of the supported types first to be used for representing the point. Let's just use str and separate the coordinates using a semicolon. Then you need to give your class a method `__conform__(self, protocol)` which must return the converted value. The parameter *protocol* will be `PrepareProtocol`.

```
import sqlite3

class Point:
    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])
```

12.6.6.2.2. Registering an adapter callable

The other possibility is to create a function that converts the type to the string representation and register the function with `register_adapter()`.

```
import sqlite3

class Point:
    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])
```

The `sqlite3` module has two default adapters for Python's built-in `datetime.date` and `datetime.datetime` types. Now let's suppose we want to store `datetime.datetime` objects not in ISO representation, but as a Unix timestamp.

```
import sqlite3
import datetime
import 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])
```

12.6.6.3. Converting SQLite values to custom Python types

Writing an adapter lets you send custom Python types to SQLite. But to make it really useful we need to make the Python to SQLite to Python roundtrip work.

Enter converters.

Let's go back to the `Point` class. We stored the `x` and `y` coordinates separated via semicolons as strings in SQLite.

First, we'll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

Note: Converter functions **always** get called with a `bytes` object, no matter under which data type you sent the value to SQLite.

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

Now you need to make the `sqlite3` module know that what you select from the database is actually a point. There are two ways of doing this:

- Implicitly via the declared type

- Explicitly via the column name

Both ways are described in section [Module functions and constants](#), in the entries for the constants `PARSE_DECLTYPES` and `PARSE_COLNAMES`.

The following example illustrates both approaches.

```
import sqlite3

class Point:
    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)).encode('ascii')

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

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#####
# 1) Using declared types
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")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

#####
# 1) Using column names
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')
print("with column names:", cur.fetchone()[0])
```

```
cur.close()
con.close()
```

12.6.6.4. Default adapters and converters

There are default adapters for the date and datetime types in the datetime module. They will be sent as ISO dates/ISO timestamps to SQLite.

The default converters are registered under the name “date” for `datetime.date` and under the name “timestamp” for `datetime.datetime`.

This way, you can use date/timestamps from Python without any additional fiddling in most cases. The format of the adapters is also compatible with the experimental SQLite date/time functions.

The following example demonstrates this.

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_TYPE_NAMES)
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]))
```

If a timestamp stored in SQLite has a fractional part longer than 6 numbers, its value will be truncated to microsecond precision by the timestamp converter.

12.6.7. Controlling Transactions

By default, the `sqlite3` module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. `INSERT/UPDATE/DELETE/REPLACE`), and commits transactions implicitly before a non-DML, non-query statement (i.e. anything other than `SELECT` or the aforementioned).

So if you are within a transaction and issue a command like `CREATE TABLE ...`, `VACUUM`, `PRAGMA`, the `sqlite3` module will commit implicitly before executing that command. There are two reasons for doing that. The first is that some of these commands don't work within transactions. The other reason is that `sqlite3` needs to keep track of the transaction state (if a transaction is active or not). The current transaction state is exposed through the `Connection.in_transaction` attribute of the connection object.

You can control which kind of `BEGIN` statements `sqlite3` implicitly executes (or none at all) via the `isolation_level` parameter to the `connect()` call, or via the `isolation_level` property of connections.

If you want **autocommit mode**, then set `isolation_level` to `None`.

Otherwise leave it at its default, which will result in a plain "BEGIN" statement, or set it to one of SQLite's supported isolation levels: "DEFERRED", "IMMEDIATE" or "EXCLUSIVE".

12.6.8. Using `sqlite3` efficiently

12.6.8.1. Using shortcut methods

Using the nonstandard `execute()`, `executemany()` and `executescript()` methods of the `Connection` object, your code can be written more concisely because you don't have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the `Connection` object.

```
import sqlite3

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

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

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

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

12.6.8.2. Accessing columns by name instead of by index

One useful feature of the `sqlite3` module is the built-in `sqlite3.Row` class designed to be used as a row factory.

Rows wrapped with this class can be accessed both by index (like tuples) and case-insensitively by name:

```
import sqlite3

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

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

12.6.8.3. Using the connection as a context manager

Connection objects can be used as context managers that automatically commit or rollback transactions. In the event of an exception, the transaction is rolled back; otherwise, the transaction is committed:

```
import sqlite3

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

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
```

12.6.9. Common issues

12.6.9.1. Multithreading

Older SQLite versions had issues with sharing connections between threads. That's why the Python module disallows sharing connections and cursors between threads. If you still try to do so, you will get an exception at runtime.

The only exception is calling the `interrupt()` method, which only makes sense to call from a different thread.

Footnotes

- [1] (1, 2) The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably Mac OS X) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass `--enable-loadable-sqlite-extensions` to configure.