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 an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.7.15 or newer.

This document includes four main sections:

- Tutorial teaches how to use the sqlite3 module.
- Reference describes the classes and functions this module defines.
- How-to guides details how to handle specific tasks.
- Explanation provides in-depth background on transaction control.

See also:

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

https://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.

Tutorial

In this tutorial, you will create a database of Monty Python movies using basic sqlite3 functionality. It assumes a fundamental understanding of database concepts, including cursors and transactions.

First, we need to create a new database and open a database connection to allow sqlite3 to work with it. Call sqlite3.connect() to create a connection to the database tutorial.db in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned Connection object con represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call con.cursor() to create the Cursor:

Now that we've got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration – thanks to the flexible typing feature of SQLite, specifying the data types is optional. Execute the `CREATE TABLE` statement by calling `cur.execute(...)`:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table for details). Execute that query by calling `cur.execute(...)`, assign the result to `res`, and call `res.fetchone()` to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a `tuple` containing the table's name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling `cur.execute(...)`:

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see Transaction control for details). Call `con.commit()` on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar `cur.execute(...)` to assign the result to `res`, and call `res.fetchall()` to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a `list` of two `tuples`, one per row, each containing that row's `score` value.

Now, insert three more rows by calling `cur.executemany(...)`:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
```

```
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks (see How to use placeholders to bind values in SQL queries for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item `tuple` of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling `con.close()` to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail', released in 197
>>> new_con.close()
```

You've now created an SQLite database using the `sqlite3` module, inserted data and retrieved values from it in multiple ways.

See also:

- How-to guides for further reading:
  - How to use placeholders to bind values in SQL queries
  - How to adapt custom Python types to SQLite values
  - How to convert SQLite values to custom Python types
  - How to use the connection context manager
  - How to create and use row factories
- Explanation for in-depth background on transaction control.

Reference

Module functions

autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL)

Open a connection to an SQLite database.

Parameters:

- **database** (path-like object) – The path to the database file to be opened. You can pass `":memory:"` to create an SQLite database existing only in memory, and open a connection to it.
- **timeout** (*float*) – How many seconds the connection should wait before raising an `OperationalError` when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.
- **detect_types** (*int*) – Control whether and how data types not natively supported by SQLite are looked up to be converted to Python types, using the converters registered with `register_converter()`. Set it to any combination (using `|`, bitwise or) of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to enable this. Column names takes precedence over declared types if both flags are set. Types cannot be detected for generated fields (for example `max(data)`), even when the *detect_types* parameter is set; `str` will be returned instead. By default (`0`), type detection is disabled.
- **isolation_level** (*str* | *None*) – Control legacy transaction handling behaviour. See `Connection.isolation_level` and Transaction control via the isolation_level attribute for more information. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"`; or `None` to disable opening transactions implicitly. Has no effect unless `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).
- **check_same_thread** (*bool*) – If `True` (default), `ProgrammingError` will be raised if the database connection is used by a thread other than the one that created it. If `False`, the connection may be accessed in multiple threads; write operations may need to be serialized by the user to avoid data corruption. See `threadsafety` for more information.
- **factory** (*Connection*) – A custom subclass of `Connection` to create the connection with, if not the default `Connection` class.
- **cached_statements** (*int*) – The number of statements that `sqlite3` should internally cache for this connection, to avoid parsing overhead. By default, 128 statements.
- **uri** (*bool*) – If set to `True`, *database* is interpreted as a URI with a file path and an optional query string. The scheme part *must* be `"file:"`, and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various How to work with SQLite URIs.
- **autocommit** (*bool*) – Control **PEP 249** transaction handling behaviour. See `Connection.autocommit` and Transaction control via the autocommit attribute for more information. *autocommit* currently defaults to `LEGACY_TRANSACTION_CONTROL`. The default will change to `False` in a future Python release.

Return type: *Connection*

Raises an auditing event `sqlite3.connect` with argument `database`.

Raises an auditing event `sqlite3.connect/handle` with argument `connection_handle`.

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

Changed in version 3.10: Added the `sqlite3.connect/handle` auditing event.

Changed in version 3.12: Added the *autocommit* parameter.

`sqlite3.`**`complete_statement`**`(`*`statement`*`)`

Return `True` if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

For example:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling `execute()`.

See `runsource()` in Lib/sqlite3/__main__.py for real-world use.

`sqlite3.`**`enable_callback_tracebacks`**`(`*`flag`*`, /)`

Enable or disable callback tracebacks. 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.

Note: Errors in user-defined function callbacks are logged as unraisable exceptions. Use an `unraisable hook handler` for introspection of the failed callback.

`sqlite3.`**`register_adapter`**`(`*`type`*`, `*`adapter`*`, /)`

Register an *adapter* callable to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a type that SQLite natively understands.

`sqlite3.`**`register_converter`**`(`*`typename`*`, `*`converter`*`, /)`

Register the *converter* callable to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a `bytes` object and should return an object of the desired Python type. Consult the parameter *detect_types* of `connect()` for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

`sqlite3.`**`LEGACY_TRANSACTION_CONTROL`**

Set `autocommit` to this constant to select old style (pre-Python 3.12) transaction control behaviour. See Transaction control via the isolation_level attribute for more information.

`sqlite3.`**`PARSE_COLNAMES`**

be wrapped in square brackets (`[]`).

```
SELECT p as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with PARSE_DECLTYPES using the | (bitwise or) operator.

sqlite3.**PARSE_DECLTYPES**

Pass this flag value to the *detect_types* parameter of connect() to look up a converter function using the declared types for each column. The types are declared when the database table is created. sqlite3 will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named "integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
 )
```

This flag may be combined with PARSE_COLNAMES using the | (bitwise or) operator.

sqlite3.**SQLITE_OK**
sqlite3.**SQLITE_DENY**
sqlite3.**SQLITE_IGNORE**

Flags that should be returned by the *authorizer_callback* callable passed to Connection.set_authorizer(), to indicate whether:

- Access is allowed (SQLITE_OK),
- The SQL statement should be aborted with an error (SQLITE_DENY)
- The column should be treated as a NULL value (SQLITE_IGNORE)

sqlite3.**apilevel**

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to "2.0".

sqlite3.**paramstyle**

String constant stating the type of parameter marker formatting expected by the sqlite3 module. Required by the DB-API. Hard-coded to "qmark".

> **Note:** The named DB-API parameter style is also supported.

sqlite3.**sqlite_version**

Version number of the runtime SQLite library as a string.

sqlite3.**sqlite_version_info**

Version number of the runtime SQLite library as a tuple of integers.

sqlite3.**threadsafety**

Integer constant required by the DB-API 2.0, stating the level of thread safety the sqlite3 module supports. This attribute is set based on the default threading mode the underlying SQLite library is compiled with. The SQLite threading modes are:

2. **Multi-thread**: In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**: In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows:

SQLite threading mode	thread-safety	SQLITE_THREADSAFE	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

Changed in version 3.11: Set *threadsafety* dynamically instead of hard-coding it to `1`.

`sqlite3.`**`version`**

Version number of this module as a `string`. This is not the version of the SQLite library.

Deprecated since version 3.12, will be removed in version 3.14: This constant used to reflect the version number of the `pysqlite` package, a third-party library which used to upstream changes to `sqlite3`. Today, it carries no meaning or practical value.

`sqlite3.`**`version_info`**

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

Deprecated since version 3.12, will be removed in version 3.14: This constant used to reflect the version number of the `pysqlite` package, a third-party library which used to upstream changes to `sqlite3`. Today, it carries no meaning or practical value.

`sqlite3.`**`SQLITE_DBCONFIG_DEFENSIVE`**
`sqlite3.`**`SQLITE_DBCONFIG_DQS_DDL`**
`sqlite3.`**`SQLITE_DBCONFIG_DQS_DML`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FKEY`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_QPSG`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_TRIGGER`**
`sqlite3.`**`SQLITE_DBCONFIG_ENABLE_VIEW`**
`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_ALTER_TABLE`**
`sqlite3.`**`SQLITE_DBCONFIG_LEGACY_FILE_FORMAT`**
`sqlite3.`**`SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE`**
`sqlite3.`**`SQLITE_DBCONFIG_RESET_DATABASE`**
`sqlite3.`**`SQLITE_DBCONFIG_TRIGGER_EQP`**
`sqlite3.`**`SQLITE_DBCONFIG_TRUSTED_SCHEMA`**
`sqlite3.`**`SQLITE_DBCONFIG_WRITABLE_SCHEMA`**

These constants are used for the `Connection.setconfig()` and `getconfig()` methods.

Added in version 3.12.

See also:

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html

SQLite docs: Database Connection Configuration Options

Connection objects

class sqlite3.**Connection**

Each open SQLite database is represented by a `Connection` object, which is created using `sqlite3.connect()`. Their main purpose is creating `Cursor` objects, and Transaction control.

See also:

- How to use connection shortcut methods
- How to use the connection context manager

An SQLite database connection has the following attributes and methods:

cursor(*factory=Cursor*)

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

blobopen(*table, column, row, /, *, readonly=False, name='main'*)

Open a `Blob` handle to an existing BLOB.

Parameters:
- **table** (*str*) – The name of the table where the blob is located.
- **column** (*str*) – The name of the column where the blob is located.
- **row** (*str*) – The name of the row where the blob is located.
- **readonly** (*bool*) – Set to `True` if the blob should be opened without write permissions. Defaults to `False`.
- **name** (*str*) – The name of the database where the blob is located. Defaults to `"main"`.

Raises: **OperationalError** – When trying to open a blob in a `WITHOUT ROWID` table.

Return type: Blob

Note: The blob size cannot be changed using the `Blob` class. Use the SQL function `zeroblob` to create a blob with a fixed size.

Added in version 3.11.

commit()

Commit any pending transaction to the database. If `autocommit` is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was committed by this method.

Roll back to the start of any pending transaction. If `autocommit` is `True`, or there is no open transaction, this method does nothing. If `autocommit` is `False`, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If `autocommit` is `False`, any pending transaction is implicitly rolled back. If `autocommit` is `True` or `LEGACY_TRANSACTION_CONTROL`, no implicit transaction control is executed. Make sure to `commit()` before closing to avoid losing pending changes.

execute(*sql, parameters=(), /*)

Create a new `Cursor` object and call `execute()` on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql, parameters, /*)

Create a new `Cursor` object and call `executemany()` on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script, /*)

Create a new `Cursor` object and call `executescript()` on it with the given *sql_script*. Return the new cursor object.

create_function(*name, narg, func, *, deterministic=False*)

Create or remove a user-defined SQL function.

Parameters:
- **name** (*str*) – The name of the SQL function.
- **narg** (*int*) – The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.
- **func** (callback | None) – A callable that is called when the SQL function is invoked. The callable must return a type natively supported by SQLite. Set to `None` to remove an existing SQL function.
- **deterministic** (*bool*) – If `True`, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

Raises: **NotSupportedError** – If *deterministic* is used with SQLite versions older than 3.8.3.

> *Changed in version 3.8:* Added the *deterministic* parameter.

Example:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

Create or remove a user-defined SQL aggregate function.

Parameters:
- **name** (*str*) – The name of the SQL aggregate function.
- **n_arg** (*int*) – The number of arguments the SQL aggregate function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (class | None) –
  A class must implement the following methods:
  - `step()`: Add a row to the aggregate.
  - `finalize()`: Return the final result of the aggregate as a type natively supported by SQLite.

  The number of arguments that the `step()` method must accept is controlled by *n_arg*.
  Set to `None` to remove an existing SQL aggregate function.

Example:

```
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.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])

con.close()
```

create_window_function(*name*, *num_params*, *aggregate_class*, /)

Create or remove a user-defined aggregate window function.

Parameters:
- **name** (*str*) – The name of the SQL aggregate window function to create or remove.
- **num_params** (*int*) – The number of arguments the SQL aggregate window function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (class | None) –
  A class that must implement the following methods:
  - `step()`: Add a row to the current window.
  - `value()`: Return the current value of the aggregate.
  - `inverse()`: Remove a row from the current window.
  - `finalize()`: Return the final result of the aggregate as a type natively supported by SQLite.

  The number of arguments that the `step()` and `value()` methods must accept is controlled by *num_params*.

not support aggregate window functions.

> *Added in version 3.11.*

Example:

```
# Example taken from https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, */*)

Create a collation named *name* using the collating function *callable*. *callable* is passed two `string` arguments, and it should return an `integer`:

- `1` if the first is ordered higher than the second
- `-1` if the first is ordered lower than the second

The following example shows a reverse sorting collation:

```
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.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()
```

Remove a collation function by setting *callable* to `None`.

> *Changed in version 3.11:* The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

`interrupt()`

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an `OperationalError`.

`set_authorizer(`*authorizer_callback*`)`

Register callable *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of `SQLITE_OK`, `SQLITE_DENY`, or `SQLITE_IGNORE` to signal how access to the column should be handled by the underlying SQLite library.

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.

Passing `None` as *authorizer_callback* will disable the authorizer.

> *Changed in version 3.11:* Added support for disabling the authorizer using `None`.

`set_progress_handler(`*progress_handler*`, `*n*`)`

Register callable *progress_handler* to be 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

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

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a `DatabaseError` exception.

`set_trace_callback(trace_callback)`

Register callable *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as `str`) 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 `sqlite3` module and the execution of triggers defined in the current database.

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

> **Note:** Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use `enable_callback_tracebacks()` to enable printing tracebacks from exceptions raised in the trace callback.

Added in version 3.3.

`enable_load_extension(enabled, /)`

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. 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.

> **Note:** The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the `--enable-loadable-sqlite-extensions` option to **configure**.

Raises an auditing event `sqlite3.enable_load_extension` with arguments `connection`, `enabled`.

Added in version 3.2.

Changed in version 3.10: Added the `sqlite3.enable_load_extension` auditing event.

```
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
```

```
# Example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew', 'broccoli peppers
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin onions ga
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie', 'broccoli cheese c
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin sugar flou
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE name MATCH
    print(row)
```

load_extension(*path*, */*, ***, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with enable_load_extension() before calling this method.

Parameters:
- **path** (*str*) – The path to the SQLite extension.
- **entrypoint** (*str* | *None*) – Entry point name. If None (the default), SQLite will come up with an entry point name of its own; see the SQLite docs Loading an Extension for details.

Raises an auditing event sqlite3.load_extension with arguments connection, path.

> *Added in version 3.2.*

> *Changed in version 3.10:* Added the sqlite3.load_extension auditing event.

> *Changed in version 3.12:* Added the *entrypoint* parameter.

iterdump()

Return an iterator to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the .dump command in the **sqlite3** shell.

Example:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

See also: How to handle non-UTF-8 text encodings

backup(*target*, ***, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

Parameters:
- **target** (*Connection*) – The database connection to save the backup to.

- **progress** (callback | None) – If set to a callable, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to `None`.
- **name** (*str*) – The name of the database to back up. Either `"main"` (the default) for the main database, `"temp"` for the temporary database, or the name of a custom database as attached using the `ATTACH DATABASE` SQL statement.
- **sleep** (*float*) – The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Added in version 3.7.

See also: How to handle non-UTF-8 text encodings

getlimit(*category*, */*)

Get a connection runtime limit.

Parameters: **category** (*int*) – The SQLite limit category to be queried.
Return type: int
Raises: **ProgrammingError** – If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for `Connection` `con` (the default is 1000000000):

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Added in version 3.11.

setlimit(*category*, *limit*, */*)

value of the limit is returned.

Parameters:
- **category** (*int*) – The SQLite limit category to be set.
- **limit** (*int*) – The value of the new limit. If negative, the current limit is unchanged.

Return type: int

Raises: **ProgrammingError** – If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for `Connection` `con` (the default limit is 10):

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Added in version 3.11.

getconfig(*op*, */*)

Query a boolean connection configuration option.

Parameters: **op** (*int*) – A SQLITE_DBCONFIG code.

Return type: bool

Added in version 3.12.

setconfig(*op*, *enable=True*, */*)

Set a boolean connection configuration option.

Parameters:
- **op** (*int*) – A SQLITE_DBCONFIG code.
- **enable** (*bool*) – `True` if the configuration option should be enabled (default); `False` if it should be disabled.

Added in version 3.12.

serialize(*, *name='main'*)

Serialize a database into a `bytes` object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

Parameters: **name** (*str*) – The database name to be serialized. Defaults to `"main"`.

Return type: bytes

Note: This method is only available if the underlying SQLite library has the serialize API.

Added in version 3.11.

deserialize(*data*, */*, *, *name='main'*)

Deserialize a `serialized` database into a `Connection`. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serial-

Parameters:
- **data** (*bytes*) – A serialized database.
- **name** (*str*) – The database name to deserialize into. Defaults to `"main"`.

Raises:
- **OperationalError** – If the database connection is currently involved in a read transaction or a backup operation.
- **DatabaseError** – If *data* does not contain a valid SQLite database.
- **OverflowError** – If `len(data)` is larger than `2**63 - 1`.

Note: This method is only available if the underlying SQLite library has the deserialize API.

Added in version 3.11.

`autocommit`

This attribute controls **PEP 249**-compliant transaction behaviour. `autocommit` has three allowed values:

- `False`: Select **PEP 249**-compliant transaction behaviour, implying that `sqlite3` ensures a transaction is always open. Use `commit()` and `rollback()` to close transactions.

  This is the recommended value of `autocommit`.

- `True`: Use SQLite's autocommit mode. `commit()` and `rollback()` have no effect in this mode.

- `LEGACY_TRANSACTION_CONTROL`: Pre-Python 3.12 (non-**PEP 249**-compliant) transaction control. See `isolation_level` for more details.

  This is currently the default value of `autocommit`.

Changing `autocommit` to `False` will open a new transaction, and changing it to `True` will commit any pending transaction.

See Transaction control via the autocommit attribute for more details.

Note: The `isolation_level` attribute has no effect unless `autocommit` is `LEGACY_TRANSACTION_CONTROL`.

Added in version 3.12.

`in_transaction`

This read-only attribute corresponds to the low-level SQLite autocommit mode.

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

Added in version 3.2.

`isolation_level`

Controls the legacy transaction handling mode of `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, implicit transaction management is performed.

Note: Using `autocommit` to control transaction handling is recommended over using `isolation_level`. `isolation_level` has no effect unless `autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).

`row_factory`

The initial `row_factory` for `Cursor` objects created from this connection. Assigning to this attribute does not affect the `row_factory` of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a `tuple`.

See How to create and use row factories for more details.

`text_factory`

A callable that accepts a `bytes` parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to `str`.

See How to handle non-UTF-8 text encodings for more details.

`total_changes`

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

Cursor objects

A `Cursor` object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using `Connection.cursor()`, or by using any of the connection shortcut methods.

Cursor objects are iterators, meaning that if you `execute()` a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

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

execute(*sql, parameters=(), /*)

Execute a single SQL statement, optionally binding Python values using placeholders.

Parameters:
- **sql** (*str*) – A single SQL statement.
- **parameters** (`dict` | sequence) – Python values to bind to placeholders in *sql*. A `dict` if named placeholders are used. A sequence if unnamed placeholders are used. See How to use placeholders to bind values in SQL queries.

Raises: **ProgrammingError** – If *sql* contains more than one SQL statement.

If `autocommit` is `LEGACY_TRANSACTION_CONTROL`, `isolation_level` is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly

> *Deprecated since version 3.12, will be removed in version 3.14:* `DeprecationWarning` is emitted if named placeholders are used and *parameters* is a sequence instead of a `dict`. Starting with Python 3.14, `ProgrammingError` will be raised instead.

Use `executescript()` to execute multiple SQL statements.

`executemany(sql, parameters, /)`

For every item in *parameters*, repeatedly execute the parameterized DML SQL statement *sql*.

Uses the same implicit transaction handling as `execute()`.

Parameters:
- **sql** (*str*) – A single SQL DML statement.
- **parameters** (iterable) – An iterable of parameters to bind with the placeholders in *sql*. See How to use placeholders to bind values in SQL queries.

Raises: **ProgrammingError** – If *sql* contains more than one SQL statement, or is not a DML statement.

Example:

```
rows = [
    ("row1",),
    ("row2",),
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

Note: Any resulting rows are discarded, including DML statements with RETURNING clauses.

> *Deprecated since version 3.12, will be removed in version 3.14:* `DeprecationWarning` is emitted if named placeholders are used and the items in *parameters* are sequences instead of `dict`s. Starting with Python 3.14, `ProgrammingError` will be raised instead.

`executescript(sql_script, /)`

Execute the SQL statements in *sql_script*. If the `autocommit` is `LEGACY_TRANSACTION_CONTROL` and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a `string`.

Example:

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

If `row_factory` is None, return the next row query result set as a `tuple`. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

`fetchmany(size=cursor.arraysize)`

Return the next set of rows of a query result as a `list`. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, `arraysize` determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are 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()`

Return all (remaining) rows of a query result as a `list`. Return an empty list if no rows are available. Note that the `arraysize` attribute can affect the performance of this operation.

`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.

`setinputsizes(sizes, /)`

Required by the DB-API. Does nothing in `sqlite3`.

`setoutputsize(size, column=None, /)`

Required by the DB-API. Does nothing in `sqlite3`.

`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.

`connection`

Read-only attribute that provides the SQLite database `Connection` belonging to the cursor. 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
>>> con.close()
```

`description`

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

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the `execute()` method. For other statements, after `executemany()` or `executescript()`, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

> **Note:** Inserts into `WITHOUT ROWID` tables are not recorded.

> *Changed in version 3.6:* Added support for the `REPLACE` statement.

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements; is `-1` for other statements, including CTE queries. It is only updated by the `execute()` and `executemany()` methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for `rowcount` to be updated.

row_factory

Control how a row fetched from this `Cursor` is represented. If `None`, a row is represented as a `tuple`. Can be set to the included `sqlite3.Row`; or a callable that accepts two arguments, a `Cursor` object and the `tuple` of row values, and returns a custom object representing an SQLite row.

Defaults to what `Connection.row_factory` was set to when the `Cursor` was created. Assigning to this attribute does not affect `Connection.row_factory` of the parent connection.

See How to create and use row factories for more details.

Row objects

class sqlite3.**Row**

A `Row` instance serves as a highly optimized `row_factory` for `Connection` objects. It supports iteration, equality testing, `len()`, and mapping access by column name and index.

Two `Row` objects compare equal if they have identical column names and values.

See How to create and use row factories for more details.

keys()

Return a `list` of column names as `strings`. Immediately after a query, it is the first member of each tuple in `Cursor.description`.

> *Changed in version 3.5:* Added support of slicing.

Blob objects

class sqlite3.**Blob**

A `Blob` instance is a file-like object that can read and write data in an SQLite BLOB. Call `len(blob)` to get the size (number of bytes) of the blob. Use indices and slices for direct access to the blob data.

Use the `Blob` as a context manager to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An `Error` (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1, /*)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF will be returned. When *length* is not specified, or is negative, `read()` will read until the end of the blob.

write(*data, /*)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise `ValueError`.

tell()

Return the current access position of the blob.

seek(*offset, origin=os.SEEK_SET, /*)

Set the current access position of the blob to *offset*. The *origin* argument defaults to `os.SEEK_SET` (absolute blob positioning). Other values for *origin* are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the blob's end).

PrepareProtocol objects

class sqlite3.**PrepareProtocol**

Exceptions

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is not currently raised by the `sqlite3` module, but may be raised by applications using `sqlite3`, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of `Exception`.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single `except` statement. `Error` is a subclass of `Exception`.

If the exception originated from within the SQLite library, the following two attributes are added to the exception:

sqlite_errorcode

The numeric error code from the SQLite API

> *Added in version 3.11.*

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API

> *Added in version 3.11.*

exception sqlite3.**InterfaceError**

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the `sqlite3` module. `InterfaceError` is a subclass of `Error`.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of `Error`.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of `DatabaseError`.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database's operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of `DatabaseError`.

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.**InternalError**

exception sqlite3.**ProgrammingError**

Exception raised for `sqlite3` API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed `Connection`. `ProgrammingError` is a subclass of `DatabaseError`.

exception sqlite3.**NotSupportedError**

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to `True` in `create_function()`, if the underlying SQLite library does not support deterministic functions. `NotSupportedError` is a subclass of `DatabaseError`.

SQLite and Python types

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
float	REAL
str	TEXT
bytes	BLOB

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

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

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

Default adapters and converters (deprecated)

Note: The default adapters and converters are deprecated as of Python 3.12. Instead, use the Adapter and converter recipes and tailor them to your needs.

- An adapter for `datetime.date` objects to `strings` in ISO 8601 format.
- An adapter for `datetime.datetime` objects to strings in ISO 8601 format.
- A converter for declared "date" types to `datetime.date` objects.
- A converter for declared "timestamp" types to `datetime.datetime` objects. Fractional parts will be truncated to 6 digits (microsecond precision).

Note: The default "timestamp" converter ignores UTC offsets in the database and always returns a naive `datetime.datetime` object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with `register_converter()`.

Deprecated since version 3.12.

Command-line interface

The `sqlite3` module can be invoked as a script, using the interpreter's `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

`-h, --help`
Print CLI help.

`-v, --version`
Print underlying SQLite library version.

Added in version 3.12.

How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a `tuple` of values to the second argument of the cursor's `execute()` method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a sequence whose length must match the num-

an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

Note: PEP 249 numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the Python types SQLite natively understands.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type `PrepareProtocol`.

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

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

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

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using `register_adapter()`.

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

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

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

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *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 are **always** passed a `bytes` object, no matter the underlying SQLite data type.

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

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of `connect()`. There are three options:

- Implicit: set *detect_types* to `PARSE_DECLTYPES`
- Explicit: set *detect_types* to `PARSE_COLNAMES`
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

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

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
```

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

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.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()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.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()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
```

```
sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the `execute()`, `executemany()`, and `executescript()` methods of the `Connection` class, 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.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A `Connection` object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If `autocommit` is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if `autocommit` is `True`, the context manager does nothing.

> **Note:** The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using `contextlib.closing()`.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
```

```
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- Open a database in read-only mode:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
```

- Do not implicitly create a new database file if it does not already exist; will raise `OperationalError` if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a `tuple`. If a `tuple` does not suit your needs, you can use the `sqlite3.Row` class or a custom `row_factory`.

While `row_factory` exists as an attribute both on the `Cursor` and the `Connection`, it is recommended to set `Connection.row_factory`, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]         # Access by index.
'Earth'
>>> row["name"]    # Access by name.
'Earth'
>>> row["RADIUS"]  # Column names are case-insensitive.
6378
>>> con.close()
```

Note: The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom `row_factory` that returns each row as a `dict`, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple`:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a named tuple:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
```

With some adjustments, the above recipe can be adapted to use a `dataclass`, or any other custom class, instead of a `namedtuple`.

How to handle non-UTF-8 text encodings

By default, `sqlite3` uses `str` to adapt SQLite values with the `TEXT` data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom `text_factory` to handle such cases.

Because of SQLite's flexible typing, it is not uncommon to encounter table columns with the `TEXT` data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let's assume we have a database with ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a `Connection` instance `con` connected to this database, we can decode the Latin-2 encoded text using this `text_factory`:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in `TEXT` table columns, you can use the following technique, borrowed from the Unicode HOWTO:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

Note: The `sqlite3` module API does not support strings containing surrogates.

See also: Unicode HOWTO

Explanation

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. Transaction control via the autocommit attribute is recommended, while Transaction control via the isolation_level attribute retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the `Connection.autocommit` attribute, which should preferably be set using the *autocommit* parameter of `connect()`.

It is suggested to set *autocommit* to `False`, which implies **PEP 249**-compliant transaction control. This means:

- `sqlite3` ensures that a transaction is always open, so `connect()`, `Connection.commit()`, and `Connection.rollback()` will implicitly open a new transaction (immediately after closing the pending one, for the latter two). `sqlite3` uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using `commit()`.
- Transactions should be rolled back explicitly using `rollback()`.
- An implicit rollback is performed if the database is `close()`-ed with pending changes.

compliant `Connection.autocommit` attribute; use `Connection.in_transaction` to query the low-level SQLite autocommit mode.

Set *autocommit* to `LEGACY_TRANSACTION_CONTROL` to leave transaction control behaviour to the `Connection.isolation_level` attribute. See Transaction control via the isolation_level attribute for more information.

Transaction control via the `isolation_level` attribute

Note: The recommended way of controlling transactions is via the `autocommit` attribute. See Transaction control via the autocommit attribute.

If `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default), transaction behaviour is controlled using the `Connection.isolation_level` attribute. Otherwise, `isolation_level` has no effect.

If the connection attribute `isolation_level` is not `None`, new transactions are implicitly opened before `execute()` and `executemany()` executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the `commit()` and `rollback()` methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour — that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes – via the `isolation_level` attribute.

If `isolation_level` is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the `in_transaction` attribute.

The `executescript()` method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of `isolation_level`.

> *Changed in version 3.6:* `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

> *Changed in version 3.12:* The recommended way of controlling transactions is now via the `autocommit` attribute.