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

New in version 2.5.

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.

sqlite3 was written by Gerhard Häring and 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 `/tmp/example` file:

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

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 cursor if we are done with it
c.close()
```

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.

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 = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
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)
```

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:

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...    print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
>>>
```

See also

http://www.pysqlite.org

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

http://www.sqlite.org

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

PEP 249 - Database API Specification 2.0

PEP written by Marc-André Lemburg.

12.13.1. Module functions and constants¶

`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*, *isolation_level*, *detect_types*, *factory*])¶

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 `Connection.isolation_level` property of `Connection` objects.

SQLite natively supports only the types TEXT, INTEGER, FLOAT, 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, 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.

`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, long, float, str (UTF-8 encoded), unicode or buffer.

`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 = raw_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, 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* as True. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.

12.13.2. Connection Objects¶

class `sqlite3.Connection`¶
A SQLite database connection has the following attributes and methods:

`Connection.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.

`Connection.cursor`([*cursorClass*])¶
The cursor method accepts a single optional parameter *cursorClass*. If supplied, this must be a custom cursor class that extends `sqlite3.Cursor`.

`Connection.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 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.

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

`Connection.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!

`Connection.execute`(*sql*[, *parameters*])¶
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `execute` method with the parameters given.

`Connection.executemany`(*sql*[, *parameters*])¶

This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `executemany` method with the parameters given.

`Connection.executescript`(*sql_script*)¶
This is a nonstandard shortcut that creates an intermediate cursor object by calling the cursor method, then calls the cursor's `executescript` method with the parameters given.

`Connection.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, and *func* is a Python callable that is called as the SQL function.

The function can return any of the types supported by SQLite: unicode, str, int, long, float, buffer and None.

Example:

```
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]
```

`Connection.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*, 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: unicode, str, int, long, float, buffer 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]
```

`Connection.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):
```

```
   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 (?)", [("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)
```

`Connection.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.

`Connection.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.

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

New in version 2.6.

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*.

`Connection.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[0]] = 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.

`Connection.text_factory`¶

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

For efficiency reasons, there's also a way to return Unicode objects only for non-ASCII data, and bytestrings otherwise. To activate it, set this attribute to `sqlite3.OptimizedUnicode`.

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()

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

AUSTRIA = u"\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 = str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == str
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that will ignore Unicode characters that cannot be
# decoded from UTF-8
con.text_factory = lambda x: unicode(x, "utf-8", "ignore")
cur.execute("select ?", ("this is latin1 and would normally create errors" +
                       u"\xe4\xf6\xfc".encode("latin1"),))
row = cur.fetchone()
assert type(row[0]) == unicode

# sqlite3 offers a built-in optimized text_factory that will return bytestring
# objects, if the data is in ASCII only, and otherwise return unicode objects
con.text_factory = sqlite3.OptimizedUnicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) == unicode

cur.execute("select ?", ("Germany",))
row = cur.fetchone()
assert type(row[0]) == str
```

`Connection.total_changes`¶

Returns the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

`Connection.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.

New in version 2.6.

Example:

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

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

12.13.3. Cursor Objects¶

class `sqlite3.Cursor`¶

A SQLite database cursor has the following attributes and methods:

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

Executes an 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()
```

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()
```

`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.

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

Executes an 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()
```

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()
```

`Cursor.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
   );
   """)
```

`Cursor.fetchone`()¶

Fetches the next row of a query result set, returning a single sequence, or `None` when no more data is available.

`Cursor.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.

`Cursor.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.

`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 DELETE statements, SQLite reports rowcount as 0 if you make a DELETE FROM table without any condition.

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.

Cursor.lastrowid¶

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

Cursor.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.

12.13.4. 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.

Changed in version 2.6: Added iteration and equality (hashability).

keys()¶

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

New in version 2.6.

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)
<type 'sqlite3.Row'>
>>> r
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.140000000000001)
>>> len(r)
5
>>> r[2]
u'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
```

12.13.5. SQLite and Python types¶

12.13.5.1. 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
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-encoded)	TEXT
unicode	TEXT
buffer	BLOB

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

SQLite type	Python type
NULL	None
INTEGER	int or long, depending on size
REAL	float
TEXT	depends on text_factory, unicode by default
BLOB	buffer

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.

12.13.5.2. 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, long, float, str, unicode, buffer.

The `sqlite3` module uses Python object adaptation, as described in **PEP 246** for this. The protocol to use is `PrepareProtocol`.

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

12.13.5.2.1. 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(object):
   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(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]
```

12.13.5.2.2. 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()`.

Note

The type/class to adapt must be a *new-style class*, i. e. it must have `object` as one of its bases.

```
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]
```

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, 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]
```

12.13.5.3. 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 string, no matter under which data type you sent the value to SQLite.

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

# 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()
```

12.13.5.4. 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_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])
```

12.13.6. 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).

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".

12.13.7. Using `sqlite3` efficiently¶

12.13.7.1. 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

# Using a dummy WHERE clause to not let SQLite take the shortcut table deletes.
print "I just deleted", con.execute("delete from person where 1=1").rowcount, "rows"
```

12.13.7.2. 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("mydb")
con.row_factory = sqlite3.Row
```

```
cur = con.cursor()
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"]
```

12.13.7.3. Using the connection as a context manager¶

New in version 2.6.

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 catched
try:
   with con:
       con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
   print "couldn't add Joe twice"
```


Previous topic

12.12. dumbdbm — Portable DBM implementation

Next topic

13. Data Compression and Archiving

This Page

- Show Source

Navigation

- index