sqlite-utils documentation

Release 4.0-10-g092f091

Simon Willison

Jul 08, 2026

CONTENTS

CLI tool and Python library for manipulating SQLite databases

This library and command-line utility helps create SQLite databases from an existing collection of data.

Most of the functionality is available as either a Python API or through the `sqlite-utils` command-line tool.

sqlite-utils is not intended to be a full ORM: the focus is utility helpers to make creating the initial database and populating it with data as productive as possible.

It is designed as a useful complement to Datasette.

Cleaning data with sqlite-utils and Datasette provides a tutorial introduction (and accompanying ten minute video) about using this tool.

CHAPTER

ONE

CONTENTS

1.1 Installation

`sqlite-utils` is tested on Linux, macOS and Windows.

1.1.1 Using Homebrew

The *sqlite-utils command-line tool* can be installed on macOS using Homebrew:

```
brew install sqlite-utils
```

If you have it installed and want to upgrade to the most recent release, you can run:

```
brew upgrade sqlite-utils
```

Then run `sqlite-utils --version` to confirm the installed version.

1.1.2 Using pip

The sqlite-utils package on PyPI includes both the *sqlite_utils Python library* and the `sqlite-utils` command-line tool. You can install them using `pip` like so:

```
pip install sqlite-utils
```

1.1.3 Using pipx

pipx is a tool for installing Python command-line applications in their own isolated environments. You can use `pipx` to install the `sqlite-utils` command-line tool like this:

```
pipx install sqlite-utils
```

1.1.4 Alternatives to sqlite3

By default, `sqlite-utils` uses the `sqlite3` package bundled with the Python standard library.

Depending on your operating system, this may come with some limitations.

On some platforms the ability to load additional extensions (via `conn.load_extension(...)` or `--load-extension=/path/to/extension`) may be disabled.

You may also see the error `sqlite3.OperationalError: table sqlite_master may not be modified` when trying to alter an existing table.

You can work around these limitations by installing the pysqlite3 package, which provides a drop-in replacement for the standard library `sqlite3` module but with a recent version of SQLite and full support for loading extensions.

To install `pysqlite3` run the following:

```
sqlite-utils install pysqlite3
```

`pysqlite3` does not provide an implementation of the `.iterdump()` method. To use that method (see *Dumping the database to SQL*) or the `sqlite-utils dump` command you should also install the `sqlite-dump` package:

```
sqlite-utils install sqlite-dump
```

1.1.5 Setting up shell completion

You can configure shell tab completion for the `sqlite-utils` command using these commands.

For `bash`:

```
eval "$(_SQLITE_UTILS_COMPLETE=bash_source sqlite-utils)"
```

For `zsh`:

```
eval "$(_SQLITE_UTILS_COMPLETE=zsh_source sqlite-utils)"
```

Add this code to `~/.zshrc` or `~/.bashrc` to automatically run it when you start a new shell.

See the Click documentation for more details.

1.2 sqlite-utils command-line tool

The `sqlite-utils` command-line tool can be used to manipulate SQLite databases in a number of different ways.

Once *installed* the tool should be available as `sqlite-utils`. It can also be run using `python -m sqlite_utils`.

1.2.1 Running SQL queries

The `sqlite-utils query` command lets you run queries directly against a SQLite database file. This is the default subcommand, so the following two examples work the same way:

```
sqlite-utils query dogs.db "select * from dogs"
```

```
sqlite-utils dogs.db "select * from dogs"
```

> **Note**
>
> In Python: *db.query()* CLI reference: *sqlite-utils query*

Pass `-` as the SQL query to read the query from standard input. This is useful for longer queries that would otherwise require careful shell escaping, or for piping in SQL generated by another tool:

```
echo "select * from dogs" | sqlite-utils query dogs.db -
```

```
sqlite-utils query dogs.db - < query.sql
```

Returning JSON

The default format returned for queries is JSON:

```
sqlite-utils dogs.db "select * from dogs"
```

```
[{"id": 1, "age": 4, "name": "Cleo"},
 {"id": 2, "age": 2, "name": "Pancakes"}]
```

If the query returns more than one column with the same name, later occurrences are renamed with a numeric suffix - `select 1 as id, 2 as id` returns `[{"id": 1, "id_2": 2}]`. This only applies to JSON output: *CSV and TSV* and *table* output keep the duplicate column headers unchanged.

Newline-delimited JSON

Use `--nl` to get back newline-delimited JSON objects:

```
sqlite-utils dogs.db "select * from dogs" --nl
```

```
{"id": 1, "age": 4, "name": "Cleo"}
{"id": 2, "age": 2, "name": "Pancakes"}
```

JSON arrays

You can use `--arrays` to request arrays instead of objects:

```
sqlite-utils dogs.db "select * from dogs" --arrays
```

```
[[1, 4, "Cleo"],
 [2, 2, "Pancakes"]]
```

You can also combine `--arrays` and `--nl`:

```
sqlite-utils dogs.db "select * from dogs" --arrays --nl
```

```
[1, 4, "Cleo"]
[2, 2, "Pancakes"]
```

If you want to pretty-print the output further, you can pipe it through `python -mjson.tool`:

```
sqlite-utils dogs.db "select * from dogs" | python -mjson.tool
```

```
[
    {
        "id": 1,
        "age": 4,
        "name": "Cleo"
    },
    {
        "id": 2,
        "age": 2,
        "name": "Pancakes"
    }
]
```

Unicode characters in JSON

JSON output includes unicode characters directly, without escaping them:

```
sqlite-utils dogs.db "select '' as text"
```

```
[{"text": ""}]
```

Use `--ascii` to escape non-ASCII characters as `\uXXXX` sequences instead:

```
sqlite-utils dogs.db "select '' as text" --ascii
```

```
[{"text": "\u65e5\u672c\u8a9e"}]
```

The `--ascii` option can help on systems that cannot display or process UTF-8, such as Windows consoles using a legacy code page. On Windows, setting the `PYTHONUTF8=1` environment variable is an alternative fix for `UnicodeEncodeError` crashes when redirecting output to a file.

Binary data in JSON

Binary strings are not valid JSON, so BLOB columns containing binary data will be returned as a JSON object containing base64 encoded data, that looks like this:

```
sqlite-utils dogs.db "select name, content from images" | python -mjson.tool
```

```
[
    {
        "name": "transparent.gif",
        "content": {
```

(continues on next page)

(continued from previous page)

```
            "$base64": true,
            "encoded": "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7"
        }
    }
]
```

Nested JSON values

If one of your columns contains JSON, by default it will be returned as an escaped string:

```
sqlite-utils dogs.db "select * from dogs" | python -mjson.tool
```

```
[
    {
        "id": 1,
        "name": "Cleo",
        "friends": "[{\"name\": \"Pancakes\"}, {\"name\": \"Bailey\"}]"
    }
]
```

You can use the `--json-cols` option to automatically detect these JSON columns and output them as nested JSON data:

```
sqlite-utils dogs.db "select * from dogs" --json-cols | python -mjson.tool
```

```
[
    {
        "id": 1,
        "name": "Cleo",
        "friends": [
            {
                "name": "Pancakes"
            },
            {
                "name": "Bailey"
            }
        ]
    }
]
```

Returning CSV or TSV

You can use the `--csv` option to return results as CSV:

```
sqlite-utils dogs.db "select * from dogs" --csv
```

```
id,age,name
1,4,Cleo
2,2,Pancakes
```

This will default to including the column names as a header row. To exclude the headers, use `--no-headers`:

```
sqlite-utils dogs.db "select * from dogs" --csv --no-headers
```

```
1,4,Cleo
2,2,Pancakes
```

Use `--tsv` instead of `--csv` to get back tab-separated values:

```
sqlite-utils dogs.db "select * from dogs" --tsv
```

```
id  age     name
1   4       Cleo
2   2       Pancakes
```

Table-formatted output

You can use the `--table` option (or `-t` shortcut) to output query results as a table:

```
sqlite-utils dogs.db "select * from dogs" --table
```

```
  id    age  name
----  -----  --------
   1      4  Cleo
   2      2  Pancakes
```

You can use the `--fmt` option to specify different table formats, for example `rst` for reStructuredText:

```
sqlite-utils dogs.db "select * from dogs" --fmt rst
```

```
====  =====  ========
  id    age  name
====  =====  ========
   1      4  Cleo
   2      2  Pancakes
====  =====  ========
```

Available `--fmt` options are:

- `asciidoc`
- `colon_grid`
- `double_grid`
- `double_outline`
- `fancy_grid`
- `fancy_outline`
- `github`
- `grid`
- `heavy_grid`
- `heavy_outline`
- `html`

- jira
- latex
- latex_booktabs
- latex_longtable
- latex_raw
- mediawiki
- mixed_grid
- mixed_outline
- moinmoin
- orgtbl
- outline
- pipe
- plain
- presto
- pretty
- psql
- rounded_grid
- rounded_outline
- rst
- simple
- simple_grid
- simple_outline
- textile
- tsv
- unsafehtml
- youtrack

This list can also be found by running `sqlite-utils query --help`.

Returning raw data, such as binary content

If your table contains binary data in a BLOB you can use the `--raw` option to output specific columns directly to standard out.

For example, to retrieve a binary image from a BLOB column and store it in a file you can use the following:

```
sqlite-utils photos.db "select contents from photos where id=1" --raw > myphoto.jpg
```

To return the first column of each result as raw data, separated by newlines, use `--raw-lines`:

```
sqlite-utils photos.db "select caption from photos" --raw-lines > captions.txt
```

Using named parameters

You can pass named parameters to the query using `-p`:

```
sqlite-utils query dogs.db "select :num * :num2" -p num 5 -p num2 6
```

```
[{":num * :num2": 30}]
```

These will be correctly quoted and escaped in the SQL query, providing a safe way to combine other values with SQL.

UPDATE, INSERT and DELETE

If you execute an `UPDATE`, `INSERT` or `DELETE` query the command will return the number of affected rows:

```
sqlite-utils dogs.db "update dogs set age = 5 where name = 'Cleo'"
```

```
[{"rows_affected": 1}]
```

Defining custom SQL functions

You can use the `--functions` option to pass a block of Python code that defines additional functions which can then be called by your SQL query.

This example defines a function which extracts the domain from a URL:

```
sqlite-utils query sites.db "select url, domain(url) from urls" --functions '
from urllib.parse import urlparse

def domain(url):
    return urlparse(url).netloc
'
```

Every callable object defined in the block will be registered as a SQL function with the same name, with the exception of functions with names that begin with an underscore.

You can also pass the path to a Python file containing function definitions:

```
sqlite-utils query sites.db "select url, domain(url) from urls" --functions functions.py
```

The `--functions` option can be used multiple times to load functions from multiple sources:

```
sqlite-utils query sites.db "select url, domain(url), extract_path(url) from urls" \
  --functions domain_funcs.py \
  --functions 'def extract_path(url):
    from urllib.parse import urlparse
    return urlparse(url).path'
```

SQLite extensions

You can load SQLite extension modules using the `--load-extension` option, see *Loading SQLite extensions*.

```
sqlite-utils dogs.db "select spatialite_version()" --load-extension=spatialite
```

```
[{"spatialite_version()": "4.3.0a"}]
```

Attaching additional databases

SQLite supports cross-database SQL queries, which can join data from tables in more than one database file.

You can attach one or more additional databases using the `--attach` option, providing an alias to use for that database and the path to the SQLite file on disk.

This example attaches the `books.db` database under the alias `books` and then runs a query that combines data from that database with the default `dogs.db` database:

```
sqlite-utils dogs.db --attach books books.db \
  'select * from sqlite_master union all select * from books.sqlite_master'
```

> **Note**
>
> In Python: *db.attach()*

1.2.2 Querying data directly using an in-memory database

The `sqlite-utils memory` command works similar to `sqlite-utils query`, but allows you to execute queries against an in-memory database.

You can also pass this command CSV or JSON files which will be loaded into a temporary in-memory table, allowing you to execute SQL against that data without a separate step to first convert it to SQLite.

Without any extra arguments, this command executes SQL against the in-memory database directly:

```
sqlite-utils memory 'select sqlite_version()'
```

```
[{"sqlite_version()": "3.35.5"}]
```

It takes all of the same output formatting options as *sqlite-utils query*: `--csv` and `--csv` and `--table` and `--nl`:

```
sqlite-utils memory 'select sqlite_version()' --csv
```

```
sqlite_version()
3.35.5
```

```
sqlite-utils memory 'select sqlite_version()' --fmt grid
```

```
+--------------------+
| sqlite_version()   |
+====================+
| 3.35.5             |
+--------------------+
```

Running queries directly against CSV or JSON

If you have data in CSV or JSON format you can load it into an in-memory SQLite database and run queries against it directly in a single command using `sqlite-utils memory` like this:

```
sqlite-utils memory data.csv "select * from data"
```

You can pass multiple files to the command if you want to run joins between data from different files:

```
sqlite-utils memory one.csv two.json \
  "select * from one join two on one.id = two.other_id"
```

If your data is JSON it should be the same format supported by the *sqlite-utils insert command* - so either a single JSON object (treated as a single row) or a list of JSON objects.

CSV data can be comma- or tab- delimited.

The in-memory tables will be named after the files without their extensions. The tool also sets up aliases for those tables (using SQL views) as `t1`, `t2` and so on, or you can use the alias `t` to refer to the first table:

```
sqlite-utils memory example.csv "select * from t"
```

If two files have the same name they will be assigned a numeric suffix:

```
sqlite-utils memory foo/data.csv bar/data.csv "select * from data_2"
```

To read from standard input, use either `-` or `stdin` as the filename - then use `stdin` or `t` or `t1` as the table name:

```
cat example.csv | sqlite-utils memory - "select * from stdin"
```

Incoming CSV data will be assumed to use `utf-8`. If your data uses a different character encoding you can specify that with `--encoding`:

```
cat example.csv | sqlite-utils memory - "select * from stdin" --encoding=latin-1
```

If you are joining across multiple CSV files they must all use the same encoding.

Column types will be automatically detected in CSV or TSV data, as described in *Inserting CSV or TSV data*. You can pass the `--no-detect-types` option to disable this automatic type detection and treat all CSV and TSV columns as `TEXT`.

Explicitly specifying the format

By default, `sqlite-utils memory` will attempt to detect the incoming data format (JSON, TSV or CSV) automatically.

You can instead specify an explicit format by adding a `:csv`, `:tsv`, `:json` or `:nl` (for newline-delimited JSON) suffix to the filename. For example:

```
sqlite-utils memory one.dat:csv two.dat:nl \
  "select * from one union select * from two"
```

Here the contents of `one.dat` will be treated as CSV and the contents of `two.dat` will be treated as newline-delimited JSON.

To explicitly specify the format for data piped into the tool on standard input, use `stdin:format` - for example:

```
cat one.dat | sqlite-utils memory stdin:csv "select * from stdin"
```

Joining in-memory data against existing databases using --attach

The *attach option* can be used to attach database files to the in-memory connection, enabling joins between in-memory data loaded from a file and tables in existing SQLite database files. An example:

```
echo "id\n1\n3\n5" | sqlite-utils memory - --attach trees trees.db \
  "select * from trees.trees where rowid in (select id from stdin)"
```

Here the `--attach trees trees.db` option makes the `trees.db` database available with an alias of `trees`.

`select * from trees.trees where ...` can then query the `trees` table in that database.

The CSV data that was piped into the script is available in the `stdin` table, so `... where rowid in (select id from stdin)` can be used to return rows from the `trees` table that match IDs that were piped in as CSV content.

--schema, --analyze, --dump and --save

To see the in-memory database schema that would be used for a file or for multiple files, use `--schema`:

```
sqlite-utils memory dogs.csv --schema
```

```
CREATE TABLE "dogs" (
    "id" INTEGER,
    "age" INTEGER,
    "name" TEXT
);
CREATE VIEW "t1" AS select * from "dogs";
CREATE VIEW "t" AS select * from "dogs";
```

You can run the equivalent of the *analyze-tables* command using `--analyze`:

```
sqlite-utils memory dogs.csv --analyze
```

```
dogs.id: (1/3)

  Total rows: 2
  Null rows: 0
  Blank rows: 0

  Distinct values: 2

dogs.name: (2/3)

  Total rows: 2
  Null rows: 0
  Blank rows: 0

  Distinct values: 2

dogs.age: (3/3)

  Total rows: 2
  Null rows: 0
  Blank rows: 0

  Distinct values: 2
```

You can output SQL that will both create the tables and insert the full data used to populate the in-memory database using `--dump`:

```
sqlite-utils memory dogs.csv --dump
```

```
BEGIN TRANSACTION;
CREATE TABLE "dogs" (
    "id" INTEGER,
    "age" INTEGER,
    "name" TEXT
);
INSERT INTO "dogs" VALUES('1','4','Cleo');
INSERT INTO "dogs" VALUES('2','2','Pancakes');
CREATE VIEW "t1" AS select * from "dogs";
CREATE VIEW "t" AS select * from "dogs";
COMMIT;
```

Passing `--save other.db` will instead use that SQL to populate a new database file:

```
sqlite-utils memory dogs.csv --save dogs.db
```

These features are mainly intended as debugging tools - for much more finely grained control over how data is inserted into a SQLite database file see *Inserting JSON data* and *Inserting CSV or TSV data*.

1.2.3 Returning all rows in a table

You can return every row in a specified table using the `rows` command:

```
sqlite-utils rows dogs.db dogs
```

```
[{"id": 1, "age": 4, "name": "Cleo"},
 {"id": 2, "age": 2, "name": "Pancakes"}]
```

This command accepts the same output options as `query` - so you can pass `--nl`, `--csv`, `--tsv`, `--no-headers`, `--table` and `--fmt`.

You can use the `-c` option to specify a subset of columns to return:

```
sqlite-utils rows dogs.db dogs -c age -c name
```

```
[{"age": 4, "name": "Cleo"},
 {"age": 2, "name": "Pancakes"}]
```

You can filter rows using a where clause with the `--where` option:

```
sqlite-utils rows dogs.db dogs -c name --where 'name = "Cleo"'
```

```
[{"name": "Cleo"}]
```

Or pass named parameters using `--where` in combination with `-p`:

```
sqlite-utils rows dogs.db dogs -c name --where 'name = :name' -p name Cleo
```

```
[{"name": "Cleo"}]
```

You can define a sort order using `--order column` or `--order 'column desc'`.

Use `--limit N` to only return the first `N` rows. Use `--offset N` to return rows starting from the specified offset.

> **Note**
>
> In Python: *table.rows* CLI reference: *sqlite-utils rows*

1.2.4 Listing tables

You can list the names of tables in a database using the `tables` command:

```
sqlite-utils tables mydb.db
```

```
[{"table": "dogs"},
 {"table": "cats"},
 {"table": "chickens"}]
```

You can output this list in CSV using the `--csv` or `--tsv` options:

```
sqlite-utils tables mydb.db --csv --no-headers
```

```
dogs
cats
chickens
```

If you just want to see the FTS4 tables, you can use `--fts4` (or `--fts5` for FTS5 tables):

```
sqlite-utils tables docs.db --fts4
```

```
[{"table": "docs_fts"}]
```

Use `--counts` to include a count of the number of rows in each table:

```
sqlite-utils tables mydb.db --counts
```

```
[{"table": "dogs", "count": 12},
 {"table": "cats", "count": 332},
 {"table": "chickens", "count": 9}]
```

Use `--columns` to include a list of columns in each table:

```
sqlite-utils tables dogs.db --counts --columns
```

```
[{"table": "Gosh", "count": 0, "columns": ["c1", "c2", "c3"]},
 {"table": "Gosh2", "count": 0, "columns": ["c1", "c2", "c3"]},
 {"table": "dogs", "count": 2, "columns": ["id", "age", "name"]}]
```

Use `--schema` to include the schema of each table:

```
sqlite-utils tables dogs.db --schema --table
```

```
table    schema
-------  ------------------------------------------------
Gosh     CREATE TABLE Gosh (c1 text, c2 text, c3 text)
Gosh2    CREATE TABLE Gosh2 (c1 text, c2 text, c3 text)
```

(continues on next page)

(continued from previous page)

```
dogs     CREATE TABLE "dogs" (
           "id" INTEGER,
           "age" INTEGER,
           "name" TEXT)
```

The `--nl`, `--csv`, `--tsv`, `--table` and `--fmt` options are also available.

> **Note**
>
> In Python: *db.tables or db.table_names()* CLI reference: *sqlite-utils tables*

1.2.5 Listing views

The `views` command shows any views defined in the database:

```
sqlite-utils views sf-trees.db --table --counts --columns --schema
```

```
view         count  columns              schema
---------  -------  -------------------  ------------------------------------------------
↪---------------
demo_view   189144  ['qSpecies']         CREATE VIEW demo_view AS select qSpecies from␣
↪Street_Tree_List
hello            1  ['sqlite_version()']  CREATE VIEW hello as select sqlite_version()
```

It takes the same options as the `tables` command:

- `--columns`
- `--schema`
- `--counts`
- `--nl`
- `--csv`
- `--tsv`
- `--table`

> **Note**
>
> In Python: *db.views or db.view_names()* CLI reference: *sqlite-utils views*

1.2.6 Listing indexes

The `indexes` command lists any indexes configured for the database:

```
sqlite-utils indexes covid.db --table
```

```
table                          index_name                                          ␣
↪    seqno    cid  name               desc  coll      key
-------------------------------  -------------------------------------------------␣
```

(continues on next page)

(continued from previous page)

```
↪ -------  -----  -----------------  ------  ------  -----
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_combined_key     ␣
↪       0     12  combined_key            0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_country_or_region␣
↪       0      1  country_or_region       0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_province_or_state␣
↪       0      2  province_or_state       0  BINARY      1
johns_hopkins_csse_daily_reports  idx_johns_hopkins_csse_daily_reports_day             ␣
↪       0      0  day                     0  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_date                       ␣
↪       0      0  date                    1  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_fips                       ␣
↪       0      3  fips                    0  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_county                     ␣
↪       0      1  county                  0  BINARY      1
ny_times_us_counties              idx_ny_times_us_counties_state                      ␣
↪       0      2  state                   0  BINARY      1
```

It shows indexes across all tables. To see indexes for specific tables, list those after the database:

```
sqlite-utils indexes covid.db johns_hopkins_csse_daily_reports --table
```

The command defaults to only showing the columns that are explicitly part of the index. To also include auxiliary columns use the `--aux` option - these columns will be listed with a `key` of `0`.

The command takes the same format options as the `tables` and `views` commands.

Note

In Python: *table.indexes* CLI reference: *sqlite-utils indexes*

1.2.7 Listing triggers

The `triggers` command shows any triggers configured for the database:

```
sqlite-utils triggers global-power-plants.db --table
```

```
name             table      sql
---------------  ---------  ------------------------------------------------------------
↪----
plants_insert    plants     CREATE TRIGGER "plants_insert" AFTER INSERT ON "plants"
                            BEGIN
                                INSERT OR REPLACE INTO "_counts"
                                VALUES (
                                  'plants',
                                  COALESCE(
                                    (SELECT count FROM "_counts" WHERE "table" = 'plants
↪'),
                                  0
                                  ) + 1
                                );
                            END
```

It defaults to showing triggers for all tables. To see triggers for one or more specific tables pass their names as arguments:

```
sqlite-utils triggers global-power-plants.db plants
```

The command takes the same format options as the `tables` and `views` commands.

> **Note**
>
> In Python: *table.triggers or db.triggers* CLI reference: *sqlite-utils triggers*

1.2.8 Showing the schema

The `sqlite-utils schema` command shows the full SQL schema for the database:

```
sqlite-utils schema dogs.db
```

```
CREATE TABLE "dogs" (
    "id" INTEGER PRIMARY KEY,
    "name" TEXT
);
```

This will show the schema for every table and index in the database. To view the schema just for a specified subset of tables pass those as additional arguments:

```
sqlite-utils schema dogs.db dogs chickens
```

> **Note**
>
> In Python: *table.schema* or *db.schema* CLI reference: *sqlite-utils schema*

1.2.9 Analyzing tables

When working with a new database it can be useful to get an idea of the shape of the data. The `sqlite-utils analyze-tables` command inspects specified tables (or all tables) and calculates some useful details about each of the columns in those tables.

To inspect the `tags` table in the `github.db` database, run the following:

```
sqlite-utils analyze-tables github.db tags
```

```
tags.repo: (1/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 14

  Most common:
    88: 107914493
    75: 140912432
```

(continues on next page)

(continued from previous page)

```
    27: 206156866

  Least common:
    1: 209590345
    2: 206649770
    2: 303218369

tags.name: (2/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 175

  Most common:
    10: 0.2
    9: 0.1
    7: 0.3

  Least common:
    1: 0.1.1
    1: 0.11.1
    1: 0.1a2

tags.sha: (3/3)

  Total rows: 261
  Null rows: 0
  Blank rows: 0

  Distinct values: 261
```

For each column this tool displays the number of null rows, the number of blank rows (rows that contain an empty string), the number of distinct values and, for columns that are not entirely distinct, the most common and least common values.

If you do not specify any tables every table in the database will be analyzed:

```
sqlite-utils analyze-tables github.db
```

If you wish to analyze one or more specific columns, use the `-c` option:

```
sqlite-utils analyze-tables github.db tags -c sha
```

To show more than 10 common values, use `--common-limit 20`. To skip the most common or least common value analysis, use `--no-most` or `--no-least`:

```
sqlite-utils analyze-tables github.db tags --common-limit 20 --no-least
```

Saving the analyzed table details

`analyze-tables` can take quite a while to run for large database files. You can save the results of the analysis to a database table called `_analyze_tables_` using the `--save` option:

```
sqlite-utils analyze-tables github.db --save
```

The `_analyze_tables_` table has the following schema:

```
CREATE TABLE "_analyze_tables_" (
    "table" TEXT,
    "column" TEXT,
    "total_rows" INTEGER,
    "num_null" INTEGER,
    "num_blank" INTEGER,
    "num_distinct" INTEGER,
    "most_common" TEXT,
    "least_common" TEXT,
    PRIMARY KEY ("table", "column")
);
```

The `most_common` and `least_common` columns will contain nested JSON arrays of the most common and least common values that look like this:

```
[
    ["Del Libertador, Av", 5068],
    ["Alberdi Juan Bautista Av.", 4612],
    ["Directorio Av.", 4552],
    ["Rivadavia, Av", 4532],
    ["Yerbal", 4512],
    ["Cosquín", 4472],
    ["Estado Plurinacional de Bolivia", 4440],
    ["Gordillo Timoteo", 4424],
    ["Montiel", 4360],
    ["Condarco", 4288]
]
```

1.2.10 Creating an empty database

You can create a new empty database file using the `create-database` command:

```
sqlite-utils create-database empty.db
```

To enable *WAL mode* on the newly created database add the `--enable-wal` option:

```
sqlite-utils create-database empty.db --enable-wal
```

To enable SpatiaLite metadata on a newly created database, add the `--init-spatialite` flag:

```
sqlite-utils create-database empty.db --init-spatialite
```

That will look for SpatiaLite in a set of predictable locations. To load it from somewhere else, use the `--load-extension` option:

```
sqlite-utils create-database empty.db --init-spatialite --load-extension /path/to/
↪spatialite.so
```

1.2.11 Running migrations

The `migrate` command applies pending Python migrations to a database. For the full migration file format and Python API, see *Database migrations*.

```
sqlite-utils migrate creatures.db path/to/migrations.py
```

If you omit the migration path it will search the current directory and subdirectories for files called `migrations.py`:

```
sqlite-utils migrate creatures.db
```

Use `--list` to list applied and pending migrations without running them:

```
sqlite-utils migrate creatures.db --list
```

Use `--stop-before` to stop before a named migration. The option can be passed more than once, and can target a specific migration set using `migration_set:migration_name`:

```
sqlite-utils migrate creatures.db path/to/migrations.py \
  --stop-before creatures:add_weight \
  --stop-before sales:drop_index
```

1.2.12 Inserting JSON data

If you have data as JSON, you can use `sqlite-utils insert tablename` to insert it into a database. The table will be created with the correct (automatically detected) columns if it does not already exist.

You can pass in a single JSON object or a list of JSON objects, either as a filename or piped directly to standard-in (by using `-` as the filename).

Here's the simplest possible example:

```
echo '{"name": "Cleo", "age": 4}' | sqlite-utils insert dogs.db dogs -
```

To specify a column as the primary key, use `--pk=column_name`.

To create a compound primary key across more than one column, use `--pk` multiple times.

If you feed it a JSON list it will insert multiple records. For example, if `dogs.json` looks like this:

```
[
    {
        "id": 1,
        "name": "Cleo",
        "age": 4
    },
    {
        "id": 2,
        "name": "Pancakes",
        "age": 2
    },
    {
```

(continues on next page)

(continued from previous page)

```
        "id": 3,
        "name": "Toby",
        "age": 6
    }
]
```

You can import all three records into an automatically created `dogs` table and set the `id` column as the primary key like so:

```
sqlite-utils insert dogs.db dogs dogs.json --pk=id
```

Pass `--pk` multiple times to define a compound primary key.

You can skip inserting any records that have a primary key that already exists using `--ignore`:

```
sqlite-utils insert dogs.db dogs dogs.json --pk=id --ignore
```

You can delete all the existing rows in the table before inserting the new records using `--truncate`:

```
sqlite-utils insert dogs.db dogs dogs.json --truncate
```

You can add the `--analyze` option to run `ANALYZE` against the table after the rows have been inserted.

Inserting binary data

You can insert binary data into a BLOB column by first encoding it using base64 and then structuring it like this:

```
[
    {
        "name": "transparent.gif",
        "content": {
            "$base64": true,
            "encoded": "R0lGODlhAQABAIAAAAAAAP///yH5BAEAAAAALAAAAAABAAEAAAIBRAA7"
        }
    }
]
```

Inserting newline-delimited JSON

You can also import newline-delimited JSON (see JSON Lines) using the `--nl` option:

```
echo '{"id": 1, "name": "Cleo"}
{"id": 2, "name": "Suna"}' | sqlite-utils insert creatures.db creatures - --nl
```

Newline-delimited JSON consists of full JSON objects separated by newlines.

If you are processing data using `jq` you can use the `jq -c` option to output valid newline-delimited JSON.

Since Datasette can export newline-delimited JSON, you can combine the Datasette and `sqlite-utils` like so:

```
curl -L "https://latest.datasette.io/fixtures/facetable.json?_shape=array&_nl=on" \
    | sqlite-utils insert nl-demo.db facetable - --pk=id --nl
```

You can also pipe `sqlite-utils` together to create a new SQLite database file containing the results of a SQL query against another database:

```
sqlite-utils sf-trees.db \
    "select TreeID, qAddress, Latitude, Longitude from Street_Tree_List" --nl \
  | sqlite-utils insert saved.db trees - --nl
```

```
sqlite-utils saved.db "select * from trees limit 5" --csv
```

```
TreeID,qAddress,Latitude,Longitude
141565,501X Baker St,37.7759676911831,-122.441396661871
232565,940 Elizabeth St,37.7517102172731,-122.441498017841
119263,495X Lakeshore Dr,,
207368,920 Kirkham St,37.760210314285,-122.47073935813
188702,1501 Evans Ave,37.7422086702947,-122.387293152263
```

Flattening nested JSON objects

`sqlite-utils insert` and `sqlite-utils memory` both expect incoming JSON data to consist of an array of JSON objects, where the top-level keys of each object will become columns in the created database table.

If your data is nested you can use the `--flatten` option to create columns that are derived from the nested data.

Consider this example document, in a file called `log.json`:

```
{
    "httpRequest": {
        "latency": "0.112114537s",
        "requestMethod": "GET",
        "requestSize": "534",
        "status": 200
    },
    "insertId": "6111722f000b5b4c4d4071e2",
    "labels": {
        "service": "datasette-io"
    }
}
```

Inserting this into a table using `sqlite-utils insert logs.db logs log.json` will create a table with the following schema:

```
CREATE TABLE "logs" (
   "httpRequest" TEXT,
   "insertId" TEXT,
   "labels" TEXT
);
```

With the `--flatten` option columns will be created using `topkey_nextkey` column names - so running `sqlite-utils insert logs.db logs log.json --flatten` will create the following schema instead:

```
CREATE TABLE "logs" (
   "httpRequest_latency" TEXT,
   "httpRequest_requestMethod" TEXT,
   "httpRequest_requestSize" TEXT,
   "httpRequest_status" INTEGER,
   "insertId" TEXT,
```

(continues on next page)

(continued from previous page)

```
   "labels_service" TEXT
);
```

1.2.13 Inserting CSV or TSV data

If your data is in CSV format, you can insert it using the `--csv` option:

```
sqlite-utils insert dogs.db dogs dogs.csv --csv
```

For tab-delimited data, use `--tsv`:

```
sqlite-utils insert dogs.db dogs dogs.tsv --tsv
```

Data is expected to be encoded as Unicode UTF-8. If your data is an another character encoding you can specify it using the `--encoding` option:

```
sqlite-utils insert dogs.db dogs dogs.tsv --tsv --encoding=latin-1
```

To stop inserting after a specified number of records - useful for getting a faster preview of a large file - use the `--stop-after` option:

```
sqlite-utils insert dogs.db dogs dogs.csv --csv --stop-after=10
```

A progress bar is displayed when inserting data from a file. You can hide the progress bar using the `--silent` option.

By default, column types are automatically detected for CSV or TSV files - resulting in a mix of `TEXT`, `INTEGER` and `REAL` columns. To disable type detection and treat all columns as `TEXT`, use the `--no-detect-types` option.

Detected types are only applied when the table is created by the command. Inserting CSV or TSV data into a table that already exists leaves the existing column types unchanged - values are inserted using the table's existing schema.

For example, given a `creatures.csv` file containing this:

```
name,age,weight
Cleo,6,45.5
Dori,1,3.5
```

The following command:

```
sqlite-utils insert creatures.db creatures creatures.csv --csv
```

Will produce this schema with automatically detected types:

```
sqlite-utils schema creatures.db
```

```
CREATE TABLE "creatures" (
   "name" TEXT,
   "age" INTEGER,
   "weight" REAL
);
```

To disable type detection and treat all columns as TEXT, use `--no-detect-types`:

```
sqlite-utils insert creatures.db creatures creatures.csv --csv --no-detect-types
```

If a CSV or TSV file includes empty cells, like this one:

```
name,age,weight
Cleo,6,
Dori,,3.5
```

They will be imported into SQLite as empty string values, "".

To import them as NULL values instead, use the --empty-null option:

```
sqlite-utils insert creatures.db creatures creatures.csv --csv --empty-null
```

Alternative delimiters and quote characters

If your file uses a delimiter other than , or a quote character other than " you can attempt to detect delimiters or you can specify them explicitly.

The --sniff option can be used to attempt to detect the delimiters:

```
sqlite-utils insert dogs.db dogs dogs.csv --sniff
```

Alternatively, you can specify them using the --delimiter and --quotechar options.

Here's a CSV file that uses ; for delimiters and the | symbol for quote characters:

```
name;description
Cleo;|Very fine; a friendly dog|
Pancakes;A local corgi
```

You can import that using:

```
sqlite-utils insert dogs.db dogs dogs.csv --delimiter=";" --quotechar="|"
```

Passing --delimiter, --quotechar or --sniff implies --csv, so you can omit the --csv option.

CSV files without a header row

The first row of any CSV or TSV file is expected to contain the names of the columns in that file.

If your file does not include this row, you can use the --no-headers option to specify that the tool should not use that fist row as headers.

If you do this, the table will be created with column names called untitled_1 and untitled_2 and so on. You can then rename them using the sqlite-utils transform ... --rename command, see *Transforming tables*.

1.2.14 Inserting unstructured data with --lines and --text

If you have an unstructured file you can insert its contents into a table with a single line column containing each line from the file using --lines. This can be useful if you intend to further analyze those lines using SQL string functions or *sqlite-utils convert*:

```
sqlite-utils insert logs.db loglines logfile.log --lines
```

This will produce the following schema:

```
CREATE TABLE "loglines" (
   "line" TEXT
);
```

You can also insert the entire contents of the file into a single column called `text` using `--text`:

```
sqlite-utils insert content.db content file.txt --text
```

The schema here will be:

```
CREATE TABLE "content" (
   "text" TEXT
);
```

1.2.15 Applying conversions while inserting data

The `--convert` option can be used to apply a Python conversion function to imported data before it is inserted into the database. It works in a similar way to *sqlite-utils convert*.

Your Python function will be passed a dictionary called `row` for each item that is being imported. You can modify that dictionary and return it - or return a fresh dictionary - to change the data that will be inserted.

Given a JSON file called `dogs.json` containing this:

```
[
    {"id": 1, "name": "Cleo"},
    {"id": 2, "name": "Pancakes"}
]
```

The following command will insert that data and add an `is_good` column set to 1 for each dog:

```
sqlite-utils insert dogs.db dogs dogs.json --convert 'row["is_good"] = 1'
```

The `--convert` option also works with the `--csv`, `--tsv` and `--nl` insert options.

As with `sqlite-utils convert` you can use `--import` to import additional Python modules, see *Importing additional modules* for details.

You can also pass code that runs some initialization steps and defines a `convert(value)` function, see *Defining a convert() function*.

--convert with --lines

Things work slightly differently when combined with the `--lines` or `--text` options.

With `--lines`, instead of being passed a `row` dictionary your function will be passed a `line` string representing each line of the input. Given a file called `access.log` containing the following:

```
INFO:     127.0.0.1:60581 - GET / HTTP/1.1 200 OK
INFO:     127.0.0.1:60581 - GET /foo/-/static/app.css?cead5a HTTP/1.1 200 OK
```

You could convert it into structured data like so:

```
sqlite-utils insert logs.db loglines access.log --convert '
type, source, _, verb, path, _, status, _ = line.split()
return {
    "type": type,
    "source": source,
    "verb": verb,
    "path": path,
```

(continues on next page)

(continued from previous page)

```
    "status": status,
}' --lines
```

The resulting table would look like this:

type	source	verb	path	status
INFO:	127.0.0.1:60581	GET	/	200
INFO:	127.0.0.1:60581	GET	/foo/-/static/app.css?cead5a	200

--convert with --text

With `--text` the entire input to the command will be made available to the function as a variable called `text`.

The function can return a single dictionary which will be inserted as a single row, or it can return a list or iterator of dictionaries, each of which will be inserted.

Here's how to use `--convert` and `--text` to insert one record per word in the input:

```
echo 'A bunch of words' | sqlite-utils insert words.db words - \
    --text --convert '({"word": w} for w in text.split())'
```

The result looks like this:

```
sqlite-utils dump words.db
```

```
BEGIN TRANSACTION;
CREATE TABLE "words" (
   "word" TEXT
);
INSERT INTO "words" VALUES('A');
INSERT INTO "words" VALUES('bunch');
INSERT INTO "words" VALUES('of');
INSERT INTO "words" VALUES('words');
COMMIT;
```

1.2.16 Inserting rows generated by Python code

Instead of providing a `FILE` to import, you can use the `--code` option to pass a block of Python code that generates the rows to insert. This is the command-line equivalent of calling `db["creatures"].insert_all(rows())` from the *Python API*.

Your code should define either a `rows()` function that returns or yields dictionaries, or a `rows` iterable such as a list of dictionaries:

```
sqlite-utils insert data.db creatures --code '
def rows():
    yield {"id": 1, "name": "Cleo"}
    yield {"id": 2, "name": "Suna"}
' --pk id
```

`--code` can also be given a path to a Python `.py` file.

The `--code` option works with both `sqlite-utils insert` and `sqlite-utils upsert`, and composes with table options such as `--pk`, `--replace`, `--alter`, `--not-null` and `--default`. It cannot be combined with a `FILE` argument or with input format options such as `--csv` or `--convert`.

1.2.17 Insert-replacing data

The `--replace` option to `insert` causes any existing records with the same primary key to be replaced entirely by the new records.

To replace a dog with in ID of 2 with a new record, run the following:

```
echo '{"id": 2, "name": "Pancakes", "age": 3}' | \
    sqlite-utils insert dogs.db dogs - --pk=id --replace
```

1.2.18 Upserting data

Upserting is update-or-insert. If a row exists with the specified primary key the provided columns will be updated. If no row exists that row will be created.

Unlike `insert --replace`, an upsert will ignore any column values that exist but are not present in the upsert document.

For example:

```
echo '{"id": 2, "age": 4}' | \
    sqlite-utils upsert dogs.db dogs - --pk=id
```

This will update the dog with an ID of 2 to have an age of 4, creating a new record (with a null name) if one does not exist. If a row DOES exist the name will be left as-is.

If the table already exists and has a primary key, you can omit the `--pk` option and `sqlite-utils` will use that existing primary key.

The command will fail if you reference columns that do not exist on the table. To automatically create missing columns, use the `--alter` option.

> **Note**
>
> `upsert` in sqlite-utils 1.x worked like `insert ... --replace` does in 2.x. See issue #66 for details of this change.

1.2.19 Executing SQL in bulk

If you have a JSON, newline-delimited JSON, CSV or TSV file you can execute a bulk SQL query using each of the records in that file using the `sqlite-utils bulk` command.

The command takes the database file, the SQL to be executed and the file containing records to be used when evaluating the SQL query.

The SQL query should include `:named` parameters that match the keys in the records.

For example, given a `chickens.csv` CSV file containing the following:

```
id,name
1,Blue
2,Snowy
3,Azi
```

(continues on next page)

(continued from previous page)

```
4,Lila
5,Suna
6,Cardi
```

You could insert those rows into a pre-created `chickens` table like so:

```
sqlite-utils bulk chickens.db \
  'insert into chickens (id, name) values (:id, :name)' \
  chickens.csv --csv
```

This command takes the same options as the `sqlite-utils insert` command - so it defaults to expecting JSON but can accept other formats using `--csv` or `--tsv` or `--nl` or other options described above.

By default all of the SQL queries will be executed in a single transaction. To commit every 20 records, use `--batch-size 20`.

1.2.20 Inserting data from files

The `insert-files` command can be used to insert the content of files, along with their metadata, into a SQLite table.

Here's an example that inserts all of the GIF files in the current directory into a `gifs.db` database, placing the file contents in an `images` table:

```
sqlite-utils insert-files gifs.db images *.gif
```

You can also pass one or more directories, in which case every file in those directories will be added recursively:

```
sqlite-utils insert-files gifs.db images path/to/my-gifs
```

By default this command will create a table with the following schema:

```
CREATE TABLE "images" (
    "path" TEXT PRIMARY KEY,
    "content" BLOB,
    "size" INTEGER
);
```

Content will be treated as binary by default and stored in a `BLOB` column. You can use the `--text` option to store that content in a `TEXT` column instead.

You can customize the schema using one or more `-c` options. For a table schema that includes just the path, MD5 hash and last modification time of the file, you would use this:

```
sqlite-utils insert-files gifs.db images *.gif -c path -c md5 -c mtime --pk=path
```

This will result in the following schema:

```
CREATE TABLE "images" (
    "path" TEXT PRIMARY KEY,
    "md5" TEXT,
    "mtime" REAL
);
```

Note that there's no `content` column here at all - if you specify custom columns using `-c` you need to include `-c content` to create that column.

You can change the name of one of these columns using a `-c colname:coldef` parameter. To rename the `mtime` column to `last_modified` you would use this:

```
sqlite-utils insert-files gifs.db images *.gif \
    -c path -c md5 -c last_modified:mtime --pk=path
```

You can pass `--replace` or `--upsert` to indicate what should happen if you try to insert a file with an existing primary key. Pass `--alter` to cause any missing columns to be added to the table.

The full list of column definitions you can use is as follows:

name
: The name of the file, e.g. `cleo.jpg`

path
: The path to the file relative to the root folder, e.g. `pictures/cleo.jpg`

fullpath
: The fully resolved path to the image, e.g. `/home/simonw/pictures/cleo.jpg`

sha256
: The SHA256 hash of the file contents

md5
: The MD5 hash of the file contents

mode
: The permission bits of the file, as an integer - you may want to convert this to octal

content
: The binary file contents, which will be stored as a BLOB

content_text
: The text file contents, which will be stored as TEXT

mtime
: The modification time of the file, as floating point seconds since the Unix epoch

ctime
: The creation time of the file, as floating point seconds since the Unix epoch

mtime_int
: The modification time as an integer rather than a float

ctime_int
: The creation time as an integer rather than a float

mtime_iso
: The modification time as an ISO timestamp, e.g. `2020-07-27T04:24:06.654246`

ctime_iso
: The creation time is an ISO timestamp

size
: The integer size of the file in bytes

stem
: The filename without the extension - for `file.txt.gz` this would be `file.txt`

suffix
: The file extension - for `file.txt.gz` this would be `.gz`

You can insert data piped from standard input like this:

```
cat dog.jpg | sqlite-utils insert-files dogs.db pics - --name=dog.jpg
```

The `-` argument indicates data should be read from standard input. The string passed using the `--name` option will be used for the file name and path values.

When inserting data from standard input only the following column definitions are supported: `name`, `path`, `content`, `content_text`, `sha256`, `md5` and `size`.

1.2.21 Converting data in columns

The `convert` command can be used to transform the data in a specified column - for example to parse a date string into an ISO timestamp, or to split a string of tags into a JSON array.

The command accepts a database, table, one or more columns and a string of Python code to be executed against the values from those columns. The following example would replace the values in the `headline` column in the `articles` table with an upper-case version:

```
sqlite-utils convert content.db articles headline 'value.upper()'
```

The Python code is passed as a string. Within that Python code the `value` variable will be the value of the current column.

The code you provide will be compiled into a function that takes `value` as a single argument. If you break your function body into multiple lines the last line should be a `return` statement:

```
sqlite-utils convert content.db articles headline '
value = str(value)
return value.upper()'
```

Your code will be automatically wrapped in a function, but you can also define a function called `convert(value)` which will be called, if available:

```
sqlite-utils convert content.db articles headline '
def convert(value):
    return value.upper()'
```

Use a `CODE` value of `-` to read from standard input:

```
cat mycode.py | sqlite-utils convert content.db articles headline -
```

Where `mycode.py` contains a fragment of Python code that looks like this:

```
def convert(value):
    return value.upper()
```

The conversion will be applied to every row in the specified table. You can limit that to just rows that match a `WHERE` clause using `--where`:

```
sqlite-utils convert content.db articles headline 'value.upper()' \
    --where "headline like '%cat%'"
```

You can include named parameters in your where clause and populate them using one or more `--param` options:

```
sqlite-utils convert content.db articles headline 'value.upper()' \
    --where "headline like :query" \
    --param query '%cat%'
```

The `--dry-run` option will output a preview of the conversion against the first ten rows, without modifying the database.

Importing additional modules

You can specify Python modules that should be imported and made available to your code using one or more `--import` options. This example uses the `textwrap` module to wrap the `content` column at 100 characters:

```
sqlite-utils convert content.db articles content \
    '"\n".join(textwrap.wrap(value, 100))' \
    --import=textwrap
```

This supports nested imports as well, for example to use ElementTree:

```
sqlite-utils convert content.db articles content \
    'xml.etree.ElementTree.fromstring(value).attrib["title"]' \
    --import=xml.etree.ElementTree
```

Using the debugger

If an error occurs while running your conversion operation you may see a message like this:

```
user-defined function raised exception
```

Add the `--pdb` option to catch the error and open the Python debugger at that point. The conversion operation will exit after you type `q` in the debugger.

Here's an example debugging session. First, create a `articles` table with invalid XML in the `content` column:

```
echo '{"content": "This is not XML"}' | sqlite-utils insert content.db articles -
```

Now run the conversion with the `--pdb` option:

```
sqlite-utils convert content.db articles content \
    'xml.etree.ElementTree.fromstring(value).attrib["title"]' \
    --import=xml.etree.ElementTree \
    --pdb
```

When the error occurs the debugger will open:

```
Exception raised, dropping into pdb...: syntax error: line 1, column 0
> .../python3.11/xml/etree/ElementTree.py(1338)XML()
-> parser.feed(text)
(Pdb) args
text = 'This is not XML'
parser = <xml.etree.ElementTree.XMLParser object at 0x102c405e0>
(Pdb) q
```

`args` here shows the arguments to the current function in the stack. The Python pdb documentation has full details on the other available commands.

Defining a convert() function

Instead of providing a single line of code to be executed against each value, you can define a function called `convert(value)`.

This mechanism can be used to execute one-off initialization code that runs once at the start of the conversion run.

The following example adds a new `score` column, then updates it to list a random number - after first seeding the random number generator to ensure that multiple runs produce the same results:

```
sqlite-utils add-column content.db articles score float --not-null-default 1.0
sqlite-utils convert content.db articles score '
import random
random.seed(10)

def convert(value):
    return random.random()
'
```

sqlite-utils convert recipes

Various built-in recipe functions are available for common operations. These are:

r.jsonsplit(value, delimiter=',', type=<class 'str'>)
Convert a string like `a,b,c` into a JSON array `["a", "b", "c"]`

The `delimiter` parameter can be used to specify a different delimiter.

The `type` parameter can be set to `float` or `int` to produce a JSON array of different types, for example if the column's string value was `1.2,3,4.5` the following:

```
r.jsonsplit(value, type=float)
```

Would produce an array like this: `[1.2, 3.0, 4.5]`

r.parsedate(value, dayfirst=False, yearfirst=False, errors=None)
Parse a date and convert it to ISO date format: `yyyy-mm-dd`

In the case of dates such as `03/04/05` U.S. `MM/DD/YY` format is assumed - you can use `dayfirst=True` or `yearfirst=True` to change how these ambiguous dates are interpreted.

Use the `errors=` parameter to specify what should happen if a value cannot be parsed.

By default, if any value cannot be parsed an error will be occurred and all values will be left as they were.

Set `errors=r.IGNORE` to ignore any values that cannot be parsed, leaving them unchanged.

Set `errors=r.SET_NULL` to set any values that cannot be parsed to `null`.

r.parsedatetime(value, dayfirst=False, yearfirst=False, errors=None)
Parse a datetime and convert it to ISO datetime format: `yyyy-mm-ddTHH:MM:SS`

These recipes can be used in the code passed to `sqlite-utils convert` like this:

```
sqlite-utils convert my.db mytable mycolumn \
  'r.jsonsplit(value)'
```

You can also pass the recipe function directly without the `(value)` part - sqlite-utils will detect that it is a callable and use it automatically:

```
sqlite-utils convert my.db mytable mycolumn r.parsedate
```

This shorter syntax works for any callable, including functions from imported modules:

```
sqlite-utils convert my.db mytable mycolumn json.loads --import json
```

To use any of the documented parameters, use the full function call syntax:

```
sqlite-utils convert my.db mytable mycolumn \
  'r.jsonsplit(value, delimiter=":")'
```

Saving the result to a different column

The `--output` and `--output-type` options can be used to save the result of the conversion to a separate column, which will be created if that column does not already exist:

```
sqlite-utils convert content.db articles headline 'value.upper()' \
  --output headline_upper
```

The type of the created column defaults to `text`, but a different column type can be specified using `--output-type`. This example will create a new floating point column called `id_as_a_float` with a copy of each item's ID increased by 0.5:

```
sqlite-utils convert content.db articles id 'float(value) + 0.5' \
  --output id_as_a_float \
  --output-type float
```

You can drop the original column at the end of the operation by adding `--drop`.

Converting a column into multiple columns

Sometimes you may wish to convert a single column into multiple derived columns. For example, you may have a `location` column containing `latitude,longitude` values which you wish to split out into separate `latitude` and `longitude` columns.

You can achieve this using the `--multi` option to `sqlite-utils convert`. This option expects your Python code to return a Python dictionary: new columns well be created and populated for each of the keys in that dictionary.

For the `latitude,longitude` example you would use the following:

```
sqlite-utils convert demo.db places location \
'bits = value.split(",")
return {
  "latitude": float(bits[0]),
  "longitude": float(bits[1]),
}' --multi
```

The type of the returned values will be taken into account when creating the new columns. In this example, the resulting database schema will look like this:

```
CREATE TABLE "places" (
    "location" TEXT,
    "latitude" REAL,
    "longitude" REAL
);
```

The code function can also return `None`, in which case its output will be ignored. You can drop the original column at the end of the operation by adding `--drop`.

1.2.22 Creating tables

Most of the time creating tables by inserting example data is the quickest approach. If you need to create an empty table in advance of inserting data you can do so using the `create-table` command:

```
sqlite-utils create-table mydb.db mytable id integer name text --pk=id
```

This will create a table called `mytable` with two columns - an integer `id` column and a text `name` column. It will set the `id` column to be the primary key.

You can pass as many column-name column-type pairs as you like. Valid types are `integer`, `text`, `float` and `blob`.

Pass `--pk` more than once for a compound primary key that covers multiple columns.

You can specify columns that should be NOT NULL using `--not-null colname`. You can specify default values for columns using `--default colname defaultvalue`.

```
sqlite-utils create-table mydb.db mytable \
    id integer \
    name text \
    age integer \
    is_good integer \
    --not-null name \
    --not-null age \
    --default is_good 1 \
    --pk=id
```

```
sqlite-utils tables mydb.db --schema -t
```

```
table    schema
-------  --------------------------------
mytable  CREATE TABLE "mytable" (
            "id" INTEGER PRIMARY KEY,
            "name" TEXT NOT NULL,
            "age" INTEGER NOT NULL,
            "is_good" INTEGER DEFAULT '1'
        )
```

You can specify foreign key relationships between the tables you are creating using `--fk colname othertable othercolumn`:

```
sqlite-utils create-table books.db authors \
    id integer \
    name text \
    --pk=id

sqlite-utils create-table books.db books \
    id integer \
    title text \
    author_id integer \
    --pk=id \
    --fk author_id authors id
```

```
sqlite-utils tables books.db --schema -t
```

```
table    schema
-------  -------------------------------------------------
authors  CREATE TABLE "authors" (
            "id" INTEGER PRIMARY KEY,
            "name" TEXT
         )
books    CREATE TABLE "books" (
            "id" INTEGER PRIMARY KEY,
            "title" TEXT,
            "author_id" INTEGER REFERENCES "authors"("id")
         )
```

You can create a table in SQLite STRICT mode using `--strict`:

```
sqlite-utils create-table mydb.db mytable id integer name text --strict
```

```
sqlite-utils tables mydb.db --schema -t
```

```
table    schema
-------  ------------------------
mytable  CREATE TABLE "mytable" (
            "id" INTEGER,
            "name" TEXT
         ) STRICT
```

If a table with the same name already exists, you will get an error. You can choose to silently ignore this error with `--ignore`, or you can replace the existing table with a new, empty table using `--replace`.

You can also pass `--transform` to transform the existing table to match the new schema. See *Explicitly creating a table* in the Python library documentation for details of how this option works.

1.2.23 Renaming a table

Yo ucan rename a table using the `rename-table` command:

```
sqlite-utils rename-table mydb.db oldname newname
```

Pass `--ignore` to ignore any errors caused by the table not existing, or the new name already being in use.

1.2.24 Duplicating tables

The `duplicate` command duplicates a table - creating a new table with the same schema and a copy of all of the rows:

```
sqlite-utils duplicate books.db authors authors_copy
```

1.2.25 Dropping tables

You can drop a table using the `drop-table` command:

```
sqlite-utils drop-table mydb.db mytable
```

Use `--ignore` to ignore the error if the table does not exist.

1.2.26 Transforming tables

The `transform` command allows you to apply complex transformations to a table that cannot be implemented using a regular SQLite `ALTER TABLE` command. See *Transforming a table* for details of how this works. The `transform` command preserves a table's `STRICT` mode.

```
sqlite-utils transform mydb.db mytable \
    --drop column1 \
    --rename column2 column_renamed
```

Every option for this table (with the exception of `--pk-none`) can be specified multiple times. The options are as follows:

`--type column-name new-type`
Change the type of the specified column. Valid types are `integer`, `text`, `float`, `blob`.

`--drop column-name`
Drop the specified column.

`--rename column-name new-name`
Rename this column to a new name.

`--column-order column`
Use this multiple times to specify a new order for your columns. `-o` shortcut is also available.

`--not-null column-name`
Set this column as `NOT NULL`.

`--not-null-false column-name`
For a column that is currently set as `NOT NULL`, remove the `NOT NULL`.

`--pk column-name`
Change the primary key column for this table. Pass `--pk` multiple times if you want to create a compound primary key.

`--pk-none`
Remove the primary key from this table, turning it into a `rowid` table.

`--default column-name value`
Set the default value of this column.

`--default-none column`
Remove the default value for this column.

`--drop-foreign-key column`
Drop the specified foreign key.

`--add-foreign-key column other_table other_column`
Add a foreign key constraint to `column` pointing to `other_table.other_column`.

If you want to see the SQL that will be executed to make the change without actually executing it, add the `--sql` flag. For example:

```
sqlite-utils transform fixtures.db roadside_attractions \
    --rename pk id \
    --default name Untitled \
    --column-order id \
    --column-order longitude \
    --column-order latitude \
    --drop address \
    --sql
```

```
CREATE TABLE "roadside_attractions_new_4033a60276b9" (
   "id" INTEGER PRIMARY KEY,
   "longitude" FLOAT,
   "latitude" FLOAT,
   "name" TEXT DEFAULT 'Untitled'
);
INSERT INTO "roadside_attractions_new_4033a60276b9" ("longitude", "latitude", "id", "name
↪")
   SELECT "longitude", "latitude", "pk", "name" FROM "roadside_attractions";
DROP TABLE "roadside_attractions";
ALTER TABLE "roadside_attractions_new_4033a60276b9" RENAME TO "roadside_attractions";
```

Adding a primary key to a rowid table

SQLite tables that are created without an explicit primary key are created as rowid tables. They still have a numeric primary key which is available in the `rowid` column, but that column is not included in the output of `select *`. Here's an example:

```
echo '[{"name": "Azi"}, {"name": "Suna"}]' | \
    sqlite-utils insert chickens.db chickens -
sqlite-utils schema chickens.db
```

```
CREATE TABLE "chickens" (
   "name" TEXT
);
```

```
sqlite-utils chickens.db 'select * from chickens'
```

```
[{"name": "Azi"},
 {"name": "Suna"}]
```

```
sqlite-utils chickens.db 'select rowid, * from chickens'
```

```
[{"rowid": 1, "name": "Azi"},
 {"rowid": 2, "name": "Suna"}]
```

You can use `sqlite-utils transform ... --pk id` to add a primary key column called `id` to the table. The primary key will be created as an `INTEGER PRIMARY KEY` and the existing `rowid` values will be copied across to it. It will automatically increment as new rows are added to the table:

```
sqlite-utils transform chickens.db chickens --pk id
```

```
sqlite-utils schema chickens.db
```

```
CREATE TABLE "chickens" (
   "id" INTEGER PRIMARY KEY,
   "name" TEXT
);
```

```
sqlite-utils chickens.db 'select * from chickens'
```

```
[{"id": 1, "name": "Azi"},
 {"id": 2, "name": "Suna"}]
```

```
echo '{"name": "Cardi"}' | sqlite-utils insert chickens.db chickens -
```

```
sqlite-utils chickens.db 'select * from chickens'
```

```
[{"id": 1, "name": "Azi"},
 {"id": 2, "name": "Suna"},
 {"id": 3, "name": "Cardi"}]
```

1.2.27 Extracting columns into a separate table

The `sqlite-utils extract` command can be used to extract specified columns into a separate table.

Take a look at the Python API documentation for *Extracting columns into a separate table* for a detailed description of how this works, including examples of table schemas before and after running an extraction operation.

Rows where every extracted column is `null` are not extracted - those rows get a `null` value in their new foreign key column and no record is created for them in the lookup table.

The command takes a database, table and one or more columns that should be extracted. To extract the `species` column from the `trees` table you would run:

```
sqlite-utils extract my.db trees species
```

This would produce the following schema:

```
CREATE TABLE "trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "species_id" INTEGER,
    FOREIGN KEY(species_id) REFERENCES species(id)
);
CREATE TABLE "species" (
    "id" INTEGER PRIMARY KEY,
    "species" TEXT
);
CREATE UNIQUE INDEX "idx_species_species"
    ON "species" ("species");
```

The command takes the following options:

--table TEXT
: The name of the lookup to extract columns to. This defaults to using the name of the columns that are being extracted.

--fk-column TEXT
: The name of the foreign key column to add to the table. Defaults to `columnname_id`.

--rename <TEXT TEXT>
: Use this option to rename the columns created in the new lookup table.

--silent
: Don't display the progress bar.

Here's a more complex example that makes use of these options. It converts this CSV file full of global power plants into SQLite, then extracts the `country` and `country_long` columns into a separate `countries` table:

```
wget 'https://github.com/wri/global-power-plant-database/blob/232a6666/output_database/
↪global_power_plant_database.csv?raw=true'
sqlite-utils insert global.db power_plants \
    'global_power_plant_database.csv?raw=true' --csv
# Extract those columns:
sqlite-utils extract global.db power_plants country country_long \
    --table countries \
    --fk-column country_id \
    --rename country_long name
```

After running the above, the command `sqlite-utils schema global.db` reveals the following schema:

```
CREATE TABLE "countries" (
   "id" INTEGER PRIMARY KEY,
   "country" TEXT,
   "name" TEXT
);
CREATE TABLE "power_plants" (
   "country_id" INTEGER,
   "name" TEXT,
   "gppd_idnr" TEXT,
   "capacity_mw" TEXT,
   "latitude" TEXT,
   "longitude" TEXT,
   "primary_fuel" TEXT,
   "other_fuel1" TEXT,
   "other_fuel2" TEXT,
   "other_fuel3" TEXT,
   "commissioning_year" TEXT,
   "owner" TEXT,
   "source" TEXT,
   "url" TEXT,
   "geolocation_source" TEXT,
   "wepp_id" TEXT,
   "year_of_capacity_data" TEXT,
   "generation_gwh_2013" TEXT,
   "generation_gwh_2014" TEXT,
   "generation_gwh_2015" TEXT,
   "generation_gwh_2016" TEXT,
   "generation_gwh_2017" TEXT,
   "generation_data_source" TEXT,
   "estimated_generation_gwh" TEXT,
   FOREIGN KEY("country_id") REFERENCES "countries"("id")
);
CREATE UNIQUE INDEX "idx_countries_country_name"
    ON "countries" ("country", "name");
```

1.2.28 Creating views

You can create a view using the `create-view` command:

```
sqlite-utils create-view mydb.db version "select sqlite_version()"
```

```
sqlite-utils mydb.db "select * from version"
```

```
[{"sqlite_version()": "3.31.1"}]
```

Use `--replace` to replace an existing view of the same name, and `--ignore` to do nothing if a view already exists.

1.2.29 Dropping views

You can drop a view using the `drop-view` command:

```
sqlite-utils drop-view myview
```

Use `--ignore` to ignore the error if the view does not exist.

1.2.30 Adding columns

You can add a column using the `add-column` command:

```
sqlite-utils add-column mydb.db mytable nameofcolumn text
```

The last argument here is the type of the column to be created. This can be one of:

- `text` or `str`
- `integer` or `int`
- `float`
- `blob` or `bytes`

This argument is optional and defaults to `text`.

You can add a column that is a foreign key reference to another table using the `--fk` option:

```
sqlite-utils add-column mydb.db dogs species_id --fk species
```

This will automatically detect the name of the primary key on the species table and use that (and its type) for the new column.

You can explicitly specify the column you wish to reference using `--fk-col`:

```
sqlite-utils add-column mydb.db dogs species_id --fk species --fk-col ref
```

You can set a `NOT NULL DEFAULT 'x'` constraint on the new column using `--not-null-default`:

```
sqlite-utils add-column mydb.db dogs friends_count integer --not-null-default 0
```

1.2.31 Adding columns automatically on insert/update

You can use the `--alter` option to automatically add new columns if the data you are inserting or upserting is of a different shape:

```
sqlite-utils insert dogs.db dogs new-dogs.json --pk=id --alter
```

1.2.32 Adding foreign key constraints

The `add-foreign-key` command can be used to add new foreign key references to an existing table - something which SQLite's `ALTER TABLE` command does not support.

To add a foreign key constraint pointing the `books.author_id` column to `authors.id` in another table, do this:

```
sqlite-utils add-foreign-key books.db books author_id authors id
```

If you omit the other table and other column references `sqlite-utils` will attempt to guess them - so the above example could instead look like this:

```
sqlite-utils add-foreign-key books.db books author_id
```

Add `--ignore` to ignore an existing foreign key (as opposed to returning an error):

```
sqlite-utils add-foreign-key books.db books author_id --ignore
```

See *Adding foreign key constraints* in the Python API documentation for further details, including how the automatic table guessing mechanism works.

Adding multiple foreign keys at once

Adding a foreign key requires a `VACUUM`. On large databases this can be an expensive operation, so if you are adding multiple foreign keys you can combine them into one operation (and hence one `VACUUM`) using `add-foreign-keys`:

```
sqlite-utils add-foreign-keys books.db \
    books author_id authors id \
    authors country_id countries id
```

When you are using this command each foreign key needs to be defined in full, as four arguments - the table, column, other table and other column.

Adding indexes for all foreign keys

If you want to ensure that every foreign key column in your database has a corresponding index, you can do so like this:

```
sqlite-utils index-foreign-keys books.db
```

1.2.33 Setting defaults and not null constraints

You can use the `--not-null` and `--default` options (to both `insert` and `upsert`) to specify columns that should be `NOT NULL` or to set database defaults for one or more specific columns:

```
sqlite-utils insert dogs.db dogs_with_scores dogs-with-scores.json \
    --not-null=age \
    --not-null=name \
    --default age 2 \
    --default score 5
```

1.2.34 Creating indexes

You can add an index to an existing table using the `create-index` command:

```
sqlite-utils create-index mydb.db mytable col1 [col2...]
```

This can be used to create indexes against a single column or multiple columns.

The name of the index will be automatically derived from the table and columns. To specify a different name, use `--name=name_of_index`.

Use the `--unique` option to create a unique index.

Use `--if-not-exists` to avoid attempting to create the index if one with that name already exists.

To add an index on a column in descending order, prefix the column with a hyphen. Since this can be confused for a command-line option you need to construct that like this:

```
sqlite-utils create-index mydb.db mytable -- col1 -col2 col3
```

This will create an index on that table on `(col1, col2 desc, col3)`.

If your column names are already prefixed with a hyphen you'll need to manually execute a `CREATE INDEX` SQL statement to add indexes to them rather than using this tool.

Add the `--analyze` option to run `ANALYZE` against the index after it has been created.

1.2.35 Configuring full-text search

You can enable SQLite full-text search on a table and a set of columns like this:

```
sqlite-utils enable-fts mydb.db documents title summary
```

This will use SQLite's FTS5 module by default. Use `--fts4` if you want to use FTS4:

```
sqlite-utils enable-fts mydb.db documents title summary --fts4
```

The `enable-fts` command will populate the new index with all existing documents. If you later add more documents you will need to use `populate-fts` to cause them to be indexed as well:

```
sqlite-utils populate-fts mydb.db documents title summary
```

A better solution here is to use database triggers. You can set up database triggers to automatically update the full-text index using the `--create-triggers` option when you first run `enable-fts`:

```
sqlite-utils enable-fts mydb.db documents title summary --create-triggers
```

To set a custom FTS tokenizer, e.g. to enable Porter stemming, use `--tokenize=`:

```
sqlite-utils populate-fts mydb.db documents title summary --tokenize=porter
```

To remove the FTS tables and triggers you created, use `disable-fts`:

```
sqlite-utils disable-fts mydb.db documents
```

To rebuild one or more FTS tables (see *Rebuilding a full-text search table*), use `rebuild-fts`:

```
sqlite-utils rebuild-fts mydb.db documents
```

You can rebuild every FTS table by running `rebuild-fts` without passing any table names:

```
sqlite-utils rebuild-fts mydb.db
```

1.2.36 Executing searches

Once you have configured full-text search for a table, you can search it using `sqlite-utils search`:

```
sqlite-utils search mydb.db documents searchterm
```

This command accepts the same output options as `sqlite-utils query`: `--table`, `--csv`, `--tsv`, `--nl` etc.

By default it shows the most relevant matches first. You can specify a different sort order using the `-o` option, which can take a column or a column followed by `desc`:

```
# Sort by rowid
sqlite-utils search mydb.db documents searchterm -o rowid
# Sort by created in descending order
sqlite-utils search mydb.db documents searchterm -o 'created desc'
```

SQLite advanced search syntax is enabled by default. To run a search with automatic quoting applied to the terms to avoid them being potentially interpreted as advanced search syntax use the `--quote` option.

You can specify a subset of columns to be returned using the `-c` option one or more times:

```
sqlite-utils search mydb.db documents searchterm -c title -c created
```

By default all search results will be returned. You can use `--limit 20` to return just the first 20 results.

Use the `--sql` option to output the SQL that would be executed, rather than running the query:

```
sqlite-utils search mydb.db documents searchterm --sql
```

```
with original as (
    select
        rowid,
        *
    from "documents"
)
select
    "original".*
from
    "original"
    join "documents_fts" on "original".rowid = "documents_fts".rowid
where
    "documents_fts" match :query
order by
    "documents_fts".rank
```

1.2.37 Enabling cached counts

`select count(*)` queries can take a long time against large tables. `sqlite-utils` can speed these up by adding triggers to maintain a `_counts` table, see *Cached table counts using triggers* for details.

The `sqlite-utils enable-counts` command can be used to configure these triggers, either for every table in the database or for specific tables.

```
# Configure triggers for every table in the database
sqlite-utils enable-counts mydb.db

# Configure triggers just for specific tables
sqlite-utils enable-counts mydb.db table1 table2
```

If the `_counts` table ever becomes out-of-sync with the actual table counts you can repair it using the `reset-counts` command:

```
sqlite-utils reset-counts mydb.db
```

1.2.38 Optimizing index usage with ANALYZE

The SQLite ANALYZE command builds a table of statistics which the query planner can use to make better decisions about which indexes to use for a given query.

You should run `ANALYZE` if your database is large and you do not think your indexes are being efficiently used.

To run `ANALYZE` against every index in a database, use this:

```
sqlite-utils analyze mydb.db
```

You can run it against specific tables, or against specific named indexes, by passing them as optional arguments:

```
sqlite-utils analyze mydb.db mytable idx_mytable_name
```

You can also run `ANALYZE` as part of another command using the `--analyze` option. This is supported by the `create-index`, `insert` and `upsert` commands.

1.2.39 Vacuum

You can run VACUUM to optimize your database like so:

```
sqlite-utils vacuum mydb.db
```

1.2.40 Optimize

The optimize command can dramatically reduce the size of your database if you are using SQLite full-text search. It runs OPTIMIZE against all of your FTS4 and FTS5 tables, then runs VACUUM.

If you just want to run OPTIMIZE without the VACUUM, use the `--no-vacuum` flag.

```
# Optimize all FTS tables and then VACUUM
sqlite-utils optimize mydb.db

# Optimize but skip the VACUUM
sqlite-utils optimize --no-vacuum mydb.db
```

To optimize specific tables rather than every FTS table, pass those tables as extra arguments:

```
sqlite-utils optimize mydb.db table_1 table_2
```

1.2.41 WAL mode

You can enable Write-Ahead Logging for a database file using the `enable-wal` command:

```
sqlite-utils enable-wal mydb.db
```

You can disable WAL mode using `disable-wal`:

```
sqlite-utils disable-wal mydb.db
```

Both of these commands accept one or more database files as arguments.

1.2.42 Dumping the database to SQL

The `dump` command outputs a SQL dump of the schema and full contents of the specified database file:

```
sqlite-utils dump mydb.db
BEGIN TRANSACTION;
CREATE TABLE ...
...
COMMIT;
```

1.2.43 Loading SQLite extensions

Many of these commands have the ability to load additional SQLite extensions using the `--load-extension=/path/to/extension` option - use `--help` to check for support, e.g. `sqlite-utils rows --help`.

This option can be applied multiple times to load multiple extensions.

Since SpatiaLite is commonly used with SQLite, the value `spatialite` is special: it will search for SpatiaLite in the most common installation locations, saving you from needing to remember exactly where that module is located:

```
sqlite-utils memory "select spatialite_version()" --load-extension=spatialite
```

```
[{"spatialite_version()": "4.3.0a"}]
```

1.2.44 SpatiaLite helpers

SpatiaLite adds geographic capability to SQLite (similar to how PostGIS builds on PostgreSQL). The SpatiaLite cookbook is a good resource for learning what's possible with it.

You can convert an existing table to a geographic table by adding a geometry column, use the `sqlite-utils add-geometry-column` command:

```
sqlite-utils add-geometry-column spatial.db locations geometry --type POLYGON --srid 4326
```

The table (`locations` in the example above) must already exist before adding a geometry column. Use `sqlite-utils create-table` first, then `add-geometry-column`.

Use the `--type` option to specify a geometry type. By default, `add-geometry-column` uses a generic `GEOMETRY`, which will work with any type, though it may not be supported by some desktop GIS applications.

Eight (case-insensitive) types are allowed:

- POINT
- LINESTRING
- POLYGON

- MULTIPOINT
- MULTILINESTRING
- MULTIPOLYGON
- GEOMETRYCOLLECTION
- GEOMETRY

Adding spatial indexes

Once you have a geometry column, you can speed up bounding box queries by adding a spatial index:

```
sqlite-utils create-spatial-index spatial.db locations geometry
```

See this SpatiaLite Cookbook recipe for examples of how to use a spatial index.

1.2.45 Installing packages

The *convert command* and the *insert –convert* and *query –functions* options can be provided with a Python script that imports additional modules from the `sqlite-utils` environment.

You can install packages from PyPI directly into the correct environment using `sqlite-utils install <package>`. This is a wrapper around `pip install`.

```
sqlite-utils install beautifulsoup4
```

Use `-U` to upgrade an existing package.

1.2.46 Uninstalling packages

You can uninstall packages that were installed using `sqlite-utils install` with `sqlite-utils uninstall <package>`:

```
sqlite-utils uninstall beautifulsoup4
```

Use `-y` to skip the request for confirmation.

1.3 sqlite_utils Python library

- *Creating a spatial index*

1.3.1 Getting started

Here's how to create a new SQLite database file containing a new `chickens` table, populated with four records:

```
from sqlite_utils import Database

db = Database("chickens.db")
db.table("chickens").insert_all([{
    "name": "Azi",
    "color": "blue",
}, {
    "name": "Lila",
    "color": "blue",
}, {
    "name": "Suna",
    "color": "gold",
}, {
    "name": "Cardi",
    "color": "black",
}])
```

The inserted rows are saved to the database file straight away - methods like `insert_all()` commit their own changes, so no `commit()` call is needed. See *Transactions and saving your changes* for how this works.

You can loop through those rows like this:

```
for row in db.table("chickens").rows:
    print(row)
```

Which outputs the following:

```
{'name': 'Azi', 'color': 'blue'}
{'name': 'Lila', 'color': 'blue'}
{'name': 'Suna', 'color': 'gold'}
{'name': 'Cardi', 'color': 'black'}
```

To run a SQL query, use *db.query()*:

```
for row in db.query("""
    select color, count(*)
    from chickens group by color
    order by count(*) desc
"""):
    print(row)
```

Which outputs:

```
{'color': 'blue', 'count(*)': 2}
{'color': 'gold', 'count(*)': 1}
{'color': 'black', 'count(*)': 1}
```

1.3.2 Connecting to or creating a database

Database objects are constructed by passing in either a path to a file on disk or an existing SQLite3 database connection:

```
from sqlite_utils import Database

db = Database("my_database.db")
```

This will create `my_database.db` if it does not already exist.

If you want to recreate a database from scratch (first removing the existing file from disk if it already exists) you can use the `recreate=True` argument:

```
db = Database("my_database.db", recreate=True)
```

Instead of a file path you can pass in an existing SQLite connection:

```
import sqlite3

db = Database(sqlite3.connect("my_database.db"))
```

The connection must use Python's default transaction handling. Connections created with the Python 3.12+ `sqlite3.connect(..., autocommit=True)` or `autocommit=False` options are rejected with a `sqlite_utils.db.TransactionError` - see *Supported connection modes*.

If you want to create an in-memory database, you can do so like this:

```
db = Database(memory=True)
```

You can also create a named in-memory database. Unlike regular memory databases these can be accessed by multiple threads, provided at least one reference to the database still exists. *del db* will clear the database from memory.

```
db = Database(memory_name="my_shared_database")
```

Connections use `PRAGMA recursive_triggers=on` by default. If you don't want to use recursive triggers you can turn them off using:

```
db = Database(memory=True, recursive_triggers=False)
```

By default, any *sqlite-utils plugins* that implement the *prepare_connection(conn)* hook will be executed against the connection when you create the `Database` object. You can opt out of executing plugins using `execute_plugins=False` like this:

```
db = Database(memory=True, execute_plugins=False)
```

You can pass `strict=True` to enable SQLite STRICT mode for all tables created using this database object:

```
db = Database("my_database.db", strict=True)
```

Closing a database

Database objects maintain a connection to the underlying SQLite database. You can explicitly close this connection using the `.close()` method:

```
db = Database("my_database.db")
# ... use the database ...
db.close()
```

The `Database` object also works as a context manager, which will automatically close the connection when the `with` block exits:

```
with Database("my_database.db") as db:
    db["my_table"].insert({"name": "Example"})
# Connection is automatically closed here
```

Exiting the block is equivalent to calling `db.close()`: the connection is closed and any transaction still open at that point is rolled back. This matches SQLite's own behavior when a connection closes.

This rarely matters in practice. Everything that writes to the database - including raw `db.execute()` statements - commits automatically, so a transaction can only be open here if you explicitly started one with `db.begin()` and have not yet committed it. In that case the decision to commit stays with you: committing automatically on exit could silently persist half-finished work, for example if your code returned early from the block. Call `db.commit()` when the work is complete.

Note this differs from the `sqlite3.Connection` context manager in the standard library, which commits on success but does not close the connection. See *Transactions and saving your changes* for the full transaction model.

Attaching additional databases

SQLite supports cross-database SQL queries, which can join data from tables in more than one database file.

You can attach an additional database using the `.attach()` method, providing an alias to use for that database and the path to the SQLite file on disk.

```
db = Database("first.db")
db.attach("second", "second.db")
# Now you can run queries like this one:
print(db.query("""
select * from table_in_first
    union all
select * from second.table_in_second
"""))
```

You can reference tables in the attached database using the alias value you passed to `db.attach(alias, filepath)` as a prefix, for example the `second.table_in_second` reference in the SQL query above.

Tracing queries

You can use the `tracer` mechanism to see SQL queries that are being executed by SQLite. A tracer is a function that you provide which will be called with `sql` and `params` arguments every time SQL is executed, for example:

```
def tracer(sql, params):
    print("SQL: {} - params: {}".format(sql, params))
```

You can pass this function to the `Database()` constructor like so:

```
db = Database(memory=True, tracer=tracer)
```

You can also turn on a tracer function temporarily for a block of code using the `with db.tracer(...)` context manager:

```
db = Database(memory=True)
# ... later
with db.tracer(print):
    db.table("dogs").insert({"name": "Cleo"})
```

This example will print queries only for the duration of the `with` block.

1.3.3 Executing queries

The `Database` class offers several methods for directly executing SQL queries.

db.query(sql, params)

The `db.query(sql)` function executes a SQL query and returns an iterator over Python dictionaries representing the resulting rows:

```
db = Database(memory=True)
db.table("dogs").insert_all([{"name": "Cleo"}, {"name": "Pancakes"}])
for row in db.query("select * from dogs"):
    print(row)
# Outputs:
# {'name': 'Cleo'}
# {'name': 'Pancakes'}
```

The SQL query is executed as soon as `db.query()` is called. The resulting rows are fetched lazily as you iterate, so large result sets are not loaded into memory all at once. Because execution is immediate, an error in your SQL will raise an exception straight away, and a statement such as `INSERT ... RETURNING` will take effect - and be committed, unless a transaction is open - even if you do not iterate over its results.

`db.query()` can only be used with SQL that returns rows. Passing a statement that returns no rows - an `INSERT` or `UPDATE` without a `RETURNING` clause, for example - will raise a `ValueError`. The rejected statement is rolled back, so it has no effect on the database. Use *db.execute()* for those statements instead.

There is one exception to the rolled-back guarantee: a `PRAGMA` statement that returns no rows, such as `PRAGMA user_version = 5`, still raises a `ValueError` but will already have taken effect. Some PRAGMA statements refuse to run inside a transaction, so PRAGMAs are executed outside the savepoint that is used to roll back other rejected statements. Use `db.execute()` for PRAGMA statements that do not return rows.

If a query returns more than one column with the same name - a join between two tables that share column names, for example - later occurrences are renamed with a numeric suffix, so every value is included in the dictionary:

```
row = next(db.query("select 1 as id, 2 as id, 3 as id"))
print(row)
# Outputs:
# {'id': 1, 'id_2': 2, 'id_3': 3}
```

A suffix that would collide with another column in the query is skipped - `select 1 as id, 2 as id, 3 as id_2` returns `{'id': 1, 'id_3': 2, 'id_2': 3}`. The same renaming is applied by `table.rows_where()` and `table.search()`.

db.execute(sql, params)

The `db.execute()` and `db.executescript()` methods provide wrappers around `.execute()` and `.executescript()` on the underlying SQLite connection. These wrappers log to the *tracer function* if one has been registered.

`db.execute(sql)` returns a sqlite3.Cursor that was used to execute the SQL.

```
db = Database(memory=True)
db.table("dogs").insert({"name": "Cleo"})
cursor = db.execute("update dogs set name = 'Cleopaws'")
print(cursor.rowcount)
```

(continues on next page)

(continued from previous page)

```
# Outputs the number of rows affected by the update
# In this case 2
```

Other cursor methods such as `.fetchone()` and `.fetchall()` are also available, see the standard library documentation.

> **Note**
>
> Write statements executed this way are committed automatically, unless a transaction is already open in which case they become part of it - see *Raw SQL writes with db.execute()*.

Passing parameters

Both `db.query()` and `db.execute()` accept an optional second argument for parameters to be passed to the SQL query.

This can take the form of either a tuple/list or a dictionary, depending on the type of parameters used in the query. Values passed in this way will be correctly quoted and escaped, helping avoid SQL injection vulnerabilities.

`?` parameters in the SQL query can be filled in using a list:

```
db.execute("update dogs set name = ?", ["Cleopaws"])
# This will rename ALL dogs to be called "Cleopaws"
```

Named parameters using `:name` can be filled using a dictionary:

```
dog = next(db.query(
    "select rowid, name from dogs where name = :name",
    {"name": "Cleopaws"}
))
# dog is now {'rowid': 1, 'name': 'Cleopaws'}
```

In this example `next()` is used to retrieve the first result in the iterator returned by the `db.query()` method.

1.3.4 Transactions and saving your changes

Every method in this library that writes to the database - `insert()`, `upsert()`, `update()`, `delete()`, `delete_where()`, `transform()`, `create_table()`, `create_index()`, `enable_fts()` and the rest - runs inside its own transaction and commits it before returning. Your changes are saved to disk as soon as the method call finishes:

```
db = Database("data.db")
db.table("news").insert({"headline": "Dog wins award"})
# The new row is already saved - no commit() required
```

The same applies to raw SQL executed with *db.execute()* - a write statement is committed as soon as it has run.

You never need to call `commit()`, and you do not need to close the database to persist your changes. There are exactly two situations where you need to think about transactions:

1. You want to group several write operations together, so they either all succeed or all fail - use *db.atomic()*.
2. You are *managing a transaction yourself* with `db.begin()`, in which case nothing is committed until you commit - the library will never commit a transaction you opened.

Grouping changes with db.atomic()

Use `db.atomic()` to group multiple operations in a single transaction:

```
with db.atomic():
    db.table("dogs").insert({"id": 1, "name": "Cleo"}, pk="id")
    db.table("dogs").insert({"id": 2, "name": "Pancakes"})
```

The transaction commits when the block exits. If an exception is raised, changes made inside the block will be rolled back.

`db.atomic()` can be nested. Nested blocks use SQLite savepoints, so an exception in an inner block can roll back to that savepoint without rolling back the entire outer transaction:

```
with db.atomic():
    db.table("dogs").insert({"id": 1, "name": "Cleo"}, pk="id")
    try:
        with db.atomic():
            db.table("dogs").insert({"id": 2, "name": "Pancakes"})
            raise ValueError("skip this one")
    except ValueError:
        pass
    db.table("dogs").insert({"id": 3, "name": "Marnie"})
```

The transaction is opened with a deferred `BEGIN` - SQLite takes the necessary locks when the first statement inside the block runs.

Raw SQL writes with db.execute()

Write statements executed with *db.execute()* follow the same rule as everything else: they are committed automatically as soon as they have run.

```
db.execute("insert into news (headline) values (?)", ["Dog wins award"])
# Already committed
```

If a transaction is open - because the call happens inside a `db.atomic()` block, or after `db.begin()` - the statement becomes part of that transaction instead, and commits when the transaction commits:

```
with db.atomic():
    db.execute("insert into news (headline) values (?)", ["Dog wins award"])
    db.execute("insert into news (headline) values (?)", ["Cat unimpressed"])
# Both rows committed together
```

One corner case: a row-returning write such as `INSERT ... RETURNING` executed through `db.execute()` cannot be auto-committed, because its rows have not been read yet - call `db.commit()` after fetching them, or use *db.query()* for those statements, which executes the write and commits it immediately.

Managing transactions yourself

You can take full manual control using the `db.begin()`, `db.commit()` and `db.rollback()` methods:

```
db.begin()
db.table("news").insert({"headline": "Dog wins award"})
if all_looks_good:
    db.commit()
```

(continues on next page)

(continued from previous page)

```
else:
    db.rollback()
```

`db.begin()` raises `sqlite3.OperationalError` if a transaction is already open. `db.commit()` and `db.rollback()` do nothing if there is no open transaction.

The library will never commit a transaction you opened. If you call write methods such as `insert()` - or use `db.atomic()` - while your transaction is open, they participate in it using SQLite savepoints instead of committing: exiting an `atomic()` block releases its savepoint, but nothing is saved to disk until you commit the outer transaction yourself. If you roll back, their changes are rolled back too.

Two related safeguards to be aware of:

- `db.enable_wal()` and `db.disable_wal()` raise a `sqlite_utils.db.TransactionError` if called while a transaction is open, because changing the journal mode would commit it as a side effect.
- Closing the database - explicitly with `db.close()`, or by exiting a `with Database(...) as db:` block - rolls back any transaction that is still open, see *Closing a database*.

Supported connection modes

`db.atomic()` and the automatic per-method transactions require a connection in Python's default transaction handling mode. Passing a connection created with the Python 3.12+ `sqlite3.connect(..., autocommit=True)` or `autocommit=False` options to `Database()` raises a `sqlite_utils.db.TransactionError`.

This is because `commit()` and `rollback()` behave differently on those connections - under `autocommit=True` they are documented no-ops - which would cause every write made by this library to be silently discarded when the connection closed, rather than failing loudly.

1.3.5 Accessing tables

Tables are accessed using the `db.table()` method, like so:

```
table = db.table("my_table")
```

Using this factory function allows you to set *Table configuration options*. Additional keyword arguments to `db.table()` will be used if a further method call causes the table to be created.

The `db.table()` method will always return a *sqlite_utils.db.Table* instance, or raise a `sqlite_utils.db.NoTable` exception if the table name is actually a SQL view.

You can also access tables or views using dictionary-style syntax, like this:

```
table_or_view = db["my_table_or_view_name"]
```

If a table accessed using either of these methods does not yet exist, it will be created the first time you attempt to insert or upsert data into it.

1.3.6 Accessing views

SQL views can be accessed using the `db.view()` method, like so:

```
view = db.view("my_view")
```

This will return a *sqlite_utils.db.View* instance, or raise a `sqlite_utils.db.NoView` exception if the view does not exist.

1.3.7 Listing tables

You can list the names of tables in a database using the `.table_names()` method:

```
>>> db.table_names()
['dogs']
```

To see just the FTS4 tables, use `.table_names(fts4=True)`. For FTS5, use `.table_names(fts5=True)`.

You can also iterate through the table objects themselves using the `.tables` property:

```
>>> db.tables
[<Table dogs>]
```

1.3.8 Listing views

`.view_names()` shows you a list of views in the database:

```
>>> db.view_names()
['good_dogs']
```

You can iterate through view objects using the `.views` property:

```
>>> db.views
[<View good_dogs>]
```

View objects are similar to Table objects, except that any attempts to insert or update data will throw an error. The full list of methods and properties available on a view object is as follows:

- `columns`
- `columns_dict`
- `count`
- `schema`
- `rows`
- `rows_where(where, where_args, order_by, select)`
- `drop()`

1.3.9 Listing rows

To iterate through dictionaries for each of the rows in a table, use `.rows`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db.table("dogs").rows:
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
{'id': 2, 'age': 2, 'name': 'Pancakes'}
```

You can filter rows by a WHERE clause using `.rows_where(where, where_args)`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db.table("dogs").rows_where("age > ?", [3]):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

The first argument is a fragment of SQL. The second, optional argument is values to be passed to that fragment - you can use ? placeholders and pass an array, or you can use `:named` parameters and pass a dictionary, like this:

```
>>> for row in db.table("dogs").rows_where("age > :age", {"age": 3}):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

To return custom columns (instead of the default that uses `select *`) pass `select="column1, column2"`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> for row in db.table("dogs").rows_where(select='name, age'):
...     print(row)
{'name': 'Cleo', 'age': 4}
```

To specify an order, use the `order_by=` argument:

```
>>> for row in db.table("dogs").rows_where("age > 1", order_by="age"):
...     print(row)
{'id': 2, 'age': 2, 'name': 'Pancakes'}
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

You can use `order_by="age desc"` for descending order.

You can order all records in the table by excluding the `where` argument:

```
>>> for row in db.table("dogs").rows_where(order_by="age desc"):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
{'id': 2, 'age': 2, 'name': 'Pancakes'}
```

This method also accepts `offset=` and `limit=` arguments, for specifying an OFFSET and a LIMIT for the SQL query:

```
>>> for row in db.table("dogs").rows_where(order_by="age desc", limit=1):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

Counting rows

To count the number of rows that would be returned by a where filter, use `.count_where(where, where_args)`:

```
>>> db.table("dogs").count_where("age > ?", [1])
2
```

1.3.10 Listing rows with their primary keys

Sometimes it can be useful to retrieve the primary key along with each row, in order to pass that key (or primary key tuple) to the `.get()` or `.update()` methods.

The `.pks_and_rows_where()` method takes the same signature as `.rows_where()` (with the exception of the `select=` parameter) but returns a generator that yields pairs of `(primary key, row dictionary)`.

The primary key value will usually be a single value but can also be a tuple if the table has a compound primary key.

If the table is a `rowid` table (with no explicit primary key column) then that ID will be returned.

```
>>> db = sqlite_utils.Database(memory=True)
>>> db.table("dogs").insert({"name": "Cleo"})
>>> for pk, row in db.table("dogs").pks_and_rows_where():
...     print(pk, row)
1 {'rowid': 1, 'name': 'Cleo'}

>>> db.table("dogs_with_pk").insert({"id": 5, "name": "Cleo"}, pk="id")
>>> for pk, row in db.table("dogs_with_pk").pks_and_rows_where():
...     print(pk, row)
5 {'id': 5, 'name': 'Cleo'}

>>> db.table("dogs_with_compound_pk").insert(
...     {"species": "dog", "id": 3, "name": "Cleo"},
...     pk=("species", "id")
... )
>>> for pk, row in db.table("dogs_with_compound_pk").pks_and_rows_where():
...     print(pk, row)
('dog', 3) {'species': 'dog', 'id': 3, 'name': 'Cleo'}
```

1.3.11 Retrieving a specific record

You can retrieve a record by its primary key using `table.get()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db.table("dogs").get(1))
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

If the table has a compound primary key you can pass in the primary key values as a tuple:

```
>>> db.table("compound_dogs").get(("mixed", 3))
```

If the record does not exist a `NotFoundError` will be raised:

```
from sqlite_utils.db import NotFoundError

try:
    row = db.table("dogs").get(5)
except NotFoundError:
    print("Dog not found")
```

1.3.12 Showing the schema

The `db.schema` property returns the full SQL schema for the database as a string:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db.schema)
CREATE TABLE "dogs" (
    "id" INTEGER PRIMARY KEY,
    "name" TEXT
);
```

1.3.13 Creating tables

The easiest way to create a new table is to insert a record into it:

```
from sqlite_utils import Database
import sqlite3

db = Database("dogs.db")
dogs = db.table("dogs")
dogs.insert({
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
})
```

This will automatically create a new table called "dogs" with the following schema:

```
CREATE TABLE dogs (
    name TEXT,
    twitter TEXT,
    age INTEGER,
    is_good_dog INTEGER
)
```

You can also specify a primary key by passing the `pk=` parameter to the `.insert()` call. This will only be obeyed if the record being inserted causes the table to be created:

```
dogs.insert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, pk="id")
```

After inserting a row like this, the `dogs.last_rowid` property will return the SQLite `rowid` assigned to the most recently inserted record.

The `dogs.last_pk` property will return the last inserted primary key value, if you specified one. This can be very useful when writing code that creates foreign keys or many-to-many relationships.

Custom column order and column types

The order of the columns in the table will be derived from the order of the keys in the dictionary, provided you are using Python 3.6 or later.

If you want to explicitly set the order of the columns you can do so using the `column_order=` parameter:

```
db.table("dogs").insert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, pk="id", column_order=("id", "twitter", "name"))
```

You don't need to pass all of the columns to the `column_order` parameter. If you only pass a subset of the columns the remaining columns will be ordered based on the key order of the dictionary.

Column types are detected based on the example data provided. Sometimes you may find you need to over-ride these detected types - to create an integer column for data that was provided as a string for example, or to ensure that a table where the first example was `None` is created as an `INTEGER` rather than a `TEXT` column. You can do this using the `columns=` parameter:

```
db.table("dogs").insert({
    "id": 1,
    "name": "Cleo",
    "age": "5",
}, pk="id", columns={"age": int, "weight": float})
```

This will create a table with the following schema:

```
CREATE TABLE "dogs" (
    "id" INTEGER PRIMARY KEY,
    "name" TEXT,
    "age" INTEGER,
    "weight" REAL
)
```

Explicitly creating a table

You can directly create a new table without inserting any data into it using the `.create()` method:

```
db.table("cats").create({
    "id": int,
    "name": str,
    "weight": float,
}, pk="id")
```

The first argument here is a dictionary specifying the columns you would like to create. Each column is paired with a Python type indicating the type of column. See *Adding columns* for full details on how these types work.

This method takes optional arguments `pk=`, `column_order=`, `foreign_keys=`, `not_null=set()` and `defaults=dict()` - explained below.

A `sqlite_utils.utils.sqlite3.OperationalError` will be raised if a table of that name already exists.

You can pass `ignore=True` to ignore that error. You can also use `if_not_exists=True` to use the SQL `CREATE TABLE IF NOT EXISTS` pattern to achieve the same effect:

```
db.table("cats").create({
    "id": int,
    "name": str,
}, pk="id", if_not_exists=True)
```

To drop and replace any existing table of that name, pass `replace=True`. This is a **dangerous operation** that will result in loss of existing data in the table.

You can also pass `transform=True` to have any existing tables *transformed* to match your new table specification. This is a **dangerous operation** as it will drop columns that are no longer listed in your call to `.create()`, so be careful when running this.

```
db.table("cats").create({
    "id": int,
    "name": str,
    "weight": float,
}, pk="id", transform=True)
```

The `transform=True` option will update the table schema if any of the following have changed:

- The specified columns or their types
- The specified primary key
- The order of the columns, defined using `column_order=`
- The `not_null=` or `defaults=` arguments

Changes to `foreign_keys=` are not currently detected and applied by `transform=True`.

You can pass `strict=True` to create a table in `STRICT` mode:

```
db.table("cats").create({
    "id": int,
    "name": str,
}, strict=True)
```

Compound primary keys

If you want to create a table with a compound primary key that spans multiple columns, you can do so by passing a tuple of column names to any of the methods that accept a `pk=` parameter. For example:

```
db.table("cats").create({
    "id": int,
    "breed": str,
    "name": str,
    "weight": float,
}, pk=("breed", "id"))
```

This also works for the `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()` methods.

Specifying foreign keys

Any operation that can create a table (`.create()`, `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()`) accepts an optional `foreign_keys=` argument which can be used to set up foreign key constraints for the table that is being created.

If you are using your database with Datasette, Datasette will detect these constraints and use them to generate hyperlinks to associated records.

The `foreign_keys` argument takes a list that indicates which foreign keys should be created. The list can take several forms. The simplest is a list of columns:

```
foreign_keys=["author_id"]
```

The library will guess which tables you wish to reference based on the column names using the rules described in *Adding foreign key constraints*.

You can also be more explicit, by passing in a list of tuples:

```
foreign_keys=[
    ("author_id", "authors", "id")
]
```

This means that the `author_id` column should be a foreign key that references the `id` column in the `authors` table.

You can leave off the third item in the tuple to have the referenced column automatically set to the primary key of that table. A full example:

```
db.table("authors").insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
], pk="id")
db.table("books").insert_all([
    {"title": "Hedgehogs of the world", "author_id": 1},
    {"title": "How to train your wolf", "author_id": 2},
], foreign_keys=[
    ("author_id", "authors")
])
```

Compound foreign keys

To create a compound (multi-column) foreign key, use tuples of column names in place of the single column names:

```
db.table("courses").create({
    "course_code": str,
    "campus_name": str,
    "dept_code": str,
}, pk="course_code", foreign_keys=[
    (("campus_name", "dept_code"), "departments", ("campus_name", "dept_code"))
])
```

This creates a table-level constraint:

```
CREATE TABLE "courses" (
   "course_code" TEXT PRIMARY KEY,
   "campus_name" TEXT,
   "dept_code" TEXT,
   FOREIGN KEY ("campus_name", "dept_code") REFERENCES "departments"("campus_name",
↪"dept_code")
)
```

As with single columns, you can leave off the tuple of other columns to reference the compound primary key of the other table:

```
foreign_keys=[
    (("campus_name", "dept_code"), "departments")
]
```

To specify `ON DELETE` or `ON UPDATE` actions, pass `ForeignKey` objects instead:

```
from sqlite_utils.db import ForeignKey

db.table("books").create({
```

(continues on next page)

(continued from previous page)

```
    "id": int,
    "author_id": int,
}, pk="id", foreign_keys=[
    ForeignKey(
        table="books", column="author_id",
        other_table="authors", other_column="id",
        on_delete="CASCADE",
    )
])
```

Foreign key actions are preserved by *table.transform()* - prior to sqlite-utils 4.0 they were silently dropped when a table was transformed.

Table configuration options

The `.insert()`, `.upsert()`, `.insert_all()` and `.upsert_all()` methods each take a number of keyword arguments, some of which influence what happens should they cause a table to be created and some of which affect the behavior of those methods.

You can set default values for these methods by accessing the table through the `db.table(...)` method (instead of using `db.table("table_name")`), like so:

```
table = db.table(
    "authors",
    pk="id",
    not_null={"name", "score"},
    column_order=("id", "name", "score", "url")
)
# Now you can call .insert() like so:
table.insert({"id": 1, "name": "Tracy", "score": 5})
```

The configuration options that can be specified in this way are `pk`, `foreign_keys`, `column_order`, `not_null`, `defaults`, `batch_size`, `hash_id`, `hash_id_columns`, `alter`, `ignore`, `replace`, `extracts`, `conversions`, `columns`, `strict`. These are all documented below.

Setting defaults and not null constraints

Each of the methods that can cause a table to be created take optional arguments `not_null=set()` and `defaults=dict()`. The methods that take these optional arguments are:

- `db.create_table(...)`
- `table.create(...)`
- `table.insert(...)`
- `table.insert_all(...)`
- `table.upsert(...)`
- `table.upsert_all(...)`

You can use `not_null=` to pass a set of column names that should have a `NOT NULL` constraint set on them when they are created.

You can use `defaults=` to pass a dictionary mapping columns to the default value that should be specified in the `CREATE TABLE` statement.

Here's an example that uses these features:

```
db.table("authors").insert_all(
    [{"id": 1, "name": "Sally", "score": 2}],
    pk="id",
    not_null={"name", "score"},
    defaults={"score": 1},
)
db.table("authors").insert({"name": "Dharma"})

list(db.table("authors").rows)
# Outputs:
# [{'id': 1, 'name': 'Sally', 'score': 2},
#  {'id': 3, 'name': 'Dharma', 'score': 1}]
print(db.table("authors").schema)
# Outputs:
# CREATE TABLE "authors" (
#     "id" INTEGER PRIMARY KEY,
#     "name" TEXT NOT NULL,
#     "score" INTEGER NOT NULL DEFAULT 1
# )
```

1.3.14 Renaming a table

The `db.rename_table(old_name, new_name)` method can be used to rename a table:

```
db.rename_table("my_table", "new_name_for_my_table")
```

This executes the following SQL:

```
ALTER TABLE [my_table] RENAME TO [new_name_for_my_table]
```

1.3.15 Duplicating tables

The `table.duplicate()` method creates a copy of the table, copying both the table schema and all of the rows in that table:

```
db.table("authors").duplicate("authors_copy")
```

The new `authors_copy` table will now contain a duplicate copy of the data from `authors`.

This method raises `sqlite_utils.db.NoTable` if the table does not exist.

1.3.16 Bulk inserts

If you have more than one record to insert, the `insert_all()` method is a much more efficient way of inserting them. Just like `insert()` it will automatically detect the columns that should be created, but it will inspect the first batch of 100 items to help decide what those column types should be.

Use it like this:

```
db.table("dogs").insert_all([{
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
```

(continues on next page)

(continued from previous page)

```
    "is_good_dog": True,
}, {
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
    "is_good_dog": True,
}], pk="id", column_order=("id", "twitter", "name"))
```

The column types used in the `CREATE TABLE` statement are automatically derived from the types of data in that first batch of rows. Any additional columns in subsequent batches will cause a `sqlite3.OperationalError` exception to be raised unless the `alter=True` argument is supplied, in which case the new columns will be created.

The function can accept an iterator or generator of rows and will commit them according to the batch size. The default batch size is 100, but you can specify a different size using the `batch_size` parameter:

```
db.table("big_table").insert_all(({
    "id": 1,
    "name": "Name {}".format(i),
} for i in range(10000)), batch_size=1000)
```

You can skip inserting any records that have a primary key that already exists using `ignore=True`. This works with both `.insert({...}, ignore=True)` and `.insert_all([...], ignore=True)`.

You can delete all the existing rows in the table before inserting the new records using `truncate=True`. This is useful if you want to replace the data in the table.

Pass `analyze=True` to run `ANALYZE` against the table after inserting the new records.

Inserting data from a list or tuple iterator

As an alternative to passing an iterator of dictionaries, you can pass an iterator of lists or tuples. The first item yielded by the iterator must be a list or tuple of string column names, and subsequent items should be lists or tuples of values:

```
db["creatures"].insert_all([
    ["name", "species"],
    ["Cleo", "dog"],
    ["Lila", "chicken"],
    ["Bants", "chicken"],
])
```

This also works with generators:

```
def creatures():
    yield "id", "name", "city"
    yield 1, "Cleo", "San Francisco"
    yield 2, "Lila", "Los Angeles"

db["creatures"].insert_all(creatures())
```

Tuples and lists are both supported.

1.3.17 Insert-replacing data

If you try to insert data using a primary key that already exists, the `.insert()` or `.insert_all()` method will raise a `sqlite3.IntegrityError` exception.

This example that catches that exception:

```
from sqlite_utils.utils import sqlite3

try:
    db.table("dogs").insert({"id": 1, "name": "Cleo"}, pk="id")
except sqlite3.IntegrityError:
    print("Record already exists with that primary key")
```

Importing from `sqlite_utils.utils.sqlite3` ensures your code continues to work even if you are using the `pysqlite3` library instead of the Python standard library `sqlite3` module.

Use the `ignore=True` parameter to ignore this error:

```
# This fails silently if a record with id=1 already exists
db.table("dogs").insert({"id": 1, "name": "Cleo"}, pk="id", ignore=True)
```

To replace any existing records that have a matching primary key, use the `replace=True` parameter to `.insert()` or `.insert_all()`:

```
db.table("dogs").insert_all([{
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 3,
    "is_good_dog": True,
}, {
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
    "is_good_dog": True,
}], pk="id", replace=True)
```

Note

Prior to sqlite-utils 2.0 the `.upsert()` and `.upsert_all()` methods worked the same way as `.insert(replace=True)` does today. See *Upserting data* for the new behaviour of those methods introduced in 2.0.

1.3.18 Updating a specific record

You can update a record by its primary key using `table.update()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db.table("dogs").get(1))
{'id': 1, 'age': 4, 'name': 'Cleo'}
>>> db.table("dogs").update(1, {"age": 5})
>>> print(db.table("dogs").get(1))
{'id': 1, 'age': 5, 'name': 'Cleo'}
```

The first argument to `update()` is the primary key. This can be a single value, or a tuple if that table has a compound primary key:

```
>>> db.table("compound_dogs").update((5, 3), {"name": "Updated"})
```

The second argument is a dictionary of columns that should be updated, along with their new values.

You can cause any missing columns to be added automatically using `alter=True`:

```
>>> db.table("dogs").update(1, {"breed": "Mutt"}, alter=True)
```

1.3.19 Deleting a specific record

You can delete a record using `table.delete()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> db.table("dogs").delete(1)
```

The `delete()` method takes the primary key of the record. This can be a tuple of values if the row has a compound primary key:

```
>>> db.table("compound_dogs").delete((5, 3))
```

1.3.20 Deleting multiple records

You can delete all records in a table that match a specific WHERE statement using `table.delete_where()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> # Delete every dog with age less than 3
>>> db.table("dogs").delete_where("age < ?", [3])
```

Calling `table.delete_where()` with no other arguments will delete every row in the table.

Pass `analyze=True` to run `ANALYZE` against the table after deleting the rows.

1.3.21 Upserting data

Upserting allows you to insert records if they do not exist and update them if they DO exist, based on matching against their primary key.

For example, given the dogs database you could upsert the record for Cleo like so:

```
db.table("dogs").upsert({
    "id": 1,
    "name": "Cleo",
    "twitter": "cleopaws",
    "age": 4,
    "is_good_dog": True,
}, pk="id", column_order=("id", "twitter", "name"))
```

If a record exists with id=1, it will be updated to match those fields. If it does not exist it will be created.

Any existing columns that are not referenced in the dictionary passed to `.upsert()` will be unchanged. If you want to replace a record entirely, use `.insert(doc, replace=True)` instead.

Note that the `pk` and `column_order` parameters here are optional if you are certain that the table has already been created. You should pass them if the table may not exist at the time the first upsert is performed.

An `upsert_all()` method is also available, which behaves like `insert_all()` but performs upserts instead.

Every record passed to `upsert()` or `upsert_all()` must include a value for each primary key column - a record without one could never match an existing row, so a `sqlite_utils.db.PrimaryKeyRequired` exception is raised instead of quietly inserting a new row.

Note

`.upsert()` and `.upsert_all()` in sqlite-utils 1.x worked like `.insert(..., replace=True)` and `.insert_all(..., replace=True)` do in 2.x. See issue #66 for details of this change.

Alternative upserts using INSERT OR IGNORE

Upserts use `INSERT INTO ... ON CONFLICT SET`. Prior to `sqlite-utils 4.0` these used a sequence of `INSERT OR IGNORE` followed by an `UPDATE`. This older method is still used for SQLite 3.23.1 and earlier. You can force the older implementation by passing `use_old_upsert=True` to the `Database()` constructor.

1.3.22 Converting data in columns

The `table.convert(...)` method can be used to apply a conversion function to the values in a column, either to update that column or to populate new columns. It is the Python library equivalent of the *sqlite-utils convert* command.

This feature works by registering a custom SQLite function that applies a Python transformation, then running a SQL query equivalent to `UPDATE table SET column = convert_value(column);`

To transform a specific column to uppercase, you would use the following:

```
db.table("dogs").convert("name", lambda value: value.upper())
```

You can pass a list of columns, in which case the transformation will be applied to each one:

```
db.table("dogs").convert(["name", "twitter"], lambda value: value.upper())
```

To save the output to of the transformation to a different column, use the `output=` parameter:

```
db.table("dogs").convert("name", lambda value: value.upper(), output="name_upper")
```

This will add the new column, if it does not already exist. You can pass `output_type=int` or some other type to control the type of the new column - otherwise it will default to text.

If you want to drop the original column after saving the results in a separate output column, pass `drop=True`.

You can create multiple new columns from a single input column by passing `multi=True` and a conversion function that returns a Python dictionary. This example creates new `upper` and `lower` columns populated from the single `title` column:

```
table.convert(
    "title", lambda v: {"upper": v.upper(), "lower": v.lower()}, multi=True
)
```

The `.convert()` method accepts optional `where=` and `where_args=` parameters which can be used to apply the conversion to a subset of rows specified by a where clause. Here's how to apply the conversion only to rows with an `id` that is higher than 20:

```
table.convert("title", lambda v: v.upper(), where="id > :id", where_args={"id": 20})
```

These behave the same as the corresponding parameters to the *.rows_where()* method, so you can use ? placeholders and a list of values instead of `:named` placeholders with a dictionary.

1.3.23 Working with lookup tables

A useful pattern when populating large tables in to break common values out into lookup tables. Consider a table of `Trees`, where each tree has a species. Ideally these species would be split out into a separate `Species` table, with each one assigned an integer primary key that can be referenced from the `Trees` table `species_id` column.

Creating lookup tables explicitly

Calling `db.table("Species").lookup({"name": "Palm"})` creates a table called `Species` (if one does not already exist) with two columns: `id` and `name`. It sets up a unique constraint on the `name` column to guarantee it will not contain duplicate rows. It then inserts a new row with the `name` set to `Palm` and returns the new integer primary key value.

If the `Species` table already exists, it will insert the new row and return the primary key. If a row with that `name` already exists, it will return the corresponding primary key value directly.

If you call `.lookup()` against an existing table without the unique constraint it will attempt to add the constraint, raising an `IntegrityError` if the constraint cannot be created.

If you pass in a dictionary with multiple values, both values will be used to insert or retrieve the corresponding ID and any unique constraint that is created will cover all of those columns, for example:

```
db.table("Trees").insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": db.table("Species").lookup({
        "common_name": "Common Juniper",
        "latin_name": "Juniperus communis"
    })
})
```

The `.lookup()` method has an optional second argument which can be used to populate other columns in the table but only if the row does not exist yet. These columns will not be included in the unique index.

To create a species record with a note on when it was first seen, you can use this:

```
db.table("Species").lookup({"name": "Palm"}, {"first_seen": "2021-03-04"})
```

The first time this is called the record will be created for `name="Palm"`. Any subsequent calls with that name will ignore the second argument, even if it includes different values.

`None` values are matched correctly: calling `.lookup()` a second time with the same values will return the primary key of the existing row even if some of those values are `None`.

`.lookup()` also accepts keyword arguments, which are passed through to the *insert() method* and can be used to influence the shape of the created table. Supported parameters are:

- `pk` - which defaults to `id`
- `foreign_keys`
- `column_order`
- `not_null`
- `defaults`
- `extracts`

- conversions
- columns
- strict

Populating lookup tables automatically during insert/upsert

A more efficient way to work with lookup tables is to define them using the `extracts=` parameter, which is accepted by `.insert()`, `.upsert()`, `.insert_all()`, `.upsert_all()` and by the `.table(...)` factory function.

`extracts=` specifies columns which should be "extracted" out into a separate lookup table during the data insertion.

It can be either a list of column names, in which case the extracted table names will match the column names exactly, or it can be a dictionary mapping column names to the desired name of the extracted table.

To extract the `species` column out to a separate `Species` table, you can do this:

```
# Using the table factory
trees = db.table("Trees", extracts={"species": "Species"})
trees.insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": "Common Juniper"
})

# If you want the table to be called 'species', you can do this:
trees = db.table("Trees", extracts=["species"])

# Using .insert() directly
db.table("Trees").insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": "Common Juniper"
}, extracts={"species": "Species"})
```

`None` values are not extracted: no record is created for them in the lookup table and the column value stays `null`.

1.3.24 Working with many-to-many relationships

`sqlite-utils` includes a shortcut for creating records using many-to-many relationships in the form of the `table.m2m(...)` method.

Here's how to create two new records and connect them via a many-to-many table in a single line of code:

```
db.table("dogs").insert({"id": 1, "name": "Cleo"}, pk="id").m2m(
    "humans", {"id": 1, "name": "Natalie"}, pk="id"
)
```

Running this example actually creates three tables: `dogs`, `humans` and a many-to-many `dogs_humans` table. It will insert a record into each of those tables.

The `.m2m()` method executes against the last record that was affected by `.insert()` or `.update()` - the record identified by the `table.last_pk` property. To execute `.m2m()` against a specific record you can first select it by passing its primary key to `.update()`:

```
db.table("dogs").update(1).m2m(
    "humans", {"id": 2, "name": "Simon"}, pk="id"
)
```

The first argument to `.m2m()` can be either the name of a table as a string or it can be the table object itself.

The second argument can be a single dictionary record or a list of dictionaries. These dictionaries will be passed to `.upsert()` against the specified table.

Here's alternative code that creates the dog record and adds two people to it:

```
db = Database(memory=True)
dogs = db.table("dogs", pk="id")
humans = db.table("humans", pk="id")
dogs.insert({"id": 1, "name": "Cleo"}).m2m(
    humans, [
        {"id": 1, "name": "Natalie"},
        {"id": 2, "name": "Simon"}
    ]
)
```

The method will attempt to find an existing many-to-many table by looking for a table that has foreign key relationships against both of the tables in the relationship.

If it cannot find such a table, it will create a new one using the names of the two tables - `dogs_humans` in this example. You can customize the name of this table using the `m2m_table=` argument to `.m2m()`.

It it finds multiple candidate tables with foreign keys to both of the specified tables it will raise a `sqlite_utils.db.NoObviousTable` exception. You can avoid this error by specifying the correct table using `m2m_table=`.

The `.m2m()` method also takes an optional `pk=` argument to specify the primary key that should be used if the table is created, and an optional `alter=True` argument to specify that any missing columns of an existing table should be added if they are needed.

Using m2m and lookup tables together

You can work with (or create) lookup tables as part of a call to `.m2m()` using the `lookup=` parameter. This accepts the same argument as `table.lookup()` does - a dictionary of values that should be used to lookup or create a row in the lookup table.

This example creates a dogs table, populates it, creates a characteristics table, populates that and sets up a many-to-many relationship between the two. It chains `.m2m()` twice to create two associated characteristics:

```
db = Database(memory=True)
dogs = db.table("dogs", pk="id")
dogs.insert({"id": 1, "name": "Cleo"}).m2m(
    "characteristics", lookup={
        "name": "Playful"
    }
).m2m(
    "characteristics", lookup={
        "name": "Opinionated"
    }
)
```

You can inspect the database to see the results like this:

```
>>> db.table_names()
['dogs', 'characteristics', 'characteristics_dogs']
>>> list(db.table("dogs").rows)
[{'id': 1, 'name': 'Cleo'}]
>>> list(db.table("characteristics").rows)
[{'id': 1, 'name': 'Playful'}, {'id': 2, 'name': 'Opinionated'}]
>>> list(db.table("characteristics_dogs").rows)
[{'characteristics_id': 1, 'dogs_id': 1}, {'characteristics_id': 2, 'dogs_id': 1}]
>>> print(db.table("characteristics_dogs").schema)
CREATE TABLE "characteristics_dogs" (
    "characteristics_id" INTEGER REFERENCES "characteristics"("id"),
    "dogs_id" INTEGER REFERENCES "dogs"("id"),
    PRIMARY KEY ("characteristics_id", "dogs_id")
)
```

1.3.25 Analyzing a column

The `table.analyze_column(column)` method is used by the *analyze-tables* CLI command.

It takes the following arguments and options:

column - required
: The name of the column to analyze

common_limit
: The number of most common values to return. Defaults to 10.

value_truncate
: If set to an integer, values longer than this will be truncated to this length. Defaults to None.

most_common
: If set to False, the `most_common` field of the returned `ColumnDetails` will be set to None. Defaults to True.

least_common
: If set to False, the `least_common` field of the returned `ColumnDetails` will be set to None. Defaults to True.

And returns a `ColumnDetails` named tuple with the following fields:

table
: The name of the table

column
: The name of the column

total_rows
: The total number of rows in the table

num_null
: The number of rows for which this column is null

num_blank
: The number of rows for which this column is blank (the empty string)

num_distinct
: The number of distinct values in this column

most_common
: The `N` most common values as a list of `(value, count)` tuples`, or `None` if the table consists entirely of distinct values

`least_common`
: The `N` least common values as a list of `(value, count)` tuples`, or `None` if the table is entirely distinct or if the number of distinct values is less than N (since they will already have been returned in `most_common`)

1.3.26 Adding columns

You can add a new column to a table using the `.add_column(col_name, col_type)` method:

```
db.table("dogs").add_column("instagram", str)
db.table("dogs").add_column("weight", float)
db.table("dogs").add_column("dob", datetime.date)
db.table("dogs").add_column("image", "BLOB")
db.table("dogs").add_column("website") # str by default
```

You can specify the `col_type` argument either using a SQLite type as a string, or by directly passing a Python type e.g. `str` or `float`.

The `col_type` is optional - if you omit it the type of `TEXT` will be used.

SQLite types you can specify are `"TEXT"`, `"INTEGER"`, `"FLOAT"`, `"REAL"` or `"BLOB"`.

If you pass a Python type, it will be mapped to SQLite types as shown here:

```
float: "REAL"
int: "INTEGER"
bool: "INTEGER"
str: "TEXT"
bytes: "BLOB"
datetime.datetime: "TEXT"
datetime.date: "TEXT"
datetime.time: "TEXT"
datetime.timedelta: "TEXT"

# If numpy is installed
np.int8: "INTEGER"
np.int16: "INTEGER"
np.int32: "INTEGER"
np.int64: "INTEGER"
np.uint8: "INTEGER"
np.uint16: "INTEGER"
np.uint32: "INTEGER"
np.uint64: "INTEGER"
np.float16: "REAL"
np.float32: "REAL"
np.float64: "REAL"
```

You can also add a column that is a foreign key reference to another table using the `fk` parameter:

```
db.table("dogs").add_column("species_id", fk="species")
```

This will automatically detect the name of the primary key on the species table and use that (and its type) for the new column.

You can explicitly specify the column you wish to reference using `fk_col`:

```
db.table("dogs").add_column("species_id", fk="species", fk_col="ref")
```

You can set a `NOT NULL DEFAULT 'x'` constraint on the new column using `not_null_default`:

```
db.table("dogs").add_column("friends_count", int, not_null_default=0)
```

1.3.27 Adding columns automatically on insert/update

You can insert or update data that includes new columns and have the table automatically altered to fit the new schema using the `alter=True` argument. This can be passed to all four of `.insert()`, `.upsert()`, `.insert_all()` and `.upsert_all()`, or it can be passed to `db.table(table_name, alter=True)` to enable it by default for all method calls against that table instance.

```
db.table("new_table").insert({"name": "Gareth"})
# This will throw an exception:
db.table("new_table").insert({"name": "Gareth", "age": 32})
# This will succeed and add a new "age" integer column:
db.table("new_table").insert({"name": "Gareth", "age": 32}, alter=True)
# You can see confirm the new column like so:
print(db.table("new_table").columns_dict)
# Outputs this:
# {'name': <class 'str'>, 'age': <class 'int'>}

# This works too:
new_table = db.table("new_table", alter=True)
new_table.insert({"name": "Gareth", "age": 32, "shoe_size": 11})
```

1.3.28 Adding foreign key constraints

The SQLite `ALTER TABLE` statement doesn't have the ability to add foreign key references to an existing column.

The `add_foreign_key()` method here is a convenient wrapper around *table.transform()*.

It's also possible to add foreign keys by directly updating the *sqlite_master* table. The sqlite-utils-fast-fks plugin implements this pattern, using code that was included with `sqlite-utils` prior to version 3.35.

Here's an example of this mechanism in action:

```
db.table("authors").insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
], pk="id")
db.table("books").insert_all([
    {"title": "Hedgehogs of the world", "author_id": 1},
    {"title": "How to train your wolf", "author_id": 2},
])
db.table("books").add_foreign_key("author_id", "authors", "id")
```

The `table.add_foreign_key(column, other_table, other_column)` method takes the name of the column, the table that is being referenced and the key column within that other table. If you omit the `other_column` argument the primary key from that table will be used automatically. If you omit the `other_table` argument the table will be guessed based on some simple rules:

- If the column is of format `author_id`, look for tables called `author` or `authors`
- If the column does not end in `_id`, try looking for a table with the exact name of the column or that name with an added `s`

This method first checks that the specified foreign key references tables and columns that exist and does not clash with an existing foreign key. It will raise a `sqlite_utils.db.AlterError` exception if these checks fail.

To ignore the case where the key already exists, use `ignore=True`:

```
db.table("books").add_foreign_key("author_id", "authors", "id", ignore=True)
```

To add a compound foreign key, pass tuples of columns:

```
db.table("courses").add_foreign_key(
    ("campus_name", "dept_code"), "departments", ("campus_name", "dept_code")
)
```

As with single columns, omitting the other columns will use the compound primary key of the other table. `other_table` must always be specified for a compound foreign key.

Use `on_delete=` and `on_update=` to specify `ON DELETE` and `ON UPDATE` actions for the foreign key:

```
db.table("books").add_foreign_key(
    "author_id", "authors", "id", on_delete="CASCADE"
)
```

This creates a foreign key with an `ON DELETE CASCADE` clause, so deleting an author will also delete their books (provided foreign key enforcement is enabled with `PRAGMA foreign_keys = ON`). Valid actions are `"SET NULL"`, `"SET DEFAULT"`, `"CASCADE"`, `"RESTRICT"` and the default `"NO ACTION"`.

Adding multiple foreign key constraints at once

You can use `db.add_foreign_keys(...)` to add multiple foreign keys in one go. This method takes a list of four-tuples, each one specifying a `table`, `column`, `other_table` and `other_column`.

Here's an example adding two foreign keys at once:

```
db.add_foreign_keys([
    ("dogs", "breed_id", "breeds", "id"),
    ("dogs", "home_town_id", "towns", "id")
])
```

This method runs the same checks as `.add_foreign_keys()` and will raise `sqlite_utils.db.AlterError` if those checks fail.

Foreign keys that already exist are silently skipped, so repeated calls are idempotent - but only if they match exactly. Requesting a foreign key that exists with different `ON DELETE`/`ON UPDATE` actions raises `AlterError`: use `table.transform()` to change the actions of an existing foreign key.

Adding indexes for all foreign keys

If you want to ensure that every foreign key column in your database has a corresponding index, you can do so like this:

```
db.index_foreign_keys()
```

Compound foreign keys get a single composite index across their columns.

1.3.29 Dropping a table or view

You can drop a table or view using the `.drop()` method:

```
db.table("my_table").drop()
# Or for a view:
db.view("my_view").drop()
# Or for either:
db["table_or_view_name"].drop()
```

Pass `ignore=True` if you want to ignore the error caused by the table or view not existing.

```
db.table("my_table").drop(ignore=True)
```

1.3.30 Transforming a table

The SQLite `ALTER TABLE` statement is limited. It can add and drop columns and rename tables, but it cannot change column types, change `NOT NULL` status or change the primary key for a table.

The `table.transform()` method can do all of these things, by implementing a multi-step pattern described in the SQLite documentation:

1. Start a transaction
2. `CREATE TABLE tablename_new_x123` with the required changes
3. Copy the old data into the new table using `INSERT INTO tablename_new_x123 SELECT * FROM tablename;`
4. `DROP TABLE tablename;`
5. `ALTER TABLE tablename_new_x123 RENAME TO tablename;`
6. Commit the transaction

The `.transform()` method takes a number of parameters, all of which are optional.

As a bonus, calling `.transform()` will reformat the schema for the table that is stored in SQLite to make it more readable. This works even if you call it without any arguments.

To keep the original table around instead of dropping it, pass the `keep_table=` option and specify the name of the table you would like it to be renamed to:

```
table.transform(types={"age": int}, keep_table="original_table")
```

This method raises a `sqlite_utils.db.TransformError` exception if the table cannot be transformed, usually because there are existing constraints or indexes that are incompatible with modifications to the columns.

Altering column types

To alter the type of a column, use the `types=` argument:

```
# Convert the 'age' column to an integer, and 'weight' to a float
table.transform(types={"age": int, "weight": float})
```

See *Adding columns* for a list of available types.

Renaming columns

The `rename=` parameter can rename columns:

```
# Rename 'age' to 'initial_age':
table.transform(rename={"age": "initial_age"})
```

Dropping columns

To drop columns, pass them in the `drop=` set:

```
# Drop the 'age' column:
table.transform(drop={"age"})
```

Changing primary keys

To change the primary key for a table, use `pk=`. This can be passed a single column for a regular primary key, or a tuple of columns to create a compound primary key. Passing `pk=None` will remove the primary key and convert the table into a `rowid` table.

```
# Make `user_id` the new primary key
table.transform(pk="user_id")
```

Changing not null status

You can change the `NOT NULL` status of columns by using `not_null=`. You can pass this a set of columns to make those columns `NOT NULL`:

```
# Make the 'age' and 'weight' columns NOT NULL
table.transform(not_null={"age", "weight"})
```

If you want to take existing `NOT NULL` columns and change them to allow null values, you can do so by passing a dictionary of true/false values instead:

```
# 'age' is NOT NULL but we want to allow NULL:
table.transform(not_null={"age": False})

# Make age allow NULL and switch weight to being NOT NULL:
table.transform(not_null={"age": False, "weight": True})
```

Altering column defaults

The `defaults=` parameter can be used to set or change the defaults for different columns:

```
# Set default age to 1:
table.transform(defaults={"age": 1})

# Now remove the default from that column:
table.transform(defaults={"age": None})
```

Changing column order

The `column_order=` parameter can be used to change the order of the columns. If you pass the names of a subset of the columns those will go first and columns you omitted will appear in their existing order after them.

```
# Change column order
table.transform(column_order=("name", "age", "id")
```

Adding foreign key constraints

You can add one or more foreign key constraints to a table using the `add_foreign_keys=` parameter:

```
db.table("places").transform(
    add_foreign_keys=(
        ("country", "country", "id"),
        ("continent", "continent", "id")
    )
)
```

This accepts the same arguments described in *specifying foreign keys* - so you can specify them as a full tuple of `(column, other_table, other_column)`, or you can take a shortcut and pass just the name of the column, provided the table can be automatically derived from the column name:

```
db.table("places").transform(
    add_foreign_keys=(("country", "continent"))
)
```

Replacing foreign key constraints

The `foreign_keys=` parameter is similar to to `add_foreign_keys=` but can be be used to replace all foreign key constraints on a table, dropping any that are not explicitly mentioned:

```
db.table("places").transform(
    foreign_keys=(
        ("continent", "continent", "id"),
    )
)
```

Dropping foreign key constraints

You can use `.transform()` to remove foreign key constraints from a table.

This example drops two foreign keys - the one from `places.country` to `country.id` and the one from `places.continent` to `continent.id`:

```
db.table("places").transform(
    drop_foreign_keys=("country", "continent")
)
```

A bare column name drops any foreign key that column participates in, including compound foreign keys. To target a compound foreign key precisely, pass a tuple of its columns:

```
db.table("courses").transform(
    drop_foreign_keys=[("campus_name", "dept_code")]
)
```

Renaming a column with `rename=` updates any foreign keys that use it, and dropping a column with `drop=` also drops any foreign keys it participates in - for a compound foreign key this removes the whole constraint.

Custom transformations with .transform_sql()

The `.transform()` method can handle most cases, but it does not automatically upgrade indexes, views or triggers associated with the table that is being transformed.

If you want to do something more advanced, you can call the `table.transform_sql(...)` method with the same arguments that you would have passed to `table.transform(...)`.

This method will return a list of SQL statements that should be executed to implement the change. You can then make modifications to that SQL - or add additional SQL statements - before executing it yourself.

1.3.31 Extracting columns into a separate table

The `table.extract()` method can be used to extract specified columns into a separate table.

Imagine a `Trees` table that looks like this:

id	TreeAddress	Species
1	52 Vine St	Palm
2	12 Draft St	Oak
3	51 Dark Ave	Palm
4	1252 Left St	Palm

The `Species` column contains duplicate values. This database could be improved by extracting that column out into a separate `Species` table and pointing to it using a foreign key column.

The schema of the above table is:

```
CREATE TABLE "Trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "Species" TEXT
)
```

Here's how to extract the `Species` column using `.extract()`:

```
db.table("Trees").extract("Species")
```

After running this code the table schema now looks like this:

```
CREATE TABLE "Trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "Species_id" INTEGER,
    FOREIGN KEY(Species_id) REFERENCES Species(id)
)
```

A new `Species` table will have been created with the following schema:

```
CREATE TABLE "Species" (
    "id" INTEGER PRIMARY KEY,
    "Species" TEXT
)
```

The `.extract()` method defaults to creating a table with the same name as the column that was extracted, and adding a foreign key column called `tablename_id`.

You can specify a custom table name using `table=`, and a custom foreign key name using `fk_column=`. This example creates a table called `tree_species` and a foreign key column called `tree_species_id`:

```
db.table("Trees").extract("Species", table="tree_species", fk_column="tree_species_id")
```

The resulting schema looks like this:

```
CREATE TABLE "Trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "tree_species_id" INTEGER,
    FOREIGN KEY(tree_species_id) REFERENCES tree_species(id)
)

CREATE TABLE "tree_species" (
    "id" INTEGER PRIMARY KEY,
    "Species" TEXT
)
```

You can also extract multiple columns into the same external table. Say for example you have a table like this:

id	TreeAddress	CommonName	LatinName
1	52 Vine St	Palm	Arecaceae
2	12 Draft St	Oak	Quercus
3	51 Dark Ave	Palm	Arecaceae
4	1252 Left St	Palm	Arecaceae

You can pass `["CommonName", "LatinName"]` to `.extract()` to extract both of those columns:

```
db.table("Trees").extract(["CommonName", "LatinName"])
```

This produces the following schema:

```
CREATE TABLE "Trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "CommonName_LatinName_id" INTEGER,
    FOREIGN KEY(CommonName_LatinName_id) REFERENCES CommonName_LatinName(id)
)
CREATE TABLE "CommonName_LatinName" (
    "id" INTEGER PRIMARY KEY,
    "CommonName" TEXT,
    "LatinName" TEXT
)
```

The table name `CommonName_LatinName` is derived from the extract columns. You can use `table=` and `fk_column=` to specify custom names like this:

```
db.table("Trees").extract(["CommonName", "LatinName"], table="Species", fk_column=
↪"species_id")
```

This produces the following schema:

```
CREATE TABLE "Trees" (
    "id" INTEGER PRIMARY KEY,
    "TreeAddress" TEXT,
    "species_id" INTEGER,
    FOREIGN KEY(species_id) REFERENCES Species(id)
)
CREATE TABLE "Species" (
    "id" INTEGER PRIMARY KEY,
    "CommonName" TEXT,
    "LatinName" TEXT
)
```

You can use the `rename=` argument to rename columns in the lookup table. To create a `Species` table with columns called `name` and `latin` you can do this:

```
db.table("Trees").extract(
    ["CommonName", "LatinName"],
    table="Species",
    fk_column="species_id",
    rename={"CommonName": "name", "LatinName": "latin"}
)
```

This produces a lookup table like so:

```
CREATE TABLE "Species" (
    "id" INTEGER PRIMARY KEY,
    "name" TEXT,
    "latin" TEXT
)
```

Rows where every extracted column is `null` are not extracted: no record is created for them in the lookup table and their foreign key column is left as `null`. When extracting multiple columns, rows where at least one of the extracted columns has a value will be extracted as usual.

1.3.32 Setting an ID based on the hash of the row contents

Sometimes you will find yourself working with a dataset that includes rows that do not have a provided obvious ID, but where you would like to assign one so that you can later upsert into that table without creating duplicate records.

In these cases, a useful technique is to create an ID that is derived from the sha1 hash of the row contents.

`sqlite-utils` can do this for you using the `hash_id=` option. For example:

```
db = sqlite_utils.Database("dogs.db")
db.table("dogs").upsert({"name": "Cleo", "twitter": "cleopaws"}, hash_id="id")
print(list(db["dogs"]))
```

Outputs:

```
[{'id': 'f501265970505d9825d8d9f590bfab3519fb20b1', 'name': 'Cleo', 'twitter': 'cleopaws
↪'}]
```

If you are going to use that ID straight away, you can access it using `last_pk`:

```
dog_id = db.table("dogs").upsert({
    "name": "Cleo",
    "twitter": "cleopaws"
}, hash_id="id").last_pk
# dog_id is now "f501265970505d9825d8d9f590bfab3519fb20b1"
```

The hash will be created using all of the column values. To create a hash using a subset of the columns, pass the `hash_id_columns=` parameter:

```
db.table("dogs").upsert(
    {"name": "Cleo", "twitter": "cleopaws", "age": 7},
    hash_id_columns=("name", "twitter")
)
```

The `hash_id=` parameter is optional if you specify `hash_id_columns=` - it will default to putting the hash in a column called `id`.

You can manually calculate these hashes using the *hash_record(record, keys=...)* utility function.

1.3.33 Creating views

The `.create_view()` method on the database class can be used to create a view:

```
db.create_view("good_dogs", """
    select * from dogs where is_good_dog = 1
""")
```

This will raise a `sqlite_utils.utils.OperationalError` if a view with that name already exists.

You can pass `ignore=True` to silently ignore an existing view and do nothing, or `replace=True` to replace an existing view with a new definition if your select statement differs from the current view:

```
db.create_view("good_dogs", """
    select * from dogs where is_good_dog = 1
""", replace=True)
```

1.3.34 Storing JSON

SQLite has excellent JSON support, and `sqlite-utils` can help you take advantage of this: if you attempt to insert a value that can be represented as a JSON list or dictionary, `sqlite-utils` will create TEXT column and store your data as serialized JSON. This means you can quickly store even complex data structures in SQLite and query them using JSON features.

For example:

```
db.table("niche_museums").insert({
    "name": "The Bigfoot Discovery Museum",
    "url": "http://bigfootdiscoveryproject.com/"
    "hours": {
        "Monday": [11, 18],
        "Wednesday": [11, 18],
        "Thursday": [11, 18],
        "Friday": [11, 18],
        "Saturday": [11, 18],
        "Sunday": [11, 18]
```

(continues on next page)

(continued from previous page)

```
    },
    "address": {
        "streetAddress": "5497 Highway 9",
        "addressLocality": "Felton, CA",
        "postalCode": "95018"
    }
})
db.execute("""
    select json_extract(address, '$.addressLocality')
    from niche_museums
""").fetchall()
# Returns [('Felton, CA',)]
```

1.3.35 Converting column values using SQL functions

Sometimes it can be useful to run values through a SQL function prior to inserting them. A simple example might be converting a value to upper case while it is being inserted.

The `conversions={...}` parameter can be used to specify custom SQL to be used as part of a `INSERT` or `UPDATE` SQL statement.

You can specify an upper case conversion for a specific column like so:

```
db.table("example").insert({
    "name": "The Bigfoot Discovery Museum"
}, conversions={"name": "upper(?)"})

# list(db.table("example").rows) now returns:
# [{'name': 'THE BIGFOOT DISCOVERY MUSEUM'}]
```

The dictionary key is the column name to be converted. The value is the SQL fragment to use, with a `?` placeholder for the original value.

A more useful example: if you are working with SpatiaLite you may find yourself wanting to create geometry values from a WKT value. Code to do that could look like this:

```
import sqlite3
import sqlite_utils
from shapely.geometry import shape
import httpx

db = sqlite_utils.Database("places.db")
# Initialize SpatiaLite
db.init_spatialite()
# Use sqlite-utils to create a places table
places = db.table("places").create({"id": int, "name": str})

# Add a SpatiaLite 'geometry' column
places.add_geometry_column("geometry", "MULTIPOLYGON")

# Fetch some GeoJSON from Who's On First:
geojson = httpx.get(
    "https://raw.githubusercontent.com/whosonfirst-data/"
    "whosonfirst-data-admin-gb/master/data/404/227/475/404227475.geojson"
```

(continues on next page)

(continued from previous page)

```
).json()

# Convert to "Well Known Text" format using shapely
wkt = shape(geojson["geometry"]).wkt

# Insert the record, converting the WKT to a SpatiaLite geometry:
db.table("places").insert(
    {"name": "Wales", "geometry": wkt},
    conversions={"geometry": "GeomFromText(?, 4326)"},
)
```

This example uses gographical data from Who's On First and depends on the Shapely and HTTPX Python libraries.

1.3.36 Checking the SQLite version

The `db.sqlite_version` property returns a tuple of integers representing the version of SQLite used for that database object:

```
>>> db.sqlite_version
(3, 36, 0)
```

1.3.37 Dumping the database to SQL

The `db.iterdump()` method returns a sequence of SQL strings representing a complete dump of the database. Use it like this:

```
full_sql = "".join(db.iterdump())
```

This uses the sqlite3.Connection.iterdump() method.

If you are using `pysqlite3` the underlying method may be missing. If you install the sqlite-dump package then the `db.iterdump()` method will use that implementation instead:

```
pip install sqlite-dump
```

1.3.38 Introspecting tables and views

If you have loaded an existing table or view, you can use introspection to find out more about it:

```
>>> db.table("PlantType")
<Table PlantType (id, value)>
### db.view("NameOfView")
<View NameOfView (id, value)>
```

.exists()

The `.exists()` method can be used to find out if a table exists or not:

```
>>> db.table("PlantType").exists()
True
>>> db.table("PlantType2").exists()
False
```

(continues on next page)

(continued from previous page)

```
>>> db["table_or_view_name"].exists()
False
```

.count

The `.count` property shows the current number of rows (`select count(*) from table`):

```
>>> db.table("PlantType").count
3
>>> db.table("Street_Tree_List").count
189144
```

This property will take advantage of *Cached table counts using triggers* if the `use_counts_table` property is set on the database. You can avoid that optimization entirely by calling `table.count_where()` instead of accessing the property.

.columns

The `.columns` property shows the columns in the table or view. It returns a list of `Column(cid, name, type, notnull, default_value, is_pk)` named tuples.

```
>>> db.table("PlantType").columns
[Column(cid=0, name='id', type='INTEGER', notnull=0, default_value=None, is_pk=1),
 Column(cid=1, name='value', type='TEXT', notnull=0, default_value=None, is_pk=0)]
```

.columns_dict

The `.columns_dict` property returns a dictionary version of the columns with just the names and Python types:

```
>>> db.table("PlantType").columns_dict
{'id': <class 'int'>, 'value': <class 'str'>}
```

.default_values

The `.default_values` property returns a dictionary of default values for each column that has a default:

```
>>> db.table("table_with_defaults").default_values
{'score': 5}
```

.pks

The `.pks` property returns a list of strings naming the primary key columns for the table:

```
>>> db.table("PlantType").pks
['id']
```

If a table has no primary keys but is a rowid table, this property will return `['rowid']`.

.use_rowid

Almost all SQLite tables have a `rowid` column, but a table with no explicitly defined primary keys must use that `rowid` as the primary key for identifying individual rows. The `.use_rowid` property checks to see if a table needs to use the `rowid` in this way - it returns `True` if the table has no explicitly defined primary keys and `False` otherwise.

```
>>> db.table("PlantType").use_rowid
False
```

.foreign_keys

The `.foreign_keys` property returns any foreign key relationships for the table, as a list of `ForeignKey` objects. It is not available on views.

Each `ForeignKey` has the following attributes:

table
: The table the foreign key is defined on.

column
: The column on this table, or `None` for a compound foreign key.

other_table
: The table being referenced.

other_column
: The referenced column, or `None` for a compound foreign key.

columns
: A tuple of the columns on this table, always populated (a one-item tuple for single-column foreign keys).

other_columns
: A tuple of the referenced columns.

is_compound
: `True` if this is a compound (multi-column) foreign key.

on_delete
: The `ON DELETE` action, e.g. `"CASCADE"` - `"NO ACTION"` if not set.

on_update
: The `ON UPDATE` action - `"NO ACTION"` if not set.

`ForeignKey` was a `namedtuple` prior to sqlite-utils 4.0. It is now a dataclass and can no longer be unpacked or indexed as a tuple - access its fields by name instead. See *Upgrading from 3.x to 4.0* for details.

```
>>> db.table("Street_Tree_List").foreign_keys
[ForeignKey(table='Street_Tree_List', column='qLegalStatus', other_table='qLegalStatus',␣
↪other_column='id', columns=('qLegalStatus',), other_columns=('id',), is_compound=False,
↪ on_delete='NO ACTION', on_update='NO ACTION'),
 ForeignKey(table='Street_Tree_List', column='qCareAssistant', other_table=
↪'qCareAssistant', other_column='id', columns=('qCareAssistant',), other_columns=('id',
↪), is_compound=False, on_delete='NO ACTION', on_update='NO ACTION'),
 ...]
```

Compound foreign keys - defined with `FOREIGN KEY (col_a, col_b) REFERENCES other(col_a, col_b)` - are returned as a single `ForeignKey` with `is_compound=True`, `column` and `other_column` set to `None`, and the participating columns available in the `columns` and `other_columns` tuples.

.schema

The `.schema` property outputs the table's schema as a SQL string:

```
>>> print(db.table("Street_Tree_List").schema)
CREATE TABLE "Street_Tree_List" (
"TreeID" INTEGER,
  "qLegalStatus" INTEGER,
  "qSpecies" INTEGER,
  "qAddress" TEXT,
  "SiteOrder" INTEGER,
  "qSiteInfo" INTEGER,
  "PlantType" INTEGER,
  "qCaretaker" INTEGER,
  "qCareAssistant" INTEGER,
  "PlantDate" TEXT,
  "DBH" INTEGER,
  "PlotSize" TEXT,
  "PermitNotes" TEXT,
  "XCoord" REAL,
  "YCoord" REAL,
  "Latitude" REAL,
  "Longitude" REAL,
  "Location" TEXT
,
FOREIGN KEY ("PlantType") REFERENCES "PlantType"(id),
    FOREIGN KEY ("qCaretaker") REFERENCES "qCaretaker"(id),
    FOREIGN KEY ("qSpecies") REFERENCES "qSpecies"(id),
    FOREIGN KEY ("qSiteInfo") REFERENCES "qSiteInfo"(id),
    FOREIGN KEY ("qCareAssistant") REFERENCES "qCareAssistant"(id),
    FOREIGN KEY ("qLegalStatus") REFERENCES "qLegalStatus"(id))
```

.strict

The `.strict` property identifies if the table is a SQLite STRICT table.

```
>>> db.table("ny_times_us_counties").strict
False
```

.indexes

The `.indexes` property returns all indexes created for a table, as a list of `Index(seq, name, unique, origin, partial, columns)` named tuples. It is not available on views.

```
>>> db.table("Street_Tree_List").indexes
[Index(seq=0, name='"Street_Tree_List_qLegalStatus"', unique=0, origin='c', partial=0,␣
↪columns=['qLegalStatus']),
 Index(seq=1, name='"Street_Tree_List_qCareAssistant"', unique=0, origin='c', partial=0,␣
↪columns=['qCareAssistant']),
 Index(seq=2, name='"Street_Tree_List_qSiteInfo"', unique=0, origin='c', partial=0,␣
↪columns=['qSiteInfo']),
 Index(seq=3, name='"Street_Tree_List_qSpecies"', unique=0, origin='c', partial=0,␣
↪columns=['qSpecies']),
 Index(seq=4, name='"Street_Tree_List_qCaretaker"', unique=0, origin='c', partial=0,␣
↪columns=['qCaretaker']),
 Index(seq=5, name='"Street_Tree_List_PlantType"', unique=0, origin='c', partial=0,␣
↪columns=['PlantType'])]
```

.xindexes

The `.xindexes` property returns more detailed information about the indexes on the table, using the SQLite PRAGMA index_xinfo() mechanism. It returns a list of `XIndex(name, columns)` named tuples, where `columns` is a list of `XIndexColumn(seqno, cid, name, desc, coll, key)` named tuples.

```
>>> db.table("ny_times_us_counties").xindexes
[
    XIndex(
        name='idx_ny_times_us_counties_date',
        columns=[
            XIndexColumn(seqno=0, cid=0, name='date', desc=1, coll='BINARY', key=1),
            XIndexColumn(seqno=1, cid=-1, name=None, desc=0, coll='BINARY', key=0)
        ]
    ),
    XIndex(
        name='idx_ny_times_us_counties_fips',
        columns=[
            XIndexColumn(seqno=0, cid=3, name='fips', desc=0, coll='BINARY', key=1),
            XIndexColumn(seqno=1, cid=-1, name=None, desc=0, coll='BINARY', key=0)
        ]
    )
]
```

.triggers

The `.triggers` property lists database triggers. It can be used on both database and table objects. It returns a list of `Trigger(name, table, sql)` named tuples.

```
>>> db.table("authors").triggers
[Trigger(name='authors_ai', table='authors', sql='CREATE TRIGGER [authors_ai] AFTER␣
↪INSERT...'),
 Trigger(name='authors_ad', table='authors', sql="CREATE TRIGGER [authors_ad] AFTER␣
↪DELETE..."),
 Trigger(name='authors_au', table='authors', sql="CREATE TRIGGER [authors_au] AFTER␣
↪UPDATE")]
>>> db.triggers
... similar output to db.table("authors").triggers
```

.triggers_dict

The `.triggers_dict` property returns the triggers for that table as a dictionary mapping their names to their SQL definitions.

```
>>> db.table("authors").triggers_dict
{'authors_ai': 'CREATE TRIGGER [authors_ai] AFTER INSERT...',
 'authors_ad': 'CREATE TRIGGER [authors_ad] AFTER DELETE...',
 'authors_au': 'CREATE TRIGGER [authors_au] AFTER UPDATE'}
```

The same property exists on the database, and will return all triggers across all tables:

```
>>> db.triggers_dict
{'authors_ai': 'CREATE TRIGGER [authors_ai] AFTER INSERT...',
 'authors_ad': 'CREATE TRIGGER [authors_ad] AFTER DELETE...',
 'authors_au': 'CREATE TRIGGER [authors_au] AFTER UPDATE'}
```

.detect_fts()

The `detect_fts()` method returns the associated SQLite FTS table name, if one exists for this table. If the table has not been configured for full-text search it returns `None`.

```
>>> db.table("authors").detect_fts()
"authors_fts"
```

.virtual_table_using

The `.virtual_table_using` property reveals if a table is a virtual table. It returns `None` for regular tables and the upper case version of the type of virtual table otherwise. For example:

```
>>> db.table("authors").enable_fts(["name"])
>>> db.table("authors_fts").virtual_table_using
"FTS5"
```

.has_counts_triggers

The `.has_counts_triggers` property shows if a table has been configured with triggers for updating a `_counts` table, as described in *Cached table counts using triggers*.

```
>>> db.table("authors").has_counts_triggers
False
>>> db.table("authors").enable_counts()
>>> db.table("authors").has_counts_triggers
True
```

db.supports_strict

This property on the database object returns `True` if the available SQLite version supports STRICT mode, which was added in SQLite 3.37.0 (on 2021-11-27).

```
>>> db.supports_strict
True
```

1.3.39 Full-text search

SQLite includes bundled extensions that implement powerful full-text search.

Enabling full-text search for a table

You can enable full-text search on a table using `.enable_fts(columns)`:

```
db.table("dogs").enable_fts(["name", "twitter"])
```

You can then run searches using the `.search()` method:

```
rows = list(db.table("dogs").search("cleo"))
```

This method returns a generator that can be looped over to get dictionaries for each row, similar to *Listing rows*.

If you insert additional records into the table you will need to refresh the search index using `populate_fts()`:

```
db.table("dogs").insert({
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
    "is_good_dog": True,
}, pk="id")
db.table("dogs").populate_fts(["name", "twitter"])
```

A better solution is to use database triggers. You can set up database triggers to automatically update the full-text index using `create_triggers=True`:

```
db.table("dogs").enable_fts(["name", "twitter"], create_triggers=True)
```

`.enable_fts()` defaults to using FTS5. If you wish to use FTS4 instead, use the following:

```
db.table("dogs").enable_fts(["name", "twitter"], fts_version="FTS4")
```

You can customize the tokenizer configured for the table using the `tokenize=` parameter. For example, to enable Porter stemming, where English words like "running" will match stemmed alternatives such as "run", use `tokenize="porter"`:

```
db.table("articles").enable_fts(["headline", "body"], tokenize="porter")
```

The SQLite documentation has more on FTS5 tokenizers and FTS4 tokenizers. `porter` is a valid option for both.

If you attempt to configure a FTS table where one already exists, a `sqlite3.OperationalError` exception will be raised.

You can replace the existing table with a new configuration using `replace=True`:

```
db.table("articles").enable_fts(["headline"], tokenize="porter", replace=True)
```

This will have no effect if the FTS table already exists, otherwise it will drop and recreate the table with the new settings. This takes into consideration the columns, the tokenizer, the FTS version used and whether or not the table has triggers.

To remove the FTS tables and triggers you created, use the `disable_fts()` table method:

```
db.table("dogs").disable_fts()
```

Quoting characters for use in search

SQLite supports advanced search query syntax. In some situations you may wish to disable this, since characters such as `.` may have special meaning that causes errors when searching for strings provided by your users.

The `db.quote_fts(query)` method returns the query with SQLite full-text search quoting applied such that the query should be safe to use in a search:

```
db.quote_fts("Search term.")
# Returns: '"Search" "term."'
```

Searching with table.search()

The `table.search(q)` method returns a generator over Python dictionaries representing rows that match the search phrase `q`, ordered by relevance with the most relevant results first.

```
for article in db.table("articles").search("jquery"):
    print(article)
```

The `.search()` method also accepts the following optional parameters:

order_by string
: The column to sort by. Defaults to relevance score. Can optionally include a `desc`, e.g. `rowid desc`.

columns array of strings
: Columns to return. Defaults to all columns.

limit integer
: Number of results to return. Defaults to all results.

offset integer
: Offset to use along side the limit parameter.

where string
: Extra SQL fragment for the WHERE clause

where_args dictionary
: Arguments to use for `:param` placeholders in the extra WHERE clause

include_rank bool
: If set a `rank` column will be included with the BM25 ranking score - for FTS5 tables only.

quote bool
: Apply *FTS quoting rules* to the search query, disabling advanced query syntax in a way that avoids surprising errors.

To return just the title and published columns for three matches for `"dog"` where the `id` is greater than 10 ordered by `published` with the most recent first, use the following:

```
for article in db.table("articles").search(
    "dog",
    order_by="published desc",
    limit=3,
    where="id > :min_id",
    where_args={"min_id": 10},
    columns=["title", "published"]
):
    print(article)
```

Building SQL queries with table.search_sql()

You can generate the SQL query that would be used for a search using the `table.search_sql()` method. It takes the same arguments as `table.search()`, with the exception of the search query and the `where_args` parameter, since those should be provided when the returned SQL is executed.

```
print(db.table("articles").search_sql(columns=["title", "author"]))
```

Outputs:

```
with original as (
    select
        rowid,
        [title],
```

(continues on next page)

(continued from previous page)

```
        [author]
    from [articles]
)
select
    [original].[title],
    [original].[author]
from
    [original]
    join [articles_fts] on [original].rowid = [articles_fts].rowid
where
    [articles_fts] match :query
order by
    [articles_fts].rank
```

This method detects if a SQLite table uses FTS4 or FTS5, and outputs the correct SQL for ordering by relevance depending on the search type.

The FTS4 output looks something like this:

```
with original as (
    select
        rowid,
        [title],
        [author]
    from [articles]
)
select
    [original].[title],
    [original].[author]
from
    [original]
    join [articles_fts] on [original].rowid = [articles_fts].rowid
where
    [articles_fts] match :query
order by
    rank_bm25(matchinfo([articles_fts], 'pcnalx'))
```

This uses the `rank_bm25()` custom SQL function from sqlite-fts4. You can register that custom function against a `Database` connection using this method:

```
db.register_fts4_bm25()
```

1.3.40 Rebuilding a full-text search table

You can rebuild a table using the `table.rebuild_fts()` method. This is useful for if the table configuration changes or the indexed data has become corrupted in some way.

```
db.table("dogs").rebuild_fts()
```

This method can be called on a table that has been configured for full-text search - `dogs` in this instance - or directly on a `_fts` table:

```
db.table("dogs_fts").rebuild_fts()
```

This runs the following SQL:

```
INSERT INTO dogs_fts (dogs_fts) VALUES ("rebuild");
```

1.3.41 Optimizing a full-text search table

Once you have populated a FTS table you can optimize it to dramatically reduce its size like so:

```
db.table("dogs").optimize()
```

This runs the following SQL:

```
INSERT INTO dogs_fts (dogs_fts) VALUES ("optimize");
```

1.3.42 Cached table counts using triggers

The `select count(*)` query in SQLite requires a full scan of the primary key index, and can take an increasingly long time as the table grows larger.

The `table.enable_counts()` method can be used to configure triggers to continuously update a record in a `_counts` table. This value can then be used to quickly retrieve the count of rows in the associated table.

```
db.table("dogs").enable_counts()
```

This will create the `_counts` table if it does not already exist, with the following schema:

```
CREATE TABLE "_counts" (
   "table" TEXT PRIMARY KEY,
   "count" INTEGER DEFAULT 0
)
```

You can enable cached counts for every table in a database (except for virtual tables and the `_counts` table itself) using the database `enable_counts()` method:

```
db.enable_counts()
```

Once enabled, table counts will be stored in the `_counts` table. The count records will be automatically kept up-to-date by the triggers when rows are added or deleted to the table.

To access these counts you can query the `_counts` table directly or you can use the `db.cached_counts()` method. This method returns a dictionary mapping tables to their counts:

```
>>> db.cached_counts()
{'global-power-plants': 33643,
 'global-power-plants_fts_data': 136,
 'global-power-plants_fts_idx': 199,
 'global-power-plants_fts_docsize': 33643,
 'global-power-plants_fts_config': 1}
```

You can pass a list of table names to this method to retrieve just those counts:

```
>>> db.cached_counts(["global-power-plants"])
{'global-power-plants': 33643}
```

The `table.count` property executes a `select count(*)` query by default, unless the `db.use_counts_table` property is set to `True`.

You can set `use_counts_table` to `True` when you instantiate the database object:

```
db = Database("global-power-plants.db", use_counts_table=True)
```

If the property is `True` any calls to the `table.count` property will first attempt to find the cached count in the `_counts` table, and fall back on a `count(*)` query if the value is not available or the table is missing.

Calling the `.enable_counts()` method on a database or table object will set `use_counts_table` to `True` for the lifetime of that database object.

If the `_counts` table ever becomes out-of-sync with the actual table counts you can repair it using the `.reset_counts()` method:

```
db.reset_counts()
```

1.3.43 Creating indexes

You can create an index on a table using the `.create_index(columns)` method. The method takes a list of columns:

```
db.table("dogs").create_index(["is_good_dog"])
```

By default the index will be named `idx_{table-name}_{columns}`. If you pass `find_unique_name=True` and the automatically derived name already exists, an available name will be found by incrementing a suffix number, for example `idx_items_title_2`.

You can customize the name of the created index by passing the `index_name` parameter:

```
db.table("dogs").create_index(
    ["is_good_dog", "age"],
    index_name="good_dogs_by_age"
)
```

To create an index in descending order for a column, wrap the column name in `db.DescIndex()` like this:

```
from sqlite_utils.db import DescIndex

db.table("dogs").create_index(
    ["is_good_dog", DescIndex("age")],
    index_name="good_dogs_by_age"
)
```

You can create a unique index by passing `unique=True`:

```
db.table("dogs").create_index(["name"], unique=True)
```

Use `if_not_exists=True` to do nothing if an index with that name already exists.

Pass `analyze=True` to run `ANALYZE` against the new index after creating it.

1.3.44 Optimizing index usage with ANALYZE

The SQLite ANALYZE command builds a table of statistics which the query planner can use to make better decisions about which indexes to use for a given query.

You should run `ANALYZE` if your database is large and you do not think your indexes are being efficiently used.

To run `ANALYZE` against every index in a database, use this:

```
db.analyze()
```

To run it just against a specific named index, pass the name of the index to that method:

```
db.analyze("idx_countries_country_name")
```

To run against all indexes attached to a specific table, you can either pass the table name to `db.analyze(...)` or you can call the method directly on the table, like this:

```
db.table("dogs").analyze()
```

1.3.45 Vacuum

You can optimize your database by running VACUUM against it like so:

```
Database("my_database.db").vacuum()
```

1.3.46 WAL mode

You can enable Write-Ahead Logging for a database with `.enable_wal()`:

```
Database("my_database.db").enable_wal()
```

You can disable WAL mode using `.disable_wal()`:

```
Database("my_database.db").disable_wal()
```

The journal mode can only be changed outside of a transaction. Calling either method while a transaction is open - inside a `db.atomic()` block, for example - raises a `sqlite_utils.db.TransactionError`, unless the database is already in the requested mode in which case the call is a no-op.

You can check the current journal mode for a database using the `journal_mode` property:

```
journal_mode = Database("my_database.db").journal_mode
```

This will usually be `wal` or `delete` (meaning WAL is disabled), but can have other values - see the PRAGMA journal_mode documentation.

1.3.47 Suggesting column types

When you create a new table for a list of inserted or upserted Python dictionaries, those methods detect the correct types for the database columns based on the data you pass in.

In some situations you may need to intervene in this process, to customize the columns that are being created in some way - see *Explicitly creating a table*.

That table `.create()` method takes a dictionary mapping column names to the Python type they should store:

```
db.table("cats").create({
    "id": int,
    "name": str,
    "weight": float,
})
```

You can use the `suggest_column_types()` helper function to derive a dictionary of column names and types from a list of records, suitable to be passed to `table.create()`.

For example:

```
from sqlite_utils import Database, suggest_column_types

cats = [{
    "id": 1,
    "name": "Snowflake"
}, {
    "id": 2,
    "name": "Crabtree",
    "age": 4
}]
types = suggest_column_types(cats)
# types now looks like this:
# {"id": <class 'int'>,
#  "name": <class 'str'>,
#  "age": <class 'int'>}

# Manually add an extra field:
types["thumbnail"] = bytes
# types now looks like this:
# {"id": <class 'int'>,
#  "name": <class 'str'>,
#  "age": <class 'int'>,
#  "thumbnail": <class 'bytes'>}

# Create the table
db = Database("cats.db")
db.table("cats").create(types, pk="id")
# Insert the records
db.table("cats").insert_all(cats)

# list(db.table("cats").rows) now returns:
# [{"id": 1, "name": "Snowflake", "age": None, "thumbnail": None}
#  {"id": 2, "name": "Crabtree", "age": 4, "thumbnail": None}]

# The table schema looks like this:
# print(db.table("cats").schema)
# CREATE TABLE "cats" (
#    "id" INTEGER PRIMARY KEY,
#    "name" TEXT,
#    "age" INTEGER,
#    "thumbnail" BLOB
# )
```

1.3.48 Registering custom SQL functions

SQLite supports registering custom SQL functions written in Python. The `db.register_function()` method lets you register these functions, and keeps track of functions that have already been registered.

If you use it as a method it will automatically detect the name and number of arguments needed by the function:

```
from sqlite_utils import Database

db = Database(memory=True)

def reverse_string(s):
    return "".join(reversed(list(s)))

db.register_function(reverse_string)
print(db.execute('select reverse_string("hello")').fetchone()[0])
# This prints "olleh"
```

You can also use the method as a function decorator like so:

```
@db.register_function
def reverse_string(s):
    return "".join(reversed(list(s)))

print(db.execute('select reverse_string("hello")').fetchone()[0])
```

By default, the name of the Python function will be used as the name of the SQL function. You can customize this with the `name=` keyword argument:

```
@db.register_function(name="rev")
def reverse_string(s):
    return "".join(reversed(list(s)))

print(db.execute('select rev("hello")').fetchone()[0])
```

If a function will return the exact same result for any given inputs you can register it as a deterministic SQLite function allowing SQLite to apply some performance optimizations:

```
@db.register_function(deterministic=True)
def reverse_string(s):
    return "".join(reversed(list(s)))
```

By default registering a function with the same name and number of arguments will have no effect - the `Database` instance keeps track of functions that have already been registered and skips registering them if `@db.register_function` is called a second time.

If you want to deliberately replace the registered function with a new implementation, use the `replace=True` argument:

```
@db.register_function(deterministic=True, replace=True)
def reverse_string(s):
    return s[::-1]
```

Exceptions that occur inside a user-defined function default to returning the following error:

```
Unexpected error: user-defined function raised exception
```

You can cause `sqlite3` to return more useful errors, including the traceback from the custom function, by executing the following before your custom functions are executed:

```
from sqlite_utils.utils import sqlite3

sqlite3.enable_callback_tracebacks(True)
```

1.3.49 Quoting strings for use in SQL

In almost all cases you should pass values to your SQL queries using the optional `parameters` argument to `db.query()`, as described in *Passing parameters*.

If that option isn't relevant to your use-case you can to quote a string for use with SQLite using the `db.quote()` method, like so:

```
>>> db = Database(memory=True)
>>> db.quote("hello")
"'hello'"
>>> db.quote("hello'this'has'quotes")
"'hello''this''has''quotes'"
```

1.3.50 Reading rows from a file

The `sqlite_utils.utils.rows_from_file()` helper function can read rows (a sequence of dictionaries) from CSV, TSV, JSON or newline-delimited JSON files.

`sqlite_utils.utils.`**`rows_from_file`**(*fp*, *format=None*, *dialect=None*, *encoding=None*, *ignore_extras=False*, *extras_key=None*)

Load a sequence of dictionaries from a file-like object containing one of four different formats.

```
from sqlite_utils.utils import rows_from_file
import io

rows, format = rows_from_file(io.StringIO("id,name\n1,Cleo"))
print(list(rows), format)
# Outputs [{'id': '1', 'name': 'Cleo'}] Format.CSV
```

This defaults to attempting to automatically detect the format of the data, or you can pass in an explicit format using the format= option.

Returns a tuple of `(rows_generator, format_used)` where `rows_generator` can be iterated over to return dictionaries, while `format_used` is a value from the `sqlite_utils.utils.Format` enum:

```
class Format(enum.Enum):
    CSV = 1
    TSV = 2
    JSON = 3
    NL = 4
```

If a CSV or TSV file includes rows with more fields than are declared in the header a `sqlite_utils.utils.RowError` exception will be raised when you loop over the generator.

You can instead ignore the extra data by passing `ignore_extras=True`.

Or pass `extras_key="rest"` to put those additional values in a list in a key called `rest`.

Parameters

- **fp** (*BinaryIO*) – a file-like object containing binary data
- **format** (*Format | None*) – the format to use - omit this to detect the format
- **dialect** (*Type[Dialect] | None*) – the CSV dialect to use - omit this to detect the dialect
- **encoding** (*str | None*) – the character encoding to use when reading CSV/TSV data
- **ignore_extras** (*bool | None*) – ignore any extra fields on rows

- **extras_key** (*str | None*) – put any extra fields in a list with this key

Return type
Tuple[*Iterable*[*Dict*[str, None | int | float | str | bytes | bool | *List*[str]]], *Format*]

1.3.51 Setting the maximum CSV field size limit

Sometimes when working with CSV files that include extremely long fields you may see an error that looks like this:

```
_csv.Error: field larger than field limit (131072)
```

The Python standard library `csv` module enforces a field size limit. You can increase that limit using the `csv.field_size_limit(new_limit)` method (documented here) but if you don't want to pick a new level you may instead want to increase it to the maximum possible.

The maximum possible value for this is not documented, and varies between systems.

Calling `sqlite_utils.utils.maximize_csv_field_size_limit()` will set the value to the highest possible for the current system:

```
from sqlite_utils.utils import maximize_csv_field_size_limit

maximize_csv_field_size_limit()
```

If you need to reset to the original value after calling this function you can do so like this:

```
from sqlite_utils.utils import ORIGINAL_CSV_FIELD_SIZE_LIMIT
import csv

csv.field_size_limit(ORIGINAL_CSV_FIELD_SIZE_LIMIT)
```

1.3.52 Detecting column types using TypeTracker

Sometimes you may find yourself working with data that lacks type information - data from a CSV file for example.

The `TypeTracker` class can be used to try to automatically identify the most likely types for data that is initially represented as strings.

Consider this example:

```
import csv, io

csv_file = io.StringIO("id,name\n1,Cleo\n2,Cardi")
rows = list(csv.DictReader(csv_file))

# rows is now this:
# [{'id': '1', 'name': 'Cleo'}, {'id': '2', 'name': 'Cardi'}]
```

If we insert this data directly into a table we will get a schema that is entirely `TEXT` columns:

```
from sqlite_utils import Database

db = Database(memory=True)
db.table("creatures").insert_all(rows)
print(db.schema)
# Outputs:
```

(continues on next page)

(continued from previous page)

```
# CREATE TABLE "creatures" (
#    "id" TEXT,
#    "name" TEXT
# );
```

We can detect the best column types using a `TypeTracker` instance:

```
from sqlite_utils.utils import TypeTracker

tracker = TypeTracker()
db.table("creatures2").insert_all(tracker.wrap(rows))
print(tracker.types)
# Outputs {'id': 'integer', 'name': 'text'}
```

We can then apply those types to our new table using the *table.transform()* method:

```
db.table("creatures2").transform(types=tracker.types)
print(db.table("creatures2").schema)
# Outputs:
# CREATE TABLE "creatures2" (
#    "id" INTEGER,
#    "name" TEXT
# );
```

1.3.53 SpatiaLite helpers

SpatiaLite is a geographic extension to SQLite (similar to PostgreSQL + PostGIS). Using requires finding, loading and initializing the extension, adding geometry columns to existing tables and optionally creating spatial indexes. The utilities here help streamline that setup.

Initialize SpatiaLite

`Database.`**`init_spatialite`**(*path=None*)

The `init_spatialite` method will load and initialize the SpatiaLite extension. The `path` argument should be an absolute path to the compiled extension, which can be found using `find_spatialite`.

Returns `True` if SpatiaLite was successfully initialized.

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite(find_spatialite())
```

If you've installed SpatiaLite somewhere unexpected (for testing an alternate version, for example) you can pass in an absolute path:

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite("./local/mod_spatialite.dylib")
```

Parameters
path (*str | None*) – Path to SpatiaLite module on disk

Return type
bool

Finding SpatiaLite

sqlite_utils.utils.**find_spatialite**()

The `find_spatialite()` function searches for the SpatiaLite SQLite extension in some common places. It returns a string path to the location, or `None` if SpatiaLite was not found.

You can use it in code like this:

```
from sqlite_utils import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
spatialite = find_spatialite()
if spatialite:
    db.conn.enable_load_extension(True)
    db.conn.load_extension(spatialite)

# or use with db.init_spatialite like this
db.init_spatialite(find_spatialite())
```

Return type
str | None

Adding geometry columns

Table.**add_geometry_column**(*column_name*, *geometry_type*, *srid=4326*, *coord_dimension='XY'*, *not_null=False*)

In SpatiaLite, a geometry column can only be added to an existing table. To do so, use `table.add_geometry_column`, passing in a geometry type.

By default, this will add a nullable column using SRID 4326. This can be customized using the `column_name`, `srid` and `not_null` arguments.

Returns `True` if the column was successfully added, `False` if not.

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite(find_spatialite())

# the table must exist before adding a geometry column
table = db["locations"].create({"name": str})
table.add_geometry_column("geometry", "POINT")
```

Parameters

- **column_name** (*str*) – Name of column to add
- **geometry_type** (*str*) – Type of geometry column, for example `"GEOMETRY"` or `"POINT"` or ``"POLYGON"`

- **srid** (*int*) – Integer SRID, defaults to 4326 for WGS84
- **coord_dimension** (*str*) – Dimensions to use, defaults to "XY" - set to "XYZ" to work in three dimensions
- **not_null** (*bool*) – Should the column be NOT NULL

Return type
bool

Creating a spatial index

Table.**create_spatial_index**(*column_name*)

A spatial index allows for significantly faster bounding box queries. To create one, use create_spatial_index with the name of an existing geometry column.

Returns True if the index was successfully created, False if not. Calling this function if an index already exists is a no-op.

```
# assuming SpatiaLite is loaded, create the table, add the column
table = db["locations"].create({"name": str})
table.add_geometry_column("geometry", "POINT")

# now we can index it
table.create_spatial_index("geometry")

# the spatial index is a virtual table, which we can inspect
print(db["idx_locations_geometry"].schema)
# outputs:
# CREATE VIRTUAL TABLE "idx_locations_geometry" USING rtree(pkid, xmin, xmax, ymin,␣
↪ymax)
```

Parameters
column_name – Geometry column to create the spatial index against

Return type
bool

1.4 Database migrations

sqlite-utils includes a migration system for applying repeatable changes to SQLite database files.

A migration is a Python function that receives a sqlite_utils.Database instance and then executes Python code to modify that database - creating or transforming tables, adding indexes, inserting rows, or any other operation supported by SQLite.

Migrations are grouped into named sets using the sqlite_utils.Migrations class, and each applied migration is recorded in the _sqlite_migrations table in that database.

This means you can run the migrate operation multiple times and it will only apply migrations that have not previously been recorded.

1.4.1 Defining migrations

Ordered migration sets are defined by first creating a sqlite_utils.Migrations object.

Individual migrations are Python functions that are then registered with that migration set. Each migration function is passed a single argument that is a *sqlite_utils.Database* instance.

The name passed to `Migrations("creatures")` identifies that set of migrations. Use a name that is unique for your project, since multiple migration sets can be applied to the same database.

Here is a simple example of a `migrations.py` file which creates a table, then adds an extra column to that table in a second migration:

```
from sqlite_utils import Migrations

migrations = Migrations("creatures")

@migrations()
def create_table(db):
    db["creatures"].create(
        {"id": int, "name": str, "species": str},
        pk="id",
    )

@migrations()
def add_weight(db):
    db["creatures"].add_column("weight", float)
```

1.4.2 Applying migrations in Python

Once you have a `Migrations(name)` collection with one or more migrations registered to it, you can execute them in Python code like this:

```
from sqlite_utils import Database

db = Database("creatures.db")
migrations.apply(db)
```

Running `migrations.apply(db)` repeatedly is safe. Migrations that already have a matching `migration_set` and `name` row in `_sqlite_migrations` will be skipped.

Migration functions are applied in the order that they were registered. The function name is used as the migration name unless you pass one explicitly:

```
@migrations(name="001_create_table")
def create_table(db):
    db["creatures"].create({"id": int, "name": str}, pk="id")
```

When you apply a set of migrations you can stop part way through by specifying a `stop_before=` migration name:

```
migrations.apply(db, stop_before="add_weight")
```

1.4.3 Migrations and transactions

Each migration runs inside a transaction, together with the `_sqlite_migrations` record of it having been applied. If a migration function raises an exception, everything it did is rolled back, no record is written and the migration stays pending - so fixing the error and re-applying will run that migration again from a clean state. Migrations that completed earlier in the same `apply()` run stay applied.

Some operations cannot run inside a transaction, for example `VACUUM` or changing the journal mode with `db.enable_wal()`. Register migrations like these with `transactional=False`:

```
@migrations(transactional=False)
def compact(db):
    db.execute("VACUUM")
```

A migration registered with `transactional=False` runs without a wrapping transaction, so if it fails part way through any changes it already made will not be rolled back, and re-applying will run the whole function again.

Avoid calling `db.commit()` or otherwise managing transactions manually inside a transactional migration - register the migration with `transactional=False` if it needs to control its own transactions. Using `with db.atomic():` blocks inside a migration is fine: they nest as savepoints within the migration's transaction, so the migration as a whole still commits or rolls back as a single unit. See *Transactions and saving your changes*.

1.4.4 Applying migrations using the CLI

Run migrations using the `sqlite-utils migrate` command:

```
sqlite-utils migrate creatures.db path/to/migrations.py
```

The first argument is the database file. The remaining arguments can be paths to migration files or directories containing migration files.

If you omit migration paths, `sqlite-utils` searches the current directory and subdirectories for files called `migrations.py`:

```
sqlite-utils migrate creatures.db
```

You can also pass a directory. Every `migrations.py` file in that directory tree will be considered:

```
sqlite-utils migrate creatures.db path/to/project/
```

Running the command repeatedly is safe. Migrations that already have a matching `migration_set` and `name` row in `_sqlite_migrations` will be skipped.

1.4.5 Listing migrations

Use `--list` to show applied and pending migrations without running them. This is a read-only operation - it will not create the database file or the `_sqlite_migrations` table:

```
sqlite-utils migrate creatures.db --list
```

Example output:

```
Migrations for: creatures

  Applied:
    create_table - 2026-06-09 17:23:12.048092+00:00
    add_weight - 2026-06-09 17:23:12.051249+00:00

  Pending:
    add_age
```

1.4.6 Stopping before a migration

When applying migrations using the CLI, you can stop before a named migration:

```
sqlite-utils migrate creatures.db path/to/migrations.py --stop-before add_weight
```

This applies any pending migrations before `add_weight` and leaves `add_weight` and later migrations pending. An unqualified migration name matches in any migration set.

You can also target a specific migration set using `migration_set:migration_name`. This is useful if a migrations file contains more than one migration set, or if multiple sets use the same migration name:

```
sqlite-utils migrate creatures.db path/to/migrations.py \
  --stop-before creatures:add_weight \
  --stop-before sales:drop_index
```

The `--stop-before` option can be passed more than once.

If a `--stop-before` value does not match any known migration the command exits with an error, rather than silently applying everything. Naming a migration that has already been applied is also an error - stopping before it is impossible to honor - and no pending migrations are applied.

1.4.7 Verbose output

Use `--verbose` or `-v` to show the schema before and after migrations are applied, plus a unified diff when the schema changes:

```
sqlite-utils migrate creatures.db --verbose
```

1.4.8 Migrating from sqlite-migrate

This system uses the same migration table format as the older sqlite-migrate package. To use existing migration files directly with `sqlite-utils`, update their import from `sqlite_migrate` to `sqlite_utils`:

```
from sqlite_utils import Migrations

migration = Migrations("creatures")

@migration()
def create_table(db):
    db["creatures"].create({"id": int, "name": str}, pk="id")
```

1.4.9 Python API

class sqlite_utils.migrations.**Migrations**(*name*)

Parameters
name (*str*)

migrations_table = '_sqlite_migrations'

pending(*db*)

Return a list of pending migrations.

This is a read-only operation - it does not write to the database.

Parameters
db (Database)

Return type
list[Migrations._Migration]

`applied`(*db*)

Return a list of applied migrations, in the order they were applied.

This is a read-only operation - it does not write to the database.

Parameters
db (Database)

Return type
list[Migrations._AppliedMigration]

`apply`(*db*, ** (Keyword-only parameters separator (PEP 3102))*, *stop_before=None*)

Apply any pending migrations to the database.

Each migration runs inside a transaction, together with the record of it having been applied - if the migration raises an exception its changes are rolled back, no record is written and the migration stays pending. Migrations registered with `transactional=False` run outside of a transaction.

Raises
ValueError – if a `stop_before` name matches a migration in this set that has already been applied - stopping before it is impossible to honor, and no pending migrations are applied

Parameters

- **db** (Database)
- **stop_before** (*str | Iterable[str] | None*)

`ensure_migrations_table`(*db*)

Ensure the _sqlite_migrations table exists and has the correct schema.

Parameters
db (Database)

1.5 Plugins

`sqlite-utils` supports plugins, which can be used to add extra features to the software.

Plugins can add new commands, for example `sqlite-utils some-command ...`

Plugins can be installed using the `sqlite-utils install` command:

```
sqlite-utils install sqlite-utils-name-of-plugin
```

You can see a JSON list of plugins that have been installed by running this:

```
sqlite-utils plugins
```

Plugin hooks such as *prepare_connection(conn)* affect each instance of the `Database` class. You can opt-out of these plugins by creating that class instance like so:

```
db = Database(memory=True, execute_plugins=False)
```

1.5.1 Building a plugin

Plugins are created in a directory named after the plugin. To create a "hello world" plugin, first create a `hello-world` directory:

```
mkdir hello-world
cd hello-world
```

In that folder create two files. The first is a `pyproject.toml` file describing the plugin:

```
[project]
name = "sqlite-utils-hello-world"
version = "0.1"

[project.entry-points.sqlite_utils]
hello_world = "sqlite_utils_hello_world"
```

The `[project.entry-points.sqlite_utils]` section tells `sqlite-utils` which module to load when executing the plugin.

Then create `sqlite_utils_hello_world.py` with the following content:

```
import click
import sqlite_utils

@sqlite_utils.hookimpl
def register_commands(cli):
    @cli.command()
    def hello_world():
        "Say hello world"
        click.echo("Hello world!")
```

Install the plugin in "editable" mode - so you can make changes to the code and have them picked up instantly by `sqlite-utils` - like this:

```
sqlite-utils install -e .
```

Or pass the path to your plugin directory:

```
sqlite-utils install -e /dev/sqlite-utils-hello-world
```

Now, running this should execute your new command:

```
sqlite-utils hello-world
```

Your command will also be listed in the output of `sqlite-utils --help`.

See the LLM plugin documentation for tips on distributing your plugin.

1.5.2 Plugin hooks

Plugin hooks allow `sqlite-utils` to be customized.

register_commands(cli)

This hook can be used to register additional commands with the `sqlite-utils` CLI. It is called with the `cli` object, which is a `click.Group` instance.

Example implementation:

```
import click
import sqlite_utils

@sqlite_utils.hookimpl
def register_commands(cli):
    @cli.command()
    def hello_world():
        "Say hello world"
        click.echo("Hello world!")
```

New commands implemented by plugins can invoke existing commands using the context.invoke mechanism.

As a special niche feature, if your plugin needs to import some files and then act against an in-memory database containing those files you can forward to the *sqlite-utils memory command* and pass it `return_db=True`:

```
@cli.command()
@click.pass_context
@click.argument(
    "paths",
    type=click.Path(file_okay=True, dir_okay=False, allow_dash=True),
    required=False,
    nargs=-1,
)
def show_schema_for_files(ctx, paths):
    from sqlite_utils.cli import memory
    db = ctx.invoke(memory, paths=paths, return_db=True)
    # Now do something with that database
    click.echo(db.schema)
```

prepare_connection(conn)

This hook is called when a new SQLite database connection is created. You can use it to register custom SQL functions, aggregates and collations. For example:

```
import sqlite_utils

@sqlite_utils.hookimpl
def prepare_connection(conn):
    conn.create_function(
        "hello", 1, lambda name: f"Hello, {name}!"
    )
```

This registers a SQL function called `hello` which takes a single argument and can be called like this:

```
select hello("world"); -- "Hello, world!"
```

1.6 API reference

1.6.1 sqlite_utils.db.Database

class sqlite_utils.db.**Database**(*filename_or_conn=None*, *memory=False*, *memory_name=None*, *recreate=False*, *recursive_triggers=True*, *tracer=None*, *use_counts_table=False*, *execute_plugins=True*, *use_old_upsert=False*, *strict=False*)

Wrapper for a SQLite database connection that adds a variety of useful utility methods.

To create an instance:

```
# create data.db file, or open existing:
db = Database("data.db")
# Create an in-memory database:
dB = Database(memory=True)
```

Parameters

- **filename_or_conn** (*str | Path | Connection | None*) – String path to a file, or a `pathlib.Path` object, or a `sqlite3` connection
- **memory** (*bool*) – set to `True` to create an in-memory database
- **memory_name** (*str | None*) – creates a named in-memory database that can be shared across multiple connections
- **recreate** (*bool*) – set to `True` to delete and recreate a file database (**dangerous**)
- **recursive_triggers** (*bool*) – defaults to `True`, which sets `PRAGMA recursive_triggers=on;` - set to `False` to avoid setting this pragma

- **tracer** (*Callable[[str, Sequence[Any] | Dict[str, Any] | None], None] | None*) – set a tracer function (`print` works for this) which will be called with `sql, parameters` every time a SQL query is executed
- **use_counts_table** (*bool*) – set to `True` to use a cached counts table, if available. See *Cached table counts using triggers*
- **use_old_upsert** (*bool*) – set to `True` to force the older upsert implementation. See *Alternative upserts using INSERT OR IGNORE*
- **strict** (*bool*) – Apply STRICT mode to all created tables (unless overridden)
- **execute_plugins** (*bool*)

conn: Connection

close()

Close the SQLite connection, and the underlying database file

Return type
None

atomic()

Context manager for wrapping multiple database operations in a transaction.

Nested blocks use SQLite savepoints.

Return type
Generator[Database, None, None]

begin()

Start a transaction with `BEGIN`, taking manual control of transaction handling. End it by calling *commit()* or *rollback()*.

Raises `sqlite3.OperationalError` if a transaction is already open.

Most code should use the *atomic()* context manager instead, which commits and rolls back automatically. See *Transactions and saving your changes*.

Return type
None

commit()

Commit the current transaction. Does nothing if no transaction is open.

Return type
None

rollback()

Roll back the current transaction, discarding its changes. Does nothing if no transaction is open.

Return type
None

ensure_autocommit_on()

Ensure the connection is in driver-level autocommit mode for the duration of a block of code.

This temporarily sets `isolation_level = None` on the underlying `sqlite3` connection, so the driver does not open implicit transactions. This is useful for statements such as `PRAGMA journal_mode=wal` which cannot run inside a transaction.

Example usage:

```
with db.ensure_autocommit_on():
    # do stuff here
```

The previous `isolation_level` is restored at the end of the block.

Raises
TransactionError – if a transaction is open - assigning `isolation_level` would commit it as a side effect, silently breaking the caller's ability to roll back

Return type
Generator[None, None, None]

tracer(*tracer=None*)

Context manager to temporarily set a tracer function - all executed SQL queries will be passed to this.

The tracer function should accept two arguments: `sql` and `parameters`

Example usage:

```
with db.tracer(print):
    db["creatures"].insert({"name": "Cleo"})
```

See *Tracing queries*.

Parameters
tracer (*Callable[[str, Sequence[Any] | Dict[str, Any] | None], None] | None*) – Callable accepting `sql` and `parameters` arguments

Return type
Generator[Database, None, None]

__getitem__(*table_name*)

`db[name]` returns a *Table* object for the table with the specified name, or a *View* object if the name matches an existing SQL view. If neither exists yet, a table is assumed - it will be created the first time data is inserted into it.

Parameters
table_name (*str*) – The name of the table or view

Return type
Table | View

register_function(*fn=None*, *deterministic=False*, *replace=False*, *name=None*)

`fn` will be made available as a function within SQL, with the same name and number of arguments. Can be used as a decorator:

```
@db.register_function
def upper(value):
    return str(value).upper()
```

The decorator can take arguments:

```
@db.register_function(deterministic=True, replace=True)
def upper(value):
    return str(value).upper()
```

See *Registering custom SQL functions*.

Parameters

- **fn** (*Callable | None*) – Function to register
- **deterministic** (*bool*) – set `True` for functions that always returns the same output for a given input
- **replace** (*bool*) – set `True` to replace an existing function with the same name - otherwise throw an error
- **name** (*str | None*) – name of the SQLite function - if not specified, the Python function name will be used

Return type
Callable[[*Callable*], *Callable*] | None

register_fts4_bm25()

Register the `rank_bm25(match_info)` function used for calculating relevance with SQLite FTS4.

Return type
None

attach(*alias*, *filepath*)

Attach another SQLite database file to this connection with the specified alias, equivalent to:

```
ATTACH DATABASE 'filepath.db' AS alias
```

Parameters

- **alias** (*str*) – Alias name to use
- **filepath** (*str | Path*) – Path to SQLite database file on disk

Return type
None

query(*sql*, *params=None*)

Execute `sql` and return an iterable of dictionaries representing each row.

The SQL is executed as soon as this method is called - the resulting rows are then fetched lazily as the returned iterable is iterated over. A row-returning write such as `INSERT ... RETURNING` takes effect immediately, even if the results are never iterated.

Parameters

- **sql** (*str*) – SQL query to execute
- **params** (*Sequence | Dict[str, Any] | None*) – Parameters to use in that query - an iterable for `where id = ?` parameters, or a dictionary for `where id = :id`

Raises
ValueError – if the SQL statement does not return rows - use *execute()* for those statements instead. The rejected statement is rolled back, so it has no effect on the database. One exception: a row-less `PRAGMA` statement takes effect despite the `ValueError`, because PRAGMAs run outside the savepoint guard - some of them refuse to run inside a transaction

Return type
Generator[dict, None, None]

execute(*sql*, *parameters=None*)

Execute SQL query and return a `sqlite3.Cursor`.

A write statement - `INSERT`, `UPDATE`, `CREATE TABLE` and so on - is committed automatically, unless a transaction is already open, in which case it becomes part of that transaction. See *Transactions and saving your changes*.

Parameters

- **sql** (*str*) – SQL query to execute
- **parameters** (*Sequence | Dict[str, Any] | None*) – Parameters to use in that query - an iterable for `where id = ?` parameters, or a dictionary for `where id = :id`

Return type
Cursor

executescript(*sql*)

Execute multiple SQL statements separated by ; and return the `sqlite3.Cursor`.

Parameters
sql (*str*) – SQL to execute

Return type
Cursor

table(*table_name*, ***kwargs*)

Return a table object, optionally configured with default options.

See *sqlite_utils.db.Table* for option details.

Parameters

- **table_name** (*str*) – Name of the table
- **kwargs** (*Any*)

Return type
Table

view(*view_name*)

Return a view object.

Parameters
view_name (*str*) – Name of the view

Return type
View

quote(*value*)

Apply SQLite string quoting to a value, including wrapping it in single quotes.

Parameters
value (*str*) – String to quote

Return type
str

quote_fts(*query*)

Escape special characters in a SQLite full-text search query.

This works by surrounding each token within the query with double quotes, in order to avoid words like `NOT` and `OR` having special meaning as defined by the FTS query syntax here:

https://www.sqlite.org/fts5.html#full_text_query_syntax

If the query has unbalanced `"` characters, adds one at end.

Parameters
query (*str*) – String to escape

Return type
str

`quote_default_value`(*value*)

Parameters
value (*str*)

Return type
str

`table_names`(*fts4=False*, *fts5=False*)

List of string table names in this database.

Parameters

- **fts4** (*bool*) – Only return tables that are part of FTS4 indexes
- **fts5** (*bool*) – Only return tables that are part of FTS5 indexes

Return type
List[str]

`view_names`()

List of string view names in this database.

Return type
List[str]

`property tables: List[Table]`

List of Table objects in this database.

`property views: List[View]`

List of View objects in this database.

`property triggers: List[Trigger]`

List of `(name, table_name, sql)` tuples representing triggers in this database.

`property triggers_dict: Dict[str, str]`

A `{trigger_name: sql}` dictionary of triggers in this database.

`property schema: str`

SQL schema for this database.

`property supports_strict: bool`

Does this database support STRICT mode?

`property supports_on_conflict: bool`

`property sqlite_version: Tuple[int, ...]`

Version of SQLite, as a tuple of integers for example `(3, 36, 0)`.

`property journal_mode: str`

Current `journal_mode` of this database.

https://www.sqlite.org/pragma.html#pragma_journal_mode

enable_wal()

Sets `journal_mode` to `'wal'` to enable Write-Ahead Log mode.

Raises
TransactionError – if called while a transaction is open - the journal mode can only be changed outside of a transaction

Return type
None

disable_wal()

Sets `journal_mode` back to `'delete'` to disable Write-Ahead Log mode.

Raises
TransactionError – if called while a transaction is open - the journal mode can only be changed outside of a transaction

Return type
None

enable_counts()

Enable trigger-based count caching for every table in the database, see *Cached table counts using triggers*.

Return type
None

cached_counts(*tables=None*)

Return `{table_name: count}` dictionary of cached counts for specified tables, or all tables if `tables` not provided.

Parameters
tables (*Iterable[str] | None*) – Subset list of tables to return counts for.

Return type
Dict[str, int]

reset_counts()

Re-calculate cached counts for tables.

Return type
None

create_table_sql(*name*, *columns*, *pk=None*, *foreign_keys=None*, *column_order=None*, *not_null=None*, *defaults=None*, *hash_id=None*, *hash_id_columns=None*, *extracts=None*, *if_not_exists=False*, *strict=False*)

Returns the SQL `CREATE TABLE` statement for creating the specified table.

Parameters

- **name** (*str*) – Name of table
- **columns** (*Dict[str, Any]*) – Dictionary mapping column names to their types, for example `{"name": str, "age": int}`
- **pk** (*Any | None*) – String name of column to use as a primary key, or a tuple of strings for a compound primary key covering multiple columns
- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str*

| ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None) – List of foreign key definitions for this table

- **column_order** (*List[str] | None*) – List specifying which columns should come first
- **not_null** (*Iterable[str] | None*) – List of columns that should be created as `NOT NULL`
- **defaults** (*Dict[str, Any] | None*) – Dictionary specifying default values for columns
- **hash_id** (*str | None*) – Name of column to be used as a primary key containing a hash of the other columns
- **hash_id_columns** (*Iterable[str] | None*) – List of columns to be used when calculating the hash ID for a row
- **extracts** (*Dict[str, str] | List[str] | None*) – List or dictionary of columns to be extracted during inserts, see *Populating lookup tables automatically during insert/upsert*
- **if_not_exists** (*bool*) – Use `CREATE TABLE IF NOT EXISTS`
- **strict** (*bool*) – Apply STRICT mode to table

Return type
str

create_table(*name*, *columns*, *pk=None*, *foreign_keys=None*, *column_order=None*, *not_null=None*, *defaults=None*, *hash_id=None*, *hash_id_columns=None*, *extracts=None*, *if_not_exists=False*, *replace=False*, *ignore=False*, *transform=False*, *strict=False*)

Create a table with the specified name and the specified `{column_name: type}` columns.

See *Explicitly creating a table*.

Parameters

- **name** (*str*) – Name of table
- **columns** (*Dict[str, Any]*) – Dictionary mapping column names to their types, for example `{"name": str, "age": int}`
- **pk** (*Any | None*) – String name of column to use as a primary key, or a tuple of strings for a compound primary key covering multiple columns
- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*) – List of foreign key definitions for this table
- **column_order** (*List[str] | None*) – List specifying which columns should come first
- **not_null** (*Iterable[str] | None*) – List of columns that should be created as `NOT NULL`

- **defaults** (*Dict[str, Any] | None*) – Dictionary specifying default values for columns
- **hash_id** (*str | None*) – Name of column to be used as a primary key containing a hash of the other columns
- **hash_id_columns** (*Iterable[str] | None*) – List of columns to be used when calculating the hash ID for a row
- **extracts** (*Dict[str, str] | List[str] | None*) – List or dictionary of columns to be extracted during inserts, see *Populating lookup tables automatically during insert/upsert*
- **if_not_exists** (*bool*) – Use `CREATE TABLE IF NOT EXISTS`
- **replace** (*bool*) – Drop and replace table if it already exists
- **ignore** (*bool*) – Silently do nothing if table already exists
- **transform** (*bool*) – If table already exists transform it to fit the specified schema
- **strict** (*bool*) – Apply STRICT mode to table

Return type
Table

rename_table(*name*, *new_name*)

Rename a table.

Parameters

- **name** (*str*) – Current table name
- **new_name** (*str*) – Name to rename it to

Return type
None

create_view(*name*, *sql*, *ignore=False*, *replace=False*)

Create a new SQL view with the specified name - `sql` should start with `SELECT ...`.

Parameters

- **name** (*str*) – Name of the view
- **sql** (*str*) – SQL `SELECT` query to use for this view.
- **ignore** (*bool*) – Set to `True` to do nothing if a view with this name already exists
- **replace** (*bool*) – Set to `True` to replace the view if one with this name already exists

Return type
Database

m2m_table_candidates(*table*, *other_table*)

Given two table names returns the name of tables that could define a many-to-many relationship between those two tables, based on having foreign keys to both of the provided tables.

Parameters

- **table** (*str*) – Table name
- **other_table** (*str*) – Other table name

Return type
List[str]

add_foreign_keys(*foreign_keys*)

See *Adding multiple foreign key constraints at once*.

Parameters
foreign_keys (*Iterable[ForeignKey | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]]*) – A list of `(table, column, other_table, other_column)` tuples - for compound foreign keys, `column` and `other_column` can be tuples of column names

Return type
None

index_foreign_keys()

Create indexes for every foreign key column on every table in the database.

Return type
None

vacuum()

Run a SQLite `VACUUM` against the database.

Return type
None

analyze(*name=None*)

Run `ANALYZE` against the entire database or a named table or index.

Parameters
name (*str | None*) – Run `ANALYZE` against this specific named table or index

Return type
None

iterdump()

A sequence of strings representing a SQL dump of the database

Return type
Generator[str, None, None]

init_spatialite(*path=None*)

The `init_spatialite` method will load and initialize the SpatiaLite extension. The `path` argument should be an absolute path to the compiled extension, which can be found using `find_spatialite`.

Returns `True` if SpatiaLite was successfully initialized.

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite(find_spatialite())
```

If you've installed SpatiaLite somewhere unexpected (for testing an alternate version, for example) you can pass in an absolute path:

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite("./local/mod_spatialite.dylib")
```

Parameters
path (*str | None*) – Path to SpatiaLite module on disk

Return type
bool

1.6.2 sqlite_utils.db.Queryable

Table and *View* are both subclasses of `Queryable`, providing access to the following methods:

class sqlite_utils.db.**Queryable**(*db*, *name*)

Parameters

- **db** (Database)
- **name** (*str*)

exists()

Does this table or view exist yet?

Return type
bool

db: *Database*

name: str

count_where(*where=None*, *where_args=None*)

Executes `SELECT count(*) FROM table WHERE ...` and returns a count.

Parameters

- **where** (*str | None*) – SQL where fragment to use, for example `id > ?`
- **where_args** (*Sequence | Dict[str, Any] | None*) – Parameters to use with that fragment - an iterable for `id > ?` parameters, or a dictionary for `id > :id`

Return type
int

property count: int

A count of the rows in this table or view.

property rows: Generator[Dict[str, Any], None, None]

Iterate over every dictionaries for each row in this table or view.

rows_where(*where=None*, *where_args=None*, *order_by=None*, *select='*'*, *limit=None*, *offset=None*)

Iterate over every row in this table or view that matches the specified where clause.

Returns each row as a dictionary. See *Listing rows* for more details.

Parameters

- **where** (*str | None*) – SQL where fragment to use, for example `id > ?`
- **where_args** (*Sequence | Dict[str, Any] | None*) – Parameters to use with that fragment - an iterable for `id > ?` parameters, or a dictionary for `id > :id`
- **order_by** (*str | None*) – Column or fragment of SQL to order by
- **select** (*str*) – Comma-separated list of columns to select - defaults to *
- **limit** (*int | None*) – Integer number of rows to limit to

- **offset** (*int | None*) – Integer for SQL offset

Return type
Generator[*Dict*[str, *Any*], None, None]

pks_and_rows_where(*where=None*, *where_args=None*, *order_by=None*, *limit=None*, *offset=None*)

Like `.rows_where()` but returns `(pk, row)` pairs - `pk` can be a single value or tuple.

Parameters

- **where** (*str | None*) – SQL where fragment to use, for example `id > ?`
- **where_args** (*Sequence | Dict[str, Any] | None*) – Parameters to use with that fragment - an iterable for `id > ?` parameters, or a dictionary for `id > :id`
- **order_by** (*str | None*) – Column or fragment of SQL to order by
- **select** – Comma-separated list of columns to select - defaults to *
- **limit** (*int | None*) – Integer number of rows to limit to
- **offset** (*int | None*) – Integer for SQL offset

Return type
Generator[*Tuple*[*Any*, *Dict*[str, *Any*]], None, None]

property columns: List[*Column*]

List of *Columns* representing the columns in this table or view.

property columns_dict: Dict[str, Any]

`{column_name: python-type}` dictionary representing columns in this table or view.

property schema: str

SQL schema for this table or view.

1.6.3 sqlite_utils.db.Table

class sqlite_utils.db.**Table**(*db*, *name*, *pk=None*, *foreign_keys=None*, *column_order=None*, *not_null=None*, *defaults=None*, *batch_size=100*, *hash_id=None*, *hash_id_columns=None*, *alter=False*, *ignore=False*, *replace=False*, *extracts=None*, *conversions=None*, *columns=None*, *strict=False*)

Bases: *Queryable*

Tables should usually be initialized using the `db.table(table_name)` or `db[table_name]` methods.

The following optional parameters can be passed to `db.table(table_name, ...)`:

Parameters

- **db** (Database) – Provided by `db.table(table_name)`
- **name** (*str*) – Provided by `db.table(table_name)`
- **pk** (*Any | None*) – Name of the primary key column, or tuple of columns
- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*) – List of foreign key definitions

- **column_order** (*List[str] | None*) – List of column names in the order they should be in the table
- **not_null** (*Iterable[str] | None*) – List of columns that cannot be null
- **defaults** (*Dict[str, Any] | None*) – Dictionary of column names and default values
- **batch_size** (*int*) – Integer number of rows to insert at a time
- **hash_id** (*str | None*) – Name of a column to create and use as a primary key, where the value of that primary key is derived from a hash of the row values
- **hash_id_columns** (*Iterable[str] | None*) – List of columns to use for the hash_id
- **alter** (*bool*) – If True, automatically alter the table if it doesn't match the schema
- **ignore** (*bool*) – If True, ignore rows that already exist when inserting
- **replace** (*bool*) – If True, replace rows that already exist when inserting
- **extracts** (*Dict[str, str] | List[str] | None*) – Dictionary or list of column names to extract into a separate table on inserts
- **conversions** (*dict | None*) – Dictionary of column names and conversion functions
- **columns** (*Dict[str, Any] | None*) – Dictionary of column names to column types
- **strict** (*bool*) – If True, apply STRICT mode to table

last_rowid: int | None = None

The `rowid` of the last inserted, updated or selected row.

last_pk: Any | None = None

The primary key of the last inserted, updated or selected row.

property count: int

Count of the rows in this table - optionally from the table count cache, if configured.

exists()

Does this table or view exist yet?

Return type
bool

property pks: List[str]

Primary key columns for this table, in PRIMARY KEY declaration order - `PRAGMA table_info` sets `is_pk` to the 1-based position of each column within the primary key, which can differ from the order of the columns in the table. SQLite uses the declaration order to resolve implicit foreign key references, so this order matters.

property use_rowid: bool

Does this table use `rowid` for its primary key (no other primary keys are specified)?

get(*pk_values*)

Return row (as dictionary) for the specified primary key.

Raises `sqlite_utils.db.NotFoundError` if a matching row cannot be found.

Parameters
pk_values (*list | tuple | str | int*) – A single value, or a tuple of values for tables that have a compound primary key

Return type
dict

property foreign_keys: List[*ForeignKey*]

List of foreign keys defined on this table.

Compound (multi-column) foreign keys are returned as a single `ForeignKey` with `is_compound=True` and populated `columns`/`other_columns` lists.

property virtual_table_using: str | None

Type of virtual table, or `None` if this is not a virtual table.

property indexes: List[Index]

List of indexes defined on this table.

property xindexes: List[XIndex]

List of indexes defined on this table using the more detailed `XIndex` format.

property triggers: List[Trigger]

List of triggers defined on this table.

property triggers_dict: Dict[str, str]

`{trigger_name: sql}` dictionary of triggers defined on this table.

property default_values: Dict[str, Any]

`{column_name: default_value}` dictionary of default values for columns in this table.

property strict: bool

Is this a STRICT table?

create(*columns*, *pk=<sqlite_utils.db.Default object>*, *foreign_keys=<sqlite_utils.db.Default object>*, *column_order=<sqlite_utils.db.Default object>*, *not_null=<sqlite_utils.db.Default object>*, *defaults=<sqlite_utils.db.Default object>*, *hash_id=<sqlite_utils.db.Default object>*, *hash_id_columns=<sqlite_utils.db.Default object>*, *extracts=<sqlite_utils.db.Default object>*, *if_not_exists=False*, *replace=False*, *ignore=False*, *transform=False*, *strict=<sqlite_utils.db.Default object>*)

Create a table with the specified columns.

See *Explicitly creating a table* for full details.

Parameters

- **columns** (*Dict[str, Any]*) – Dictionary mapping column names to their types, for example `{"name": str, "age": int}`
- **pk** (*Any | None*) – String name of column to use as a primary key, or a tuple of strings for a compound primary key covering multiple columns
- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None | Default*) – List of foreign key definitions for this table
- **column_order** (*List[str] | None | Default*) – List specifying which columns should come first
- **not_null** (*Iterable[str] | None | Default*) – List of columns that should be created as `NOT NULL`

- **defaults** (*Dict[str, Any] | None | Default*) – Dictionary specifying default values for columns
- **hash_id** (*str | None | Default*) – Name of column to be used as a primary key containing a hash of the other columns
- **hash_id_columns** (*Iterable[str] | None | Default*) – List of columns to be used when calculating the hash ID for a row
- **extracts** (*Dict[str, str] | List[str] | None | Default*) – List or dictionary of columns to be extracted during inserts, see *Populating lookup tables automatically during insert/upsert*
- **if_not_exists** (*bool*) – Use `CREATE TABLE IF NOT EXISTS`
- **replace** (*bool*) – Drop and replace table if it already exists
- **ignore** (*bool*) – Silently do nothing if table already exists
- **transform** (*bool*) – If table already exists transform it to fit the specified schema
- **strict** (*bool | Default*) – Apply STRICT mode to table

Return type
Table

duplicate(*new_name*)

Create a duplicate of this table, copying across the schema and all row data.

Parameters
new_name (*str*) – Name of the new table

Return type
Table

transform(*, *types=None*, *rename=None*, *drop=None*, *pk=<sqlite_utils.db.Default object>*, *not_null=None*, *defaults=None*, *drop_foreign_keys=None*, *add_foreign_keys=None*, *foreign_keys=None*, *column_order=None*, *keep_table=None*)

Apply an advanced alter table, including operations that are not supported by `ALTER TABLE` in SQLite itself.

See *Transforming a table* for full details.

Parameters

- **types** (*dict | None*) – Columns that should have their type changed, for example `{"weight": float}`
- **rename** (*dict | None*) – Columns to rename, for example `{"headline": "title"}`
- **drop** (*Iterable | None*) – Columns to drop
- **pk** (*Any | None*) – New primary key for the table
- **not_null** (*Iterable[str] | None*) – Columns to set as `NOT NULL`
- **defaults** (*Dict[str, Any] | None*) – Default values for columns
- **drop_foreign_keys** (*Iterable[str] | None*) – Foreign key constraints to remove - a column name drops any foreign key that column participates in, a tuple of column names drops the compound foreign key with exactly those columns
- **add_foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str |*

Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None) – List of foreign keys to add to the table

- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*) – List of foreign keys to set for the table, replacing any existing foreign keys
- **column_order** (*List[str] | None*) – List of strings specifying a full or partial column order to use when creating the table
- **keep_table** (*str | None*) – If specified, the existing table will be renamed to this and will not be dropped

Return type
Table

transform_sql(*, *types=None*, *rename=None*, *drop=None*, *pk=<sqlite_utils.db.Default object>*, *not_null=None*, *defaults=None*, *drop_foreign_keys=None*, *add_foreign_keys=None*, *foreign_keys=None*, *column_order=None*, *tmp_suffix=None*, *keep_table=None*)

Return a list of SQL statements that should be executed in order to apply this transformation.

Parameters

- **types** (*dict | None*) – Columns that should have their type changed, for example `{"weight": float}`
- **rename** (*dict | None*) – Columns to rename, for example `{"headline": "title"}`
- **drop** (*Iterable | None*) – Columns to drop
- **pk** (*Any | None*) – New primary key for the table
- **not_null** (*Iterable[str] | None*) – Columns to set as `NOT NULL`
- **defaults** (*Dict[str, Any] | None*) – Default values for columns
- **drop_foreign_keys** (*Iterable | None*) – Foreign key constraints to remove - a column name drops any foreign key that column participates in, a tuple of column names drops the compound foreign key with exactly those columns
- **add_foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*) – List of foreign keys to add to the table

- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*) – List of foreign keys to set for the table, replacing any existing foreign keys
- **column_order** (*List[str] | None*) – List of strings specifying a full or partial column order to use when creating the table
- **tmp_suffix** (*str | None*) – Suffix to use for the temporary table name
- **keep_table** (*str | None*) – If specified, the existing table will be renamed to this and will not be dropped

Return type
List[str]

extract(*columns*, *table=None*, *fk_column=None*, *rename=None*)

Extract specified columns into a separate table.

See *Extracting columns into a separate table* for details.

Parameters

- **columns** (*str | Iterable[str]*) – Single column or list of columns that should be extracted
- **table** (*str | None*) – Name of table in which the new records should be created
- **fk_column** (*str | None*) – Name of the foreign key column to populate in the original table
- **rename** (*Dict[str, str] | None*) – Dictionary of columns that should be renamed when populating the new table

Return type
Table

create_index(*columns*, *index_name=None*, *unique=False*, *if_not_exists=False*, *find_unique_name=False*, *analyze=False*)

Create an index on this table.

Parameters

- **columns** (*Iterable[str | DescIndex]*) – A single columns or list of columns to index. These can be strings or, to create an index using the column in descending order, `db.DescIndex(column_name)` objects.
- **index_name** (*str | None*) – The name to use for the new index. Defaults to the column names joined on `_`.
- **unique** (*bool*) – Should the index be marked as unique, forcing unique values?
- **if_not_exists** (*bool*) – Only create the index if one with that name does not already exist.
- **find_unique_name** (*bool*) – If `index_name` is not provided and the automatically derived name already exists, keep incrementing a suffix number to find an available name.

- **analyze** (*bool*) – Run `ANALYZE` against this index after creating it.

See *Creating indexes*.

add_column(*col_name*, *col_type=None*, *fk=None*, *fk_col=None*, *not_null_default=None*)

Add a column to this table. See *Adding columns*.

Parameters

- **col_name** (*str*) – Name of the new column
- **col_type** (*Any | None*) – Column type - a Python type such as `str` or a SQLite type string such as `"BLOB"`
- **fk** (*str | None*) – Name of a table that this column should be a foreign key reference to
- **fk_col** (*str | None*) – Column in the foreign key table that this should reference
- **not_null_default** (*Any | None*) – Set this column to `not null` and give it this default value

drop(*ignore=False*)

Drop this table.

Parameters
ignore (*bool*) – Set to `True` to ignore the error if the table does not exist

Return type
None

guess_foreign_table(*column*)

For a given column, suggest another table that might be referenced by this column should it be used as a foreign key.

For example, a column called `tag_id` or `tag` or `tags` might suggest a `tag` table, if one exists.

If no candidates can be found, raises a `NoObviousTable` exception.

Parameters
column (*str*) – Name of column

Return type
str

add_foreign_key(*column*, *other_table=None*, *other_column=None*, *ignore=False*, *on_delete='NO ACTION'*, *on_update='NO ACTION'*)

Alter the schema to mark the specified column as a foreign key to another table.

Parameters

- **column** (*str | Tuple[str, ...] | List[str]*) – The column to mark as a foreign key - use a tuple of columns for a compound foreign key.
- **other_table** (*str | None*) – The table it refers to - if omitted, will be guessed based on the column name.
- **other_column** (*str | Tuple[str, ...] | List[str] | None*) – The column on the other table it - if omitted, will be guessed. Use a tuple of columns for a compound foreign key.
- **ignore** (*bool*) – Set this to `True` to ignore an existing foreign key - otherwise a `AlterError` will be raised.

- **on_delete** (*str*) – `ON DELETE` action for the foreign key, e.g. `"CASCADE"` or `"SET NULL"`.
- **on_update** (*str*) – `ON UPDATE` action for the foreign key.

enable_counts()

Set up triggers to update a cache of the count of rows in this table.

See *Cached table counts using triggers* for details.

Return type
None

property has_counts_triggers: bool

Does this table have triggers setup to update cached counts?

enable_fts(*columns*, *fts_version='FTS5'*, *create_triggers=False*, *tokenize=None*, *replace=False*)

Enable SQLite full-text search against the specified columns.

See *Full-text search* for more details.

Parameters

- **columns** (*Iterable[str]*) – List of column names to include in the search index.
- **fts_version** (*str*) – FTS version to use - defaults to `FTS5` but you may want `FTS4` for older SQLite versions.
- **create_triggers** (*bool*) – Should triggers be created to keep the search index up-to-date? Defaults to `False`.
- **tokenize** (*str | None*) – Custom SQLite tokenizer to use, for example `"porter"` to enable Porter stemming.
- **replace** (*bool*) – Should any existing FTS index for this table be replaced by the new one?

populate_fts(*columns*)

Update the associated SQLite full-text search index with the latest data from the table for the specified columns.

Parameters
columns (*Iterable[str]*) – Columns to populate the data for

Return type
Table

disable_fts()

Remove any full-text search index and related triggers configured for this table.

Return type
Table

rebuild_fts()

Run the `rebuild` operation against the associated full-text search index table.

Return type
Table

detect_fts()

Detect if table has a corresponding FTS virtual table and return it

Return type
str | None

`optimize`()

Run the `optimize` operation against the associated full-text search index table.

Return type
Table

`search_sql`(*columns=None*, *order_by=None*, *limit=None*, *offset=None*, *where=None*, *include_rank=False*)

" Return SQL string that can be used to execute searches against this table.

Parameters

- **columns** (*Iterable[str] | None*) – Columns to search against
- **order_by** (*str | None*) – Column or SQL expression to sort by
- **limit** (*int | None*) – SQL limit
- **offset** (*int | None*) – SQL offset
- **where** (*str | None*) – Extra SQL fragment for the WHERE clause
- **include_rank** (*bool*) – Select the search rank column in the final query

Return type
str

`search`(*q*, *order_by=None*, *columns=None*, *limit=None*, *offset=None*, *where=None*, *where_args=None*, *include_rank=False*, *quote=False*)

Execute a search against this table using SQLite full-text search, returning a sequence of dictionaries for each row.

Parameters

- **q** (*str*) – Terms to search for
- **order_by** (*str | None*) – Defaults to order by rank, or specify a column here.
- **columns** (*Iterable[str] | None*) – List of columns to return, defaults to all columns.
- **limit** (*int | None*) – Optional integer limit for returned rows.
- **offset** (*int | None*) – Optional integer SQL offset.
- **where** (*str | None*) – Extra SQL fragment for the WHERE clause
- **where_args** (*Iterable | dict | None*) – Arguments to use for :param placeholders in the extra WHERE clause
- **include_rank** (*bool*) – Select the search rank column in the final query
- **quote** (*bool*) – Apply quoting to disable any special characters in the search query

Return type
Generator[dict, None, None]

See *Searching with table.search()*.

`delete`(*pk_values*)

Delete row matching the specified primary key.

Parameters
pk_values (*list | tuple | str | int | float*) – A single value, or a tuple of values for tables that have a compound primary key

Return type
Table

delete_where(*where=None*, *where_args=None*, *analyze=False*)

Delete rows matching the specified where clause, or delete all rows in the table.

See *Deleting multiple records*.

Parameters

- **where** (*str | None*) – SQL where fragment to use, for example `id > ?`
- **where_args** (*Sequence | Dict[str, Any] | None*) – Parameters to use with that fragment - an iterable for `id > ?` parameters, or a dictionary for `id > :id`
- **analyze** (*bool*) – Set to `True` to run `ANALYZE` after the rows have been deleted.

Return type
Table

update(*pk_values*, *updates=None*, *alter=False*, *conversions=None*)

Execute a SQL `UPDATE` against the specified row.

See *Updating a specific record*.

Parameters

- **pk_values** (*list | tuple | str | int | float*) – The primary key of an individual record - can be a tuple if the table has a compound primary key.
- **updates** (*dict | None*) – A dictionary mapping columns to their updated values.
- **alter** (*bool*) – Set to `True` to add any missing columns.
- **conversions** (*dict | None*) – Optional dictionary of SQL functions to apply during the update, for example `{"mycolumn": "upper(?)"}`.

Return type
Table

convert(*columns*, *fn*, *output=None*, *output_type=None*, *drop=False*, *multi=False*, *where=None*, *where_args=None*, *show_progress=False*)

Apply conversion function `fn` to every value in the specified columns.

Parameters

- **columns** (*str | List[str]*) – A single column or list of string column names to convert.
- **fn** (*Callable*) – A callable that takes a single argument, `value`, and returns it converted.
- **output** (*str | None*) – Optional string column name to write the results to (defaults to the input column).
- **output_type** (*Any | None*) – If the output column needs to be created, this is the type that will be used for the new column.
- **drop** (*bool*) – Should the original column be dropped once the conversion is complete?
- **multi** (*bool*) – If `True` the return value of `fn(value)` will be expected to be a dictionary, and new columns will be created for each key of that dictionary.
- **where** (*str | None*) – SQL fragment to use as a `WHERE` clause to limit the rows to which the conversion is applied, for example `age > ?` or `age > :age`.

- **where_args** (*Sequence | Dict[str, Any] | None*) – List of arguments (if using ?) or a dictionary (if using `:age`).
- **show_progress** (*bool*) – Should a progress bar be displayed?

Return type
Table

See *Converting data in columns*.

insert(*record*, *pk=<sqlite_utils.db.Default object>*, *foreign_keys=<sqlite_utils.db.Default object>*, *column_order=<sqlite_utils.db.Default object>*, *not_null=<sqlite_utils.db.Default object>*, *defaults=<sqlite_utils.db.Default object>*, *hash_id=<sqlite_utils.db.Default object>*, *hash_id_columns=<sqlite_utils.db.Default object>*, *alter=<sqlite_utils.db.Default object>*, *ignore=<sqlite_utils.db.Default object>*, *replace=<sqlite_utils.db.Default object>*, *extracts=<sqlite_utils.db.Default object>*, *conversions=<sqlite_utils.db.Default object>*, *columns=<sqlite_utils.db.Default object>*, *strict=<sqlite_utils.db.Default object>*)

Insert a single record into the table. The table will be created with a schema that matches the inserted record if it does not already exist, see *Creating tables*.

- `record` - required: a dictionary representing the record to be inserted.

The other parameters are optional, and mostly influence how the new table will be created if that table does not exist yet.

Each of them defaults to `DEFAULT`, which indicates that the default setting for the current `Table` object (specified in the table constructor) should be used.

Parameters

- **record** (*Dict[str, Any]*) – Dictionary record to be inserted
- **pk** – If creating the table, which column should be the primary key.
- **foreign_keys** – See *Specifying foreign keys*.
- **column_order** (*List[str] | Default | None*) – List of strings specifying a full or partial column order to use when creating the table.
- **not_null** (*Iterable[str] | Default | None*) – Set of strings specifying columns that should be `NOT NULL`.
- **defaults** (*Dict[str, Any] | Default | None*) – Dictionary specifying default values for specific columns.
- **hash_id** (*str | Default | None*) – Name of a column to create and use as a primary key, where the value of that primary key will be derived as a SHA1 hash of the other column values in the record. `hash_id="id"` is a common column name used for this.
- **alter** (*bool | Default | None*) – Boolean, should any missing columns be added automatically?
- **ignore** (*bool | Default | None*) – Boolean, if a record already exists with this primary key, ignore this insert.
- **replace** (*bool | Default | None*) – Boolean, if a record already exists with this primary key, replace it with this new record.
- **extracts** (*Dict[str, str] | List[str] | Default | None*) – A list of columns to extract to other tables, or a dictionary that maps `{column_name: other_table_name}`. See *Populating lookup tables automatically during insert/upsert*.

- **conversions** (*Dict[str, str] | Default | None*) – Dictionary specifying SQL conversion functions to be applied to the data while it is being inserted, for example `{"name": "upper(?)"}`. See *Converting column values using SQL functions*.
- **columns** (*Dict[str, Any] | Default | None*) – Dictionary over-riding the detected types used for the columns, for example `{"age": int, "weight": float}`.
- **strict** (*bool | Default | None*) – Boolean, apply STRICT mode if creating the table.
- **hash_id_columns** (*Iterable[str] | Default | None*)

Return type
Table

insert_all(*records*, *pk=<sqlite_utils.db.Default object>*, *foreign_keys=<sqlite_utils.db.Default object>*, *column_order=<sqlite_utils.db.Default object>*, *not_null=<sqlite_utils.db.Default object>*, *defaults=<sqlite_utils.db.Default object>*, *batch_size=<sqlite_utils.db.Default object>*, *hash_id=<sqlite_utils.db.Default object>*, *hash_id_columns=<sqlite_utils.db.Default object>*, *alter=<sqlite_utils.db.Default object>*, *ignore=<sqlite_utils.db.Default object>*, *replace=<sqlite_utils.db.Default object>*, *truncate=False*, *extracts=<sqlite_utils.db.Default object>*, *conversions=<sqlite_utils.db.Default object>*, *columns=<sqlite_utils.db.Default object>*, *upsert=False*, *analyze=False*, *strict=<sqlite_utils.db.Default object>*)

Like `.insert()` but takes a list of records and ensures that the table that it creates (if table does not exist) has columns for ALL of that data.

Use `analyze=True` to run `ANALYZE` after the insert has completed.

Parameters
records (*Iterable[Dict[str, Any]] | Iterable[Sequence[Any]]*)

Return type
Table

upsert(*record*, *pk=<sqlite_utils.db.Default object>*, *foreign_keys=<sqlite_utils.db.Default object>*, *column_order=<sqlite_utils.db.Default object>*, *not_null=<sqlite_utils.db.Default object>*, *defaults=<sqlite_utils.db.Default object>*, *hash_id=<sqlite_utils.db.Default object>*, *hash_id_columns=<sqlite_utils.db.Default object>*, *alter=<sqlite_utils.db.Default object>*, *extracts=<sqlite_utils.db.Default object>*, *conversions=<sqlite_utils.db.Default object>*, *columns=<sqlite_utils.db.Default object>*, *strict=<sqlite_utils.db.Default object>*)

Like `.insert()` but performs an `UPSERT`, where records are inserted if they do not exist and updated if they DO exist, based on matching against their primary key.

See *Upserting data*.

Return type
Table

upsert_all(*records*, *pk=<sqlite_utils.db.Default object>*, *foreign_keys=<sqlite_utils.db.Default object>*, *column_order=<sqlite_utils.db.Default object>*, *not_null=<sqlite_utils.db.Default object>*, *defaults=<sqlite_utils.db.Default object>*, *batch_size=<sqlite_utils.db.Default object>*, *hash_id=<sqlite_utils.db.Default object>*, *hash_id_columns=<sqlite_utils.db.Default object>*, *alter=<sqlite_utils.db.Default object>*, *extracts=<sqlite_utils.db.Default object>*, *conversions=<sqlite_utils.db.Default object>*, *columns=<sqlite_utils.db.Default object>*, *analyze=False*, *strict=<sqlite_utils.db.Default object>*)

Like `.upsert()` but can be applied to a list of records.

Parameters
records (*Iterable[Dict[str, Any]] | Iterable[Sequence[Any]]*)

Return type
Table

lookup(*lookup_values*, *extra_values=None*, *pk='id'*, *foreign_keys=None*, *column_order=None*, *not_null=None*, *defaults=None*, *extracts=None*, *conversions=None*, *columns=None*, *strict=False*)

Create or populate a lookup table with the specified values.

`db["Species"].lookup({"name": "Palm"})` will create a table called `Species` (if one does not already exist) with two columns: `id` and `name`. It will set up a unique constraint on the `name` column to guarantee it will not contain duplicate rows.

It will then insert a new row with the `name` set to `Palm` and return the new integer primary key value.

An optional second argument can be provided with more `name: value` pairs to be included only if the record is being created for the first time. These will be ignored on subsequent lookup calls for records that already exist.

All other keyword arguments are passed through to `.insert()`.

See *Working with lookup tables* for more details.

Parameters

- **lookup_values** (*Dict[str, Any]*) – Dictionary specifying column names and values to use for the lookup
- **extra_values** (*Dict[str, Any] | None*) – Additional column values to be used only if creating a new record
- **strict** (*bool | None*) – Boolean, apply STRICT mode if creating the table.
- **pk** (*str | None*)
- **foreign_keys** (*Iterable[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | List[str | ForeignKey | Tuple[str | Tuple[str, ...] | List[str], str] | Tuple[str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]] | Tuple[str, str | Tuple[str, ...] | List[str], str, str | Tuple[str, ...] | List[str]]] | None*)
- **column_order** (*List[str] | None*)
- **not_null** (*Iterable[str] | None*)
- **defaults** (*Dict[str, Any] | None*)
- **extracts** (*Dict[str, str] | List[str] | None*)
- **conversions** (*Dict[str, str] | None*)
- **columns** (*Dict[str, Any] | None*)

m2m(*other_table*, *record_or_iterable=None*, *pk=<sqlite_utils.db.Default object>*, *lookup=None*, *m2m_table=None*, *alter=False*)

After inserting a record in a table, create one or more records in some other table and then create many-to-many records linking the original record and the newly created records together.

For example:

```
db["dogs"].insert({"id": 1, "name": "Cleo"}, pk="id").m2m(
    "humans", {"id": 1, "name": "Natalie"}, pk="id"
)
```

See *Working with many-to-many relationships* for details.

Parameters

- **other_table** (*str | Table*) – The name of the table to insert the new records into.
- **record_or_iterable** (*Iterable[Dict[str, Any]] | Dict[str, Any] | None*) – A single dictionary record to insert, or a list of records.
- **pk** (*Any | Default | None*) – The primary key to use if creating `other_table`.
- **lookup** (*Dict[str, Any] | None*) – Same dictionary as for `.lookup()`, to create a many-to-many lookup table.
- **m2m_table** (*str | None*) – The string name to use for the many-to-many table, defaults to creating this automatically based on the names of the two tables.
- **alter** (*bool*) – Set to `True` to add any missing columns on `other_table` if that table already exists.

analyze()

Run ANALYZE against this table

Return type
None

analyze_column(*column*, *common_limit=10*, *value_truncate=None*, *total_rows=None*, *most_common=True*, *least_common=True*)

Return statistics about the specified column.

See *Analyzing a column*.

Parameters

- **column** (*str*) – Column to analyze
- **common_limit** (*int*) – Show this many column values
- **value_truncate** – Truncate display of common values to this many characters
- **total_rows** – Optimization - pass the total number of rows in the table to save running a fresh `count(*)` query
- **most_common** (*bool*) – If `True`, calculate the most common values
- **least_common** (*bool*) – If `True`, calculate the least common values

Return type
ColumnDetails

add_geometry_column(*column_name*, *geometry_type*, *srid=4326*, *coord_dimension='XY'*, *not_null=False*)

In SpatiaLite, a geometry column can only be added to an existing table. To do so, use `table.add_geometry_column`, passing in a geometry type.

By default, this will add a nullable column using SRID 4326. This can be customized using the `column_name`, `srid` and `not_null` arguments.

Returns `True` if the column was successfully added, `False` if not.

```
from sqlite_utils.db import Database
from sqlite_utils.utils import find_spatialite

db = Database("mydb.db")
db.init_spatialite(find_spatialite())

# the table must exist before adding a geometry column
table = db["locations"].create({"name": str})
table.add_geometry_column("geometry", "POINT")
```

Parameters

- **column_name** (*str*) – Name of column to add
- **geometry_type** (*str*) – Type of geometry column, for example `"GEOMETRY"` or `"POINT"` or ``"POLYGON"`
- **srid** (*int*) – Integer SRID, defaults to 4326 for WGS84
- **coord_dimension** (*str*) – Dimensions to use, defaults to `"XY"` - set to `"XYZ"` to work in three dimensions
- **not_null** (*bool*) – Should the column be `NOT NULL`

Return type
bool

create_spatial_index(*column_name*)

A spatial index allows for significantly faster bounding box queries. To create one, use `create_spatial_index` with the name of an existing geometry column.

Returns `True` if the index was successfully created, `False` if not. Calling this function if an index already exists is a no-op.

```
# assuming SpatiaLite is loaded, create the table, add the column
table = db["locations"].create({"name": str})
table.add_geometry_column("geometry", "POINT")

# now we can index it
table.create_spatial_index("geometry")

# the spatial index is a virtual table, which we can inspect
print(db["idx_locations_geometry"].schema)
# outputs:
# CREATE VIRTUAL TABLE "idx_locations_geometry" USING rtree(pkid, xmin, xmax,␣
↪ymin, ymax)
```

Parameters
column_name – Geometry column to create the spatial index against

Return type
bool

1.6.4 sqlite_utils.db.View

class sqlite_utils.db.**View**(*db*, *name*)

Bases: *Queryable*

Parameters

- **db** (Database)
- **name** (*str*)

exists()

Does this table or view exist yet?

Return type
bool

drop(*ignore=False*)

Drop this view.

Parameters
ignore (*bool*) – Set to `True` to ignore the error if the view does not exist

Return type
None

1.6.5 Other

sqlite_utils.db.Column

class sqlite_utils.db.**Column**(*cid*, *name*, *type*, *notnull*, *default_value*, *is_pk*)

Describes a SQLite column returned by the `Table.columns` property.

cid
Column index

name
Column name

type
Column type

notnull
Does the column have a `not null` constraint

default_value
Default value for this column

is_pk
Is this column part of the primary key

sqlite_utils.db.ColumnDetails

class sqlite_utils.db.**ColumnDetails**(*table*, *column*, *total_rows*, *num_null*, *num_blank*, *num_distinct*, *most_common*, *least_common*)

Summary information about a column, see *Analyzing a column*.

table
The name of the table

column
The name of the column

total_rows
The total number of rows in the table

num_null
The number of rows for which this column is null

num_blank
The number of rows for which this column is blank (the empty string)

num_distinct
The number of distinct values in this column

most_common
The `N` most common values as a list of `(value, count)` tuples, or `None` if the table consists entirely of distinct values

least_common
The `N` least common values as a list of `(value, count)` tuples, or `None` if the table is entirely distinct or if the number of distinct values is less than N (since they will already have been returned in `most_common`)

sqlite_utils.db.ForeignKey

class sqlite_utils.db.**ForeignKey**(*table*, *column*, *other_table*, *other_column*, *columns=()*, *other_columns=()*, *is_compound=False*, *on_delete='NO ACTION'*, *on_update='NO ACTION'*)

A foreign key defined on a table.

For single-column foreign keys `column` and `other_column` hold the column names, and `columns`/`other_columns` are one-item tuples.

For compound (multi-column) foreign keys `column` and `other_column` are `None` - use `columns` and `other_columns` instead, and check `is_compound`.

`on_delete` and `on_update` hold the foreign key actions, e.g. `"CASCADE"` - `"NO ACTION"` if not set.

Instances are immutable and hashable, so they can be collected into sets and used as dictionary keys. Equality covers every compared field, including `on_delete` and `on_update` - two foreign keys differing only in their actions are different constraints.

Prior to sqlite-utils 4.0 this was a `namedtuple` and could be unpacked or indexed as `(table, column, other_table, other_column)`. It is now a dataclass - access its fields by name instead.

Parameters

- **table** (*str*)
- **column** (*str | None*)
- **other_table** (*str*)
- **other_column** (*str | None*)
- **columns** (*Tuple[str, ...]*)
- **other_columns** (*Tuple[str, ...]*)
- **is_compound** (*bool*)
- **on_delete** (*str*)
- **on_update** (*str*)

1.6.6 sqlite_utils.utils

sqlite_utils.utils.hash_record

sqlite_utils.utils.**hash_record**(*record*, *keys=None*)

`record` should be a Python dictionary. Returns a sha1 hash of the keys and values in that record.

If `keys=` is provided, uses just those keys to generate the hash.

Example usage:

```
from sqlite_utils.utils import hash_record

hashed = hash_record({"name": "Cleo", "twitter": "CleoPaws"})
# Or with the keys= option:
hashed = hash_record(
    {"name": "Cleo", "twitter": "CleoPaws", "age": 7},
    keys=("name", "twitter")
)
```

Parameters

- **record** (*Dict[str, Any]*) – Record to generate a hash for
- **keys** (*Iterable[str] | None*) – Subset of keys to use for that hash

Return type

str

sqlite_utils.utils.rows_from_file

sqlite_utils.utils.**rows_from_file**(*fp*, *format=None*, *dialect=None*, *encoding=None*, *ignore_extras=False*, *extras_key=None*)

Load a sequence of dictionaries from a file-like object containing one of four different formats.

```
from sqlite_utils.utils import rows_from_file
import io

rows, format = rows_from_file(io.StringIO("id,name\n1,Cleo"))
print(list(rows), format)
# Outputs [{'id': '1', 'name': 'Cleo'}] Format.CSV
```

This defaults to attempting to automatically detect the format of the data, or you can pass in an explicit format using the format= option.

Returns a tuple of `(rows_generator, format_used)` where `rows_generator` can be iterated over to return dictionaries, while `format_used` is a value from the `sqlite_utils.utils.Format` enum:

```
class Format(enum.Enum):
    CSV = 1
    TSV = 2
    JSON = 3
    NL = 4
```

If a CSV or TSV file includes rows with more fields than are declared in the header a `sqlite_utils.utils.RowError` exception will be raised when you loop over the generator.

You can instead ignore the extra data by passing `ignore_extras=True`.

Or pass `extras_key="rest"` to put those additional values in a list in a key called `rest`.

Parameters

- **fp** (*BinaryIO*) – a file-like object containing binary data
- **format** (*Format | None*) – the format to use - omit this to detect the format
- **dialect** (*Type[Dialect] | None*) – the CSV dialect to use - omit this to detect the dialect
- **encoding** (*str | None*) – the character encoding to use when reading CSV/TSV data
- **ignore_extras** (*bool | None*) – ignore any extra fields on rows
- **extras_key** (*str | None*) – put any extra fields in a list with this key

Return type

Tuple[*Iterable*[*Dict*[str, None | int | float | str | bytes | bool | *List*[str]]], *Format*]

sqlite_utils.utils.TypeTracker

class sqlite_utils.utils.**TypeTracker**

Wrap an iterator of dictionaries and keep track of which SQLite column types are the most likely fit for each of their keys.

Example usage:

```
from sqlite_utils.utils import TypeTracker
import sqlite_utils

db = sqlite_utils.Database(memory=True)
tracker = TypeTracker()
rows = [{"id": "1", "name": "Cleo", "id": "2", "name": "Cardi"}]
db["creatures"].insert_all(tracker.wrap(rows))
print(tracker.types)
# Outputs {'id': 'integer', 'name': 'text'}
db["creatures"].transform(types=tracker.types)
```

wrap(*iterator*)

Use this to loop through an existing iterator, tracking the column types as part of the iteration.

Parameters

iterator (*Iterable[Dict[str, Any]]*) – The iterator to wrap

Return type

Iterable[*Dict*[str, *Any*]]

property types: Dict[str, str]

A dictionary mapping column names to their detected types. This can be passed to the `db[table_name].transform(types=tracker.types)` method.

sqlite_utils.utils.chunks

sqlite_utils.utils.**chunks**(*sequence*, *size*)

Iterate over chunks of the sequence of the given size.

Parameters

- **sequence** (*Iterable[T]*) – Any Python iterator
- **size** (*int*) – The size of each chunk

Return type
Iterable[*Iterable*[*T*]]

sqlite_utils.utils.flatten

sqlite_utils.utils.**flatten**(*row*)

Turn a nested dict e.g. `{"a": {"b": 1}}` into a flat dict: `{"a_b": 1}`

Parameters
row (*Dict[str, Any]*) – A Python dictionary, optionally with nested dictionaries

Return type
Dict[str, *Any*]

1.7 CLI reference

This page lists the `--help` for every `sqlite-utils` CLI sub-command.

1.7.1 query

See *Running SQL queries*.

```
Usage: sqlite-utils query [OPTIONS] PATH SQL

  Execute SQL query and return the results as JSON

  Example:

      sqlite-utils data.db \
          "select * from chickens where age > :age" \
```

(continues on next page)

(continued from previous page)

```
          -p age 1

  Pass "-" as the SQL to read the query from standard input:

      echo "select * from chickens" | sqlite-utils data.db -

Options:
  --attach <TEXT FILE>...     Additional databases to attach - specify alias and
                              filepath
  --nl                        Output newline-delimited JSON
  --arrays                    Output rows as arrays instead of objects
  --csv                       Output CSV
  --tsv                       Output TSV
  --no-headers                Omit headers from CSV/TSV and table/--fmt output
  -t, --table                 Output as a formatted table
  --fmt TEXT                  Table format - one of asciidoc, colon_grid,
                              double_grid, double_outline, fancy_grid,
                              fancy_outline, github, grid, heavy_grid,
                              heavy_outline, html, jira, latex, latex_booktabs,
                              latex_longtable, latex_raw, mediawiki, mixed_grid,
                              mixed_outline, moinmoin, orgtbl, outline, pipe,
                              plain, presto, pretty, psql, rounded_grid,
                              rounded_outline, rst, simple, simple_grid,
                              simple_outline, textile, tsv, unsafehtml, youtrack
  --json-cols                 Detect JSON cols and output them as JSON, not
                              escaped strings
  --ascii                     Escape non-ASCII characters in JSON output as
                              \uXXXX
  -r, --raw                   Raw output, first column of first row
  --raw-lines                 Raw output, first column of each row
  -p, --param <TEXT TEXT>...  Named :parameters for SQL query
  --functions TEXT            Python code or a file path defining custom SQL
                              functions; can be used multiple times
  --load-extension TEXT       Path to SQLite extension, with optional
                              :entrypoint
  -h, --help                  Show this message and exit.
```

1.7.2 memory

See *Querying data directly using an in-memory database*.

```
Usage: sqlite-utils memory [OPTIONS] [PATHS]... SQL

  Execute SQL query against an in-memory database, optionally populated by
  imported data

  To import data from CSV, TSV or JSON files pass them on the command-line:

      sqlite-utils memory one.csv two.json \
          "select * from one join two on one.two_id = two.id"

  For data piped into the tool from standard input, use "-" or "stdin":
```

(continues on next page)

(continued from previous page)

```
    cat animals.csv | sqlite-utils memory - \
        "select * from stdin where species = 'dog'"

  The format of the data will be automatically detected. You can specify the
  format explicitly using :json, :csv, :tsv or :nl (for newline-delimited JSON)
  - for example:

    cat animals.csv | sqlite-utils memory stdin:csv places.dat:nl \
        "select * from stdin where place_id in (select id from places)"

  Use --schema to view the SQL schema of any imported files:

    sqlite-utils memory animals.csv --schema

Options:
  --functions TEXT            Python code or a file path defining custom SQL
                              functions; can be used multiple times
  --attach <TEXT FILE>...     Additional databases to attach - specify alias and
                              filepath
  --flatten                   Flatten nested JSON objects, so {"foo": {"bar":
                              1}} becomes {"foo_bar": 1}
  --nl                        Output newline-delimited JSON
  --arrays                    Output rows as arrays instead of objects
  --csv                       Output CSV
  --tsv                       Output TSV
  --no-headers                Omit headers from CSV/TSV and table/--fmt output
  -t, --table                 Output as a formatted table
  --fmt TEXT                  Table format - one of asciidoc, colon_grid,
                              double_grid, double_outline, fancy_grid,
                              fancy_outline, github, grid, heavy_grid,
                              heavy_outline, html, jira, latex, latex_booktabs,
                              latex_longtable, latex_raw, mediawiki, mixed_grid,
                              mixed_outline, moinmoin, orgtbl, outline, pipe,
                              plain, presto, pretty, psql, rounded_grid,
                              rounded_outline, rst, simple, simple_grid,
                              simple_outline, textile, tsv, unsafehtml, youtrack
  --json-cols                 Detect JSON cols and output them as JSON, not
                              escaped strings
  --ascii                     Escape non-ASCII characters in JSON output as
                              \uXXXX
  -r, --raw                   Raw output, first column of first row
  --raw-lines                 Raw output, first column of each row
  -p, --param <TEXT TEXT>...  Named :parameters for SQL query
  --encoding TEXT             Character encoding for CSV input, defaults to
                              utf-8
  -n, --no-detect-types       Treat all CSV/TSV columns as TEXT
  --schema                    Show SQL schema for in-memory database
  --dump                      Dump SQL for in-memory database
  --save FILE                 Save in-memory database to this file
  --analyze                   Analyze resulting tables and output results
  --load-extension TEXT       Path to SQLite extension, with optional
```

(continues on next page)

(continued from previous page)

```
                              :entrypoint
  -h, --help                  Show this message and exit.
```

1.7.3 insert

See *Inserting JSON data*, *Inserting CSV or TSV data*, *Inserting unstructured data with --lines and --text*, *Applying conversions while inserting data*.

```
Usage: sqlite-utils insert [OPTIONS] PATH TABLE [FILE]

  Insert records from FILE into a table, creating the table if it does not
  already exist.

  Example:

      echo '{"name": "Lila"}' | sqlite-utils insert data.db chickens -

  By default the input is expected to be a JSON object or array of objects.

  - Use --nl for newline-delimited JSON objects
  - Use --csv or --tsv for comma-separated or tab-separated input
  - Use --lines to write each incoming line to a column called "line"
  - Use --text to write the entire input to a column called "text"

  You can also use --convert to pass a fragment of Python code that will be used
  to convert each input.

  Your Python code will be passed a "row" variable representing the imported
  row, and can return a modified row.

  This example uses just the name, latitude and longitude columns from a CSV
  file, converting name to upper case and latitude and longitude to floating
  point numbers:

      sqlite-utils insert plants.db plants plants.csv --csv --convert '
        return {
          "name": row["name"].upper(),
          "latitude": float(row["latitude"]),
          "longitude": float(row["longitude"]),
        }'

  If you are using --lines your code will be passed a "line" variable, and for
  --text a "text" variable.

  When using --text your function can return an iterator of rows to insert. This
  example inserts one record per word in the input:

      echo 'A bunch of words' | sqlite-utils insert words.db words - \
        --text --convert '({"word": w} for w in text.split())'

  Instead of a FILE you can use --code to provide a block of Python code that
  defines the rows to insert, as either a rows() function that yields
```

(continues on next page)

(continued from previous page)

```
  dictionaries or a "rows" iterable. --code can also be a path to a .py file:

      sqlite-utils insert data.db creatures --code '
      def rows():
          yield {"id": 1, "name": "Cleo"}
          yield {"id": 2, "name": "Suna"}
      ' --pk id

Options:
  --pk TEXT                 Columns to use as the primary key, e.g. id
  --code TEXT               Python code defining a rows() function or iterable
                            of rows to insert
  --flatten                 Flatten nested JSON objects, so {"a": {"b": 1}}
                            becomes {"a_b": 1}
  --nl                      Expect newline-delimited JSON
  -c, --csv                 Expect CSV input
  --tsv                     Expect TSV input
  --empty-null              Treat empty strings as NULL
  --lines                   Treat each line as a single value called 'line'
  --text                    Treat input as a single value called 'text'
  --convert TEXT            Python code to convert each item
  --import TEXT             Python modules to import
  --delimiter TEXT          Delimiter to use for CSV files
  --quotechar TEXT          Quote character to use for CSV/TSV
  --sniff                   Detect delimiter and quote character
  --no-headers              CSV file has no header row
  --encoding TEXT           Character encoding for input, defaults to utf-8
  --batch-size INTEGER      Commit every X records
  --stop-after INTEGER      Stop after X records
  --alter                   Alter existing table to add any missing columns
  --not-null TEXT           Columns that should be created as NOT NULL
  --default <TEXT TEXT>...  Default value that should be set for a column
  --no-detect-types         Treat all CSV/TSV columns as TEXT
  --analyze                 Run ANALYZE at the end of this operation
  --load-extension TEXT     Path to SQLite extension, with optional :entrypoint
  --silent                  Do not show progress bar
  --strict                  Apply STRICT mode to created table
  --ignore                  Ignore records if pk already exists
  --replace                 Replace records if pk already exists
  --truncate                Truncate table before inserting records, if table
                            already exists
  -h, --help                Show this message and exit.
```

1.7.4 upsert

See *Upserting data*.

```
Usage: sqlite-utils upsert [OPTIONS] PATH TABLE [FILE]

  Upsert records based on their primary key. Works like 'insert' but if an
  incoming record has a primary key that matches an existing record the existing
  record will be updated.
```

(continues on next page)

(continued from previous page)

```
  If the table already exists and has a primary key, --pk can be omitted.

  Example:

      echo '[
          {"id": 1, "name": "Lila"},
          {"id": 2, "name": "Suna"}
      ]' | sqlite-utils upsert data.db chickens - --pk id

Options:
  --pk TEXT                  Columns to use as the primary key, e.g. id
  --code TEXT                Python code defining a rows() function or iterable
                             of rows to insert
  --flatten                  Flatten nested JSON objects, so {"a": {"b": 1}}
                             becomes {"a_b": 1}
  --nl                       Expect newline-delimited JSON
  -c, --csv                  Expect CSV input
  --tsv                      Expect TSV input
  --empty-null               Treat empty strings as NULL
  --lines                    Treat each line as a single value called 'line'
  --text                     Treat input as a single value called 'text'
  --convert TEXT             Python code to convert each item
  --import TEXT              Python modules to import
  --delimiter TEXT           Delimiter to use for CSV files
  --quotechar TEXT           Quote character to use for CSV/TSV
  --sniff                    Detect delimiter and quote character
  --no-headers               CSV file has no header row
  --encoding TEXT            Character encoding for input, defaults to utf-8
  --batch-size INTEGER       Commit every X records
  --stop-after INTEGER       Stop after X records
  --alter                    Alter existing table to add any missing columns
  --not-null TEXT            Columns that should be created as NOT NULL
  --default <TEXT TEXT>...   Default value that should be set for a column
  --no-detect-types          Treat all CSV/TSV columns as TEXT
  --analyze                  Run ANALYZE at the end of this operation
  --load-extension TEXT      Path to SQLite extension, with optional :entrypoint
  --silent                   Do not show progress bar
  --strict                   Apply STRICT mode to created table
  -h, --help                 Show this message and exit.
```

1.7.5 bulk

See *Executing SQL in bulk*.

```
Usage: sqlite-utils bulk [OPTIONS] PATH SQL FILE

  Execute parameterized SQL against the provided list of documents.

  Example:

      echo '[
```

(continues on next page)

(continued from previous page)

```
          {"id": 1, "name": "Lila2"},
          {"id": 2, "name": "Suna2"}
      ]' | sqlite-utils bulk data.db '
          update chickens set name = :name where id = :id
      ' _

Options:
  --batch-size INTEGER   Commit every X records
  --functions TEXT       Python code or a file path defining custom SQL
                         functions; can be used multiple times
  --flatten              Flatten nested JSON objects, so {"a": {"b": 1}} becomes
                         {"a_b": 1}
  --nl                   Expect newline-delimited JSON
  -c, --csv              Expect CSV input
  --tsv                  Expect TSV input
  --empty-null           Treat empty strings as NULL
  --lines                Treat each line as a single value called 'line'
  --text                 Treat input as a single value called 'text'
  --convert TEXT         Python code to convert each item
  --import TEXT          Python modules to import
  --delimiter TEXT       Delimiter to use for CSV files
  --quotechar TEXT       Quote character to use for CSV/TSV
  --sniff                Detect delimiter and quote character
  --no-headers           CSV file has no header row
  --encoding TEXT        Character encoding for input, defaults to utf-8
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.6 search

See *Executing searches*.

```
Usage: sqlite-utils search [OPTIONS] PATH DBTABLE Q

  Execute a full-text search against this table

  Example:

      sqlite-utils search data.db chickens lila

Options:
  -o, --order TEXT       Order by ('column' or 'column desc')
  -c, --column TEXT      Columns to return
  --limit INTEGER        Number of rows to return - defaults to everything
  --sql                  Show SQL query that would be run
  --quote                Apply FTS quoting rules to search term
  --nl                   Output newline-delimited JSON
  --arrays               Output rows as arrays instead of objects
  --csv                  Output CSV
  --tsv                  Output TSV
  --no-headers           Omit headers from CSV/TSV and table/--fmt output
  -t, --table            Output as a formatted table
```

(continues on next page)

(continued from previous page)

```
  --fmt TEXT             Table format - one of asciidoc, colon_grid,
                         double_grid, double_outline, fancy_grid, fancy_outline,
                         github, grid, heavy_grid, heavy_outline, html, jira,
                         latex, latex_booktabs, latex_longtable, latex_raw,
                         mediawiki, mixed_grid, mixed_outline, moinmoin, orgtbl,
                         outline, pipe, plain, presto, pretty, psql,
                         rounded_grid, rounded_outline, rst, simple,
                         simple_grid, simple_outline, textile, tsv, unsafehtml,
                         youtrack
  --json-cols            Detect JSON cols and output them as JSON, not escaped
                         strings
  --ascii                Escape non-ASCII characters in JSON output as \uXXXX
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.7 transform

See *Transforming tables*.

```
Usage: sqlite-utils transform [OPTIONS] PATH TABLE

  Transform a table beyond the capabilities of ALTER TABLE

  Example:

      sqlite-utils transform mydb.db mytable \
          --drop column1 \
          --rename column2 column_renamed

Options:
  --type <TEXT CHOICE>...         Change column type to INTEGER, TEXT, FLOAT,
                                  REAL or BLOB
  --drop TEXT                     Drop this column
  --rename <TEXT TEXT>...         Rename this column to X
  -o, --column-order TEXT         Reorder columns
  --not-null TEXT                 Set this column to NOT NULL
  --not-null-false TEXT           Remove NOT NULL from this column
  --pk TEXT                       Make this column the primary key
  --pk-none                       Remove primary key (convert to rowid table)
  --default <TEXT TEXT>...        Set default value for this column
  --default-none TEXT             Remove default from this column
  --add-foreign-key <TEXT TEXT TEXT>...
                                  Add a foreign key constraint from a column to
                                  another table with another column
  --drop-foreign-key TEXT         Drop foreign key constraint for this column
  --sql                           Output SQL without executing it
  --load-extension TEXT           Path to SQLite extension, with optional
                                  :entrypoint
  -h, --help                      Show this message and exit.
```

1.7.8 extract

See *Extracting columns into a separate table*.

```
Usage: sqlite-utils extract [OPTIONS] PATH TABLE COLUMNS...

  Extract one or more columns into a separate table

  Example:

      sqlite-utils extract trees.db Street_Trees species

Options:
  --table TEXT             Name of the other table to extract columns to
  --fk-column TEXT         Name of the foreign key column to add to the table
  --rename <TEXT TEXT>...  Rename this column in extracted table
  --load-extension TEXT    Path to SQLite extension, with optional :entrypoint
  -h, --help               Show this message and exit.
```

1.7.9 schema

See *Showing the schema*.

```
Usage: sqlite-utils schema [OPTIONS] PATH [TABLES]...

  Show full schema for this database or for specified tables

  Example:

      sqlite-utils schema trees.db

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.10 insert-files

See *Inserting data from files*.

```
Usage: sqlite-utils insert-files [OPTIONS] PATH TABLE FILE_OR_DIR...

  Insert one or more files using BLOB columns in the specified table

  Example:

      sqlite-utils insert-files pics.db images *.gif \
          -c name:name \
          -c content:content \
          -c content_hash:sha256 \
          -c created:ctime_iso \
          -c modified:mtime_iso \
          -c size:size \
          --pk name
```

(continues on next page)

(continued from previous page)

```
Options:
  -c, --column TEXT      Column definitions for the table
  --pk TEXT              Column to use as primary key
  --alter                Alter table to add missing columns
  --replace              Replace files with matching primary key
  --upsert               Upsert files with matching primary key
  --name TEXT            File name to use
  --text                 Store file content as TEXT, not BLOB
  --encoding TEXT        Character encoding for input, defaults to utf-8
  -s, --silent           Don't show a progress bar
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.11 analyze-tables

See *Analyzing tables*.

```
Usage: sqlite-utils analyze-tables [OPTIONS] PATH [TABLES]...

  Analyze the columns in one or more tables

  Example:

      sqlite-utils analyze-tables data.db trees

Options:
  -c, --column TEXT       Specific columns to analyze
  --save                  Save results to _analyze_tables table
  --common-limit INTEGER  How many common values
  --no-most               Skip most common values
  --no-least              Skip least common values
  --load-extension TEXT   Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.12 convert

See *Converting data in columns*.

```
Usage: sqlite-utils convert [OPTIONS] DB_PATH TABLE COLUMNS... CODE

  Convert columns using Python code you supply. For example:

  sqlite-utils convert my.db mytable mycolumn \
      '"\n".join(textwrap.wrap(value, 10))' \
      --import=textwrap

  "value" is a variable with the column value to be converted.

  CODE can also be a reference to a callable that takes the value, for example:

  sqlite-utils convert my.db mytable date r.parsedate
  sqlite-utils convert my.db mytable data json.loads --import json
```

(continues on next page)

(continued from previous page)

```
  Use "-" for CODE to read Python code from standard input.

  The following common operations are available as recipe functions:

  r.jsonsplit(value: 'str', delimiter: 'str' = ',', type: 'Callable[[str],
  object]' = <class 'str'>) -> 'str'

  Convert a string like a,b,c into a JSON array ["a", "b", "c"]

  r.parsedate(value: 'str', dayfirst: 'bool' = False, yearfirst: 'bool' = False,
  errors: 'Optional[object]' = None) -> 'Optional[str]'

  Parse a date and convert it to ISO date format: yyyy-mm-dd
  - dayfirst=True: treat xx as the day in xx/yy/zz
  - yearfirst=True: treat xx as the year in xx/yy/zz
  - errors=r.IGNORE to ignore values that cannot be parsed
  - errors=r.SET_NULL to set values that cannot be parsed to null

  r.parsedatetime(value: 'str', dayfirst: 'bool' = False, yearfirst: 'bool' =
  False, errors: 'Optional[object]' = None) -> 'Optional[str]'

  Parse a datetime and convert it to ISO datetime format: yyyy-mm-ddTHH:MM:SS
  - dayfirst=True: treat xx as the day in xx/yy/zz
  - yearfirst=True: treat xx as the year in xx/yy/zz
  - errors=r.IGNORE to ignore values that cannot be parsed
  - errors=r.SET_NULL to set values that cannot be parsed to null

  You can use these recipes like so:

  sqlite-utils convert my.db mytable mycolumn \
      'r.jsonsplit(value, delimiter=":")'

Options:
  --import TEXT                   Python modules to import
  --dry-run                       Show results of running this against first 10
                                  rows
  --multi                         Populate columns for keys in returned
                                  dictionary
  --where TEXT                    Optional where clause
  -p, --param <TEXT TEXT>...      Named :parameters for where clause
  --output TEXT                   Optional separate column to populate with the
                                  output
  --output-type [integer|float|blob|text]
                                  Column type to use for the output column
  --drop                          Drop original column afterwards
  -s, --silent                    Don't show a progress bar
  --pdb                           Open pdb debugger on first error
  -h, --help                      Show this message and exit.
```

1.7.13 tables

See *Listing tables*.

```
Usage: sqlite-utils tables [OPTIONS] PATH

  List the tables in the database

  Example:

      sqlite-utils tables trees.db

Options:
  --fts4                 Just show FTS4 enabled tables
  --fts5                 Just show FTS5 enabled tables
  --counts               Include row counts per table
  --nl                   Output newline-delimited JSON
  --arrays               Output rows as arrays instead of objects
  --csv                  Output CSV
  --tsv                  Output TSV
  --no-headers           Omit headers from CSV/TSV and table/--fmt output
  -t, --table            Output as a formatted table
  --fmt TEXT             Table format - one of asciidoc, colon_grid,
                         double_grid, double_outline, fancy_grid, fancy_outline,
                         github, grid, heavy_grid, heavy_outline, html, jira,
                         latex, latex_booktabs, latex_longtable, latex_raw,
                         mediawiki, mixed_grid, mixed_outline, moinmoin, orgtbl,
                         outline, pipe, plain, presto, pretty, psql,
                         rounded_grid, rounded_outline, rst, simple,
                         simple_grid, simple_outline, textile, tsv, unsafehtml,
                         youtrack
  --json-cols            Detect JSON cols and output them as JSON, not escaped
                         strings
  --ascii                Escape non-ASCII characters in JSON output as \uXXXX
  --columns              Include list of columns for each table
  --schema               Include schema for each table
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.14 views

See *Listing views*.

```
Usage: sqlite-utils views [OPTIONS] PATH

  List the views in the database

  Example:

      sqlite-utils views trees.db

Options:
  --counts               Include row counts per view
  --nl                   Output newline-delimited JSON
```

(continues on next page)

(continued from previous page)

```
  --arrays                Output rows as arrays instead of objects
  --csv                   Output CSV
  --tsv                   Output TSV
  --no-headers            Omit headers from CSV/TSV and table/--fmt output
  -t, --table             Output as a formatted table
  --fmt TEXT              Table format - one of asciidoc, colon_grid,
                          double_grid, double_outline, fancy_grid, fancy_outline,
                          github, grid, heavy_grid, heavy_outline, html, jira,
                          latex, latex_booktabs, latex_longtable, latex_raw,
                          mediawiki, mixed_grid, mixed_outline, moinmoin, orgtbl,
                          outline, pipe, plain, presto, pretty, psql,
                          rounded_grid, rounded_outline, rst, simple,
                          simple_grid, simple_outline, textile, tsv, unsafehtml,
                          youtrack
  --json-cols             Detect JSON cols and output them as JSON, not escaped
                          strings
  --ascii                 Escape non-ASCII characters in JSON output as \uXXXX
  --columns               Include list of columns for each view
  --schema                Include schema for each view
  --load-extension TEXT   Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.15 rows

See *Returning all rows in a table*.

```
Usage: sqlite-utils rows [OPTIONS] PATH DBTABLE

  Output all rows in the specified table

  Example:

      sqlite-utils rows trees.db Trees

Options:
  -c, --column TEXT           Columns to return
  --where TEXT                Optional where clause
  -o, --order TEXT            Order by ('column' or 'column desc')
  -p, --param <TEXT TEXT>...  Named :parameters for where clause
  --limit INTEGER             Number of rows to return - defaults to everything
  --offset INTEGER            SQL offset to use
  --nl                        Output newline-delimited JSON
  --arrays                    Output rows as arrays instead of objects
  --csv                       Output CSV
  --tsv                       Output TSV
  --no-headers                Omit headers from CSV/TSV and table/--fmt output
  -t, --table                 Output as a formatted table
  --fmt TEXT                  Table format - one of asciidoc, colon_grid,
                              double_grid, double_outline, fancy_grid,
                              fancy_outline, github, grid, heavy_grid,
                              heavy_outline, html, jira, latex, latex_booktabs,
                              latex_longtable, latex_raw, mediawiki, mixed_grid,
```

(continues on next page)

(continued from previous page)

```
                              mixed_outline, moinmoin, orgtbl, outline, pipe,
                              plain, presto, pretty, psql, rounded_grid,
                              rounded_outline, rst, simple, simple_grid,
                              simple_outline, textile, tsv, unsafehtml, youtrack
  --json-cols                 Detect JSON cols and output them as JSON, not
                              escaped strings
  --ascii                     Escape non-ASCII characters in JSON output as
                              \uXXXX
  --load-extension TEXT       Path to SQLite extension, with optional
                              :entrypoint
  -h, --help                  Show this message and exit.
```

1.7.16 triggers

See *Listing triggers*.

```
Usage: sqlite-utils triggers [OPTIONS] PATH [TABLES]...

  Show triggers configured in this database

  Example:

      sqlite-utils triggers trees.db

Options:
  --nl                    Output newline-delimited JSON
  --arrays                Output rows as arrays instead of objects
  --csv                   Output CSV
  --tsv                   Output TSV
  --no-headers            Omit headers from CSV/TSV and table/--fmt output
  -t, --table             Output as a formatted table
  --fmt TEXT              Table format - one of asciidoc, colon_grid,
                          double_grid, double_outline, fancy_grid, fancy_outline,
                          github, grid, heavy_grid, heavy_outline, html, jira,
                          latex, latex_booktabs, latex_longtable, latex_raw,
                          mediawiki, mixed_grid, mixed_outline, moinmoin, orgtbl,
                          outline, pipe, plain, presto, pretty, psql,
                          rounded_grid, rounded_outline, rst, simple,
                          simple_grid, simple_outline, textile, tsv, unsafehtml,
                          youtrack
  --json-cols             Detect JSON cols and output them as JSON, not escaped
                          strings
  --ascii                 Escape non-ASCII characters in JSON output as \uXXXX
  --load-extension TEXT   Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.17 indexes

See *Listing indexes*.

```
Usage: sqlite-utils indexes [OPTIONS] PATH [TABLES]...

```

(continues on next page)

(continued from previous page)

```
  Show indexes for the whole database or specific tables

  Example:

      sqlite-utils indexes trees.db Trees

Options:
  --aux                   Include auxiliary columns
  --nl                    Output newline-delimited JSON
  --arrays                Output rows as arrays instead of objects
  --csv                   Output CSV
  --tsv                   Output TSV
  --no-headers            Omit headers from CSV/TSV and table/--fmt output
  -t, --table             Output as a formatted table
  --fmt TEXT              Table format - one of asciidoc, colon_grid,
                          double_grid, double_outline, fancy_grid, fancy_outline,
                          github, grid, heavy_grid, heavy_outline, html, jira,
                          latex, latex_booktabs, latex_longtable, latex_raw,
                          mediawiki, mixed_grid, mixed_outline, moinmoin, orgtbl,
                          outline, pipe, plain, presto, pretty, psql,
                          rounded_grid, rounded_outline, rst, simple,
                          simple_grid, simple_outline, textile, tsv, unsafehtml,
                          youtrack
  --json-cols             Detect JSON cols and output them as JSON, not escaped
                          strings
  --ascii                 Escape non-ASCII characters in JSON output as \uXXXX
  --load-extension TEXT   Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.18 create-database

See *Creating an empty database*.

```
Usage: sqlite-utils create-database [OPTIONS] PATH

  Create a new empty database file

  Example:

      sqlite-utils create-database trees.db

Options:
  --enable-wal            Enable WAL mode on the created database
  --init-spatialite       Enable SpatiaLite on the created database
  --load-extension TEXT   Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.19 create-table

See *Creating tables*.

```
Usage: sqlite-utils create-table [OPTIONS] PATH TABLE COLUMNS...

  Add a table with the specified columns. Columns should be specified using
  name, type pairs, for example:

      sqlite-utils create-table my.db people \
          id integer \
          name text \
          height real \
          photo blob --pk id

  Valid column types are text, integer, real, float and blob.

Options:
  --pk TEXT                  Column to use as primary key
  --not-null TEXT            Columns that should be created as NOT NULL
  --default <TEXT TEXT>...   Default value that should be set for a column
  --fk <TEXT TEXT TEXT>...   Column, other table, other column to set as a
                             foreign key
  --ignore                   If table already exists, do nothing
  --replace                  If table already exists, replace it
  --transform                If table already exists, try to transform the schema
  --load-extension TEXT      Path to SQLite extension, with optional :entrypoint
  --strict                   Apply STRICT mode to created table
  -h, --help                 Show this message and exit.
```

1.7.20 create-index

See *Creating indexes*.

```
Usage: sqlite-utils create-index [OPTIONS] PATH TABLE COLUMN...

  Add an index to the specified table for the specified columns

  Example:

      sqlite-utils create-index chickens.db chickens name

  To create an index in descending order:

      sqlite-utils create-index chickens.db chickens -- -name

Options:
  --name TEXT                Explicit name for the new index
  --unique                   Make this a unique index
  --if-not-exists, --ignore  Ignore if index already exists
  --analyze                  Run ANALYZE after creating the index
  --load-extension TEXT      Path to SQLite extension, with optional :entrypoint
  -h, --help                 Show this message and exit.
```

1.7.21 migrate

See *Running migrations*.

```
Usage: sqlite-utils migrate [OPTIONS] DB_PATH [MIGRATIONS]...

  Apply pending database migrations.

  Usage:

      sqlite-utils migrate database.db

  This will find the migrations.py file in the current directory or
  subdirectories and apply any pending migrations.

  Or pass paths to one or more migrations.py files directly:

      sqlite-utils migrate database.db path/to/migrations.py

  Pass --list to see a list of applied and pending migrations without applying
  them.

  Use --stop-before migration_set:name to stop before a migration. This option
  can be used multiple times.

Options:
  --stop-before TEXT  Stop before applying this migration. Use set:name to
                      target a migration set.
  --list              List migrations without running them
  -v, --verbose       Show verbose output
  -h, --help          Show this message and exit.
```

1.7.22 enable-fts

See *Configuring full-text search*.

```
Usage: sqlite-utils enable-fts [OPTIONS] PATH TABLE COLUMN...

  Enable full-text search for specific table and columns

  Example:

      sqlite-utils enable-fts chickens.db chickens name

Options:
  --fts4                 Use FTS4
  --fts5                 Use FTS5
  --tokenize TEXT        Tokenizer to use, e.g. porter
  --create-triggers      Create triggers to update the FTS tables when the
                         parent table changes.
  --replace              Replace existing FTS configuration if it exists
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.23 populate-fts

```
Usage: sqlite-utils populate-fts [OPTIONS] PATH TABLE COLUMN...

  Re-populate full-text search for specific table and columns

  Example:

      sqlite-utils populate-fts chickens.db chickens name

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.24 rebuild-fts

```
Usage: sqlite-utils rebuild-fts [OPTIONS] PATH [TABLES]...

  Rebuild all or specific full-text search tables

  Example:

      sqlite-utils rebuild-fts chickens.db chickens

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.25 disable-fts

```
Usage: sqlite-utils disable-fts [OPTIONS] PATH TABLE

  Disable full-text search for specific table

  Example:

      sqlite-utils disable-fts chickens.db chickens

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.26 optimize

See *Optimize*.

```
Usage: sqlite-utils optimize [OPTIONS] PATH [TABLES]...

  Optimize all full-text search tables and then run VACUUM - should shrink the
  database file

  Example:
```

(continues on next page)

(continued from previous page)

```
      sqlite-utils optimize chickens.db

Options:
  --no-vacuum            Don't run VACUUM
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.27 analyze

See *Optimizing index usage with ANALYZE*.

```
Usage: sqlite-utils analyze [OPTIONS] PATH [NAMES]...

  Run ANALYZE against the whole database, or against specific named indexes and
  tables

  Example:

      sqlite-utils analyze chickens.db

Options:
  -h, --help  Show this message and exit.
```

1.7.28 vacuum

See *Vacuum*.

```
Usage: sqlite-utils vacuum [OPTIONS] PATH

  Run VACUUM against the database

  Example:

      sqlite-utils vacuum chickens.db

Options:
  -h, --help  Show this message and exit.
```

1.7.29 dump

See *Dumping the database to SQL*.

```
Usage: sqlite-utils dump [OPTIONS] PATH

  Output a SQL dump of the schema and full contents of the database

  Example:

      sqlite-utils dump chickens.db

Options:
```

(continues on next page)

(continued from previous page)

```
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.30 add-column

See *Adding columns*.

```
Usage: sqlite-utils add-column [OPTIONS] PATH TABLE COL_NAME
                       [integer|int|float|real|text|str|blob|bytes]

  Add a column to the specified table

  Example:

      sqlite-utils add-column chickens.db chickens weight float

Options:
  --fk TEXT                Table to reference as a foreign key
  --fk-col TEXT            Referenced column on that foreign key table - if
                           omitted will automatically use the primary key
  --not-null-default TEXT  Add NOT NULL DEFAULT 'TEXT' constraint
  --ignore                 If column already exists, do nothing
  --load-extension TEXT    Path to SQLite extension, with optional :entrypoint
  -h, --help               Show this message and exit.
```

1.7.31 add-foreign-key

See *Adding foreign key constraints*.

```
Usage: sqlite-utils add-foreign-key [OPTIONS] PATH TABLE COLUMN [OTHER_TABLE]
                           [OTHER_COLUMN]

  Add a new foreign key constraint to an existing table

  Example:

      sqlite-utils add-foreign-key my.db books author_id authors id

Options:
  --ignore               If foreign key already exists, do nothing
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.32 add-foreign-keys

See *Adding multiple foreign keys at once*.

```
Usage: sqlite-utils add-foreign-keys [OPTIONS] PATH [FOREIGN_KEY]...

  Add multiple new foreign key constraints to a database

  Example:
```

(continues on next page)

(continued from previous page)

```
    sqlite-utils add-foreign-keys my.db \
        books author_id authors id \
        authors country_id countries id

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.33 index-foreign-keys

See *Adding indexes for all foreign keys*.

```
Usage: sqlite-utils index-foreign-keys [OPTIONS] PATH

  Ensure every foreign key column has an index on it

  Example:

      sqlite-utils index-foreign-keys chickens.db

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.34 enable-wal

See *WAL mode*.

```
Usage: sqlite-utils enable-wal [OPTIONS] PATH...

  Enable WAL for database files

  Example:

      sqlite-utils enable-wal chickens.db

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.35 disable-wal

```
Usage: sqlite-utils disable-wal [OPTIONS] PATH...

  Disable WAL for database files

  Example:

      sqlite-utils disable-wal chickens.db
```

(continues on next page)

(continued from previous page)

```
Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.36 enable-counts

See *Enabling cached counts*.

```
Usage: sqlite-utils enable-counts [OPTIONS] PATH [TABLES]...

  Configure triggers to update a _counts table with row counts

  Example:

      sqlite-utils enable-counts chickens.db

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.37 reset-counts

```
Usage: sqlite-utils reset-counts [OPTIONS] PATH

  Reset calculated counts in the _counts table

  Example:

      sqlite-utils reset-counts chickens.db

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.38 duplicate

See *Duplicating tables*.

```
Usage: sqlite-utils duplicate [OPTIONS] PATH TABLE NEW_TABLE

  Create a duplicate of this table, copying across the schema and all row data.

Options:
  --ignore               If table does not exist, do nothing
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.39 rename-table

See *Renaming a table*.

```
Usage: sqlite-utils rename-table [OPTIONS] PATH TABLE NEW_NAME

  Rename this table.

Options:
  --ignore                If table does not exist, do nothing
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.40 drop-table

See *Dropping tables*.

```
Usage: sqlite-utils drop-table [OPTIONS] PATH TABLE

  Drop the specified table

  Example:

      sqlite-utils drop-table chickens.db chickens

Options:
  --ignore                If table does not exist, do nothing
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.41 create-view

See *Creating views*.

```
Usage: sqlite-utils create-view [OPTIONS] PATH VIEW SELECT

  Create a view for the provided SELECT query

  Example:

      sqlite-utils create-view chickens.db heavy_chickens \
        'select * from chickens where weight > 3'

Options:
  --ignore                If view already exists, do nothing
  --replace               If view already exists, replace it
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help              Show this message and exit.
```

1.7.42 drop-view

See *Dropping views*.

```
Usage: sqlite-utils drop-view [OPTIONS] PATH VIEW

  Drop the specified view

  Example:

      sqlite-utils drop-view chickens.db heavy_chickens

Options:
  --ignore               If view does not exist, do nothing
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.43 install

See *Installing packages*.

```
Usage: sqlite-utils install [OPTIONS] [PACKAGES]...

  Install packages from PyPI into the same environment as sqlite-utils

Options:
  -U, --upgrade        Upgrade packages to latest version
  -e, --editable TEXT  Install a project in editable mode from this path
  -h, --help           Show this message and exit.
```

1.7.44 uninstall

See *Uninstalling packages*.

```
Usage: sqlite-utils uninstall [OPTIONS] PACKAGES...

  Uninstall Python packages from the sqlite-utils environment

Options:
  -y, --yes   Don't ask for confirmation
  -h, --help  Show this message and exit.
```

1.7.45 add-geometry-column

See *SpatiaLite helpers*.

```
Usage: sqlite-utils add-geometry-column [OPTIONS] DB_PATH TABLE COLUMN_NAME

  Add a SpatiaLite geometry column to an existing table. Requires SpatiaLite
  extension.

  By default, this command will try to load the SpatiaLite extension from usual
  paths. To load it from a specific path, use --load-extension.
```

(continues on next page)

(continued from previous page)

```
Options:
  -t, --type␣
↪[point|linestring|polygon|multipoint|multilinestring|multipolygon|geometrycollection|geometry]
                                  Specify a geometry type for this column.
                                  [default: GEOMETRY]
  --srid INTEGER                  Spatial Reference ID. See
                                  https://spatialreference.org for details on
                                  specific projections.  [default: 4326]
  --dimensions TEXT               Coordinate dimensions. Use XYZ for three-
                                  dimensional geometries.
  --not-null                      Add a NOT NULL constraint.
  --load-extension TEXT           Path to SQLite extension, with optional
                                  :entrypoint
  -h, --help                      Show this message and exit.
```

1.7.46 create-spatial-index

See *Adding spatial indexes*.

```
Usage: sqlite-utils create-spatial-index [OPTIONS] DB_PATH TABLE COLUMN_NAME

  Create a spatial index on a SpatiaLite geometry column. The table and geometry
  column must already exist before trying to add a spatial index.

  By default, this command will try to load the SpatiaLite extension from usual
  paths. To load it from a specific path, use --load-extension.

Options:
  --load-extension TEXT  Path to SQLite extension, with optional :entrypoint
  -h, --help             Show this message and exit.
```

1.7.47 plugins

```
Usage: sqlite-utils plugins [OPTIONS]

  List installed plugins

Options:
  -h, --help  Show this message and exit.
```

1.8 Upgrading

This page describes the changes you may need to make to your own code or scripts when upgrading between major versions of `sqlite-utils`.

For the full list of changes in every release see the *Changelog*.

1.8.1 Upgrading from 3.x to 4.0

Requirements

- Python 3.10 or higher is required.
- The `click` dependency must be version 8.3.1 or later.

Command-line changes

Type detection is now the default for CSV and TSV imports. `sqlite-utils insert` and `sqlite-utils upsert` now detect column types when importing CSV or TSV data - previously every column was created as `TEXT` unless you passed `--detect-types`. To restore the old behavior pass the new `--no-detect-types` flag:

```
sqlite-utils insert data.db rows data.csv --csv --no-detect-types
```

Two related things have been removed:

- The `SQLITE_UTILS_DETECT_TYPES` environment variable.
- The old `-d/--detect-types` flag itself. Since detection is now the default the flag did nothing - remove it from any scripts that used it.

The convert command no longer skips falsey values. `sqlite-utils convert` previously skipped values that evaluated to `False` (empty strings, `0`) unless you passed `--no-skip-false`. All values are now converted and the `--no-skip-false` flag has been removed.

drop-table and drop-view check the object type. `sqlite-utils drop-table` now refuses to drop a view, and `drop-view` refuses to drop a table. Previously each would silently drop the wrong type of object if the name matched. If you relied on that (unlikely), use the matching command instead.

sqlite-utils tui has moved to a plugin. The optional terminal interface is now provided by the sqlite-utils-tui plugin:

```
sqlite-utils install sqlite-utils-tui
```

Python API changes

db.query() now rejects SQL that does not return rows. This is likely the most common change you will need to make to existing code. `db.query()` used to accept any SQL statement - passing one that returns no rows, such as an `INSERT` or `UPDATE` without a `RETURNING` clause or a `CREATE TABLE`, did nothing at all, silently. Those statements now raise a `ValueError`, and are rolled back so they have no effect on the database. Transaction control statements (`BEGIN`, `COMMIT`, `END`, `ROLLBACK`, `SAVEPOINT`, `RELEASE`) plus `VACUUM`, `ATTACH` and `DETACH` are also rejected with a `ValueError`, without being executed at all. Use `db.execute()` for statements that do not return rows:

```
# 3.x accepted this but silently did nothing:
db.query("update dogs set name = 'Cleopaws'")

# In 4.0 use execute() for SQL that does not return rows:
db.execute("update dogs set name = 'Cleopaws'")
```

db.query() executes immediately. `db.query(sql)` previously returned a generator that did not execute the SQL until you started iterating over it. The SQL now runs as soon as the method is called - rows are still fetched lazily, but errors in your SQL raise at the `db.query()` call site rather than on first iteration, and a write with a `RETURNING` clause takes effect even if you never iterate over its results.

db.table() no longer returns views. `db.table(name)` now raises a `sqlite_utils.db.NoTable` exception if `name` is a SQL view. Use the new `db.view(name)` method for views:

```
table = db.table("my_table")
view = db.view("my_view")
```

`db["name"]` still returns either a `Table` or a `View` depending on what exists in the database.

Upserts use INSERT ... ON CONFLICT. Upsert operations now use SQLite's `INSERT ... ON CONFLICT SET` syntax rather than the previous `INSERT OR IGNORE` followed by `UPDATE`. If your code depends on the old behavior, pass `use_old_upsert=True` to the `Database()` constructor - see *Alternative upserts using INSERT OR IGNORE*.

Upsert records must include their primary keys. `table.upsert()` and `table.upsert_all()` now raise `sqlite_utils.db.PrimaryKeyRequired` if a record is missing a value for any primary key column (or has `None` for one). Previously such records were quietly inserted as new rows. Relatedly, `pk=` is now optional when the table already exists with a primary key - it is detected automatically.

Floating point columns are now REAL. Auto-detected floating point columns are created with the correct SQLite type `REAL` instead of `FLOAT`. Code that inspects column types should expect `REAL`.

Generated schemas use double quotes. Tables created by this library now wrap table and column names in `"double-quotes"` where they previously used `[square-braces]`. If you compare `table.schema` strings against expected values you will need to update them.

table.convert() no longer skips falsey values. Matching the CLI change above, `table.convert()` now converts every value. The `skip_false` parameter has been removed - previously it defaulted to `True`, skipping empty strings and other falsey values.

Null values are no longer extracted into lookup tables. `table.extract()` and the `sqlite-utils extract` command leave rows alone if every extracted column is `null` - the new foreign key column is left as `null` instead of pointing at an all-`null` record in the lookup table. The `extracts=` insert option similarly keeps `None` values as `null`. Relatedly, `table.lookup()` now compares values using `IS` so that looking up a value containing `None` returns the existing matching row - previously it inserted a duplicate row on every call.

ensure_autocommit_off() is now ensure_autocommit_on(). The `db.ensure_autocommit_off()` context manager has been renamed to `db.ensure_autocommit_on()`. The old name described the opposite of what the method did: it temporarily puts the connection into driver-level autocommit mode (by setting `isolation_level = None`), so that statements such as `PRAGMA journal_mode=wal` can run outside of an implicit transaction. The behavior is unchanged - update any calls to use the new name.

View.enable_fts() has been removed. The `View` class previously had an `enable_fts()` method that existed only to raise `NotImplementedError` - full-text search is not supported for views. Calling it now raises `AttributeError` like any other missing method.

ForeignKey is now a dataclass, not a namedtuple. The `ForeignKey` objects returned by `table.foreign_keys` gained new fields - `columns`, `other_columns`, `is_compound`, `on_delete` and `on_update` - so that compound (multi-column) foreign keys and foreign key actions can be represented. To make room for those fields cleanly `ForeignKey` is now a dataclass rather than a `namedtuple`, so it can no longer be unpacked or indexed as a tuple. Access its fields by name instead:

```
# 3.x - tuple unpacking, no longer works:
for table, column, other_table, other_column in db["courses"].foreign_keys:
    ...

# 4.0 - access fields by name:
for fk in db["courses"].foreign_keys:
    fk.table, fk.column, fk.other_table, fk.other_column
```

Attempting the old unpacking or `fk[0]` indexing now raises `TypeError`, so any code using those patterns will fail loudly rather than silently misbehave. Like the old namedtuple, `ForeignKey` instances are immutable and hashable

- they can be collected into sets and used as dictionary keys. Note that equality now includes the `on_delete` and `on_update` actions: a `ForeignKey` with `ON DELETE CASCADE` is not equal to one without.

Compound foreign keys - previously returned as one `ForeignKey` per column, misleadingly suggesting several independent single-column keys - are now returned as a single `ForeignKey` with `is_compound=True`. For these the scalar `column` and `other_column` fields are `None`; use the `columns` and `other_columns` tuples instead. Single-column foreign keys are unaffected apart from the class change: `column`/`other_column` behave as before and `columns`/`other_columns` are one-item tuples.

Two related behavior changes to `table.transform()`: compound foreign keys now survive a transform (previously they were split into separate single-column keys), and `ON DELETE`/`ON UPDATE` actions such as `ON DELETE CASCADE` are now preserved (previously they were silently stripped from the schema).

Validation errors raise ValueError. Invalid arguments to Python API methods - for example `create_table()` with no columns, or `ignore=True` together with `replace=True` - now raise `ValueError`. They previously raised `AssertionError` from bare `assert` statements, which were silently skipped under `python -O`.

Transaction behavior is now well-defined. 4.0 introduces the *db.atomic()* context manager and uses it consistently for every write operation - the full model is described in *Transactions and saving your changes*. Changes you may notice:

- Write statements executed with raw `db.execute()` calls now commit automatically, unless a transaction is already open in which case they join it. Previously they opened an implicit transaction that nothing committed - if your code used `db.execute()` for writes and relied on `db.conn.rollback()` to undo them, open an explicit transaction with the new `db.begin()` method first.
- Multi-step operations such as `table.transform()` no longer commit an existing transaction you have open - they use savepoints inside it instead.
- `db.enable_wal()` and `db.disable_wal()` raise a `sqlite_utils.db.TransactionError` if called while a transaction is open, instead of silently committing it.
- Using `Database` as a context manager (`with Database(path) as db:`) closes the connection on exit *without* committing - a transaction you explicitly opened with `db.begin()` and did not commit is rolled back.
- `Database()` rejects connections created with the Python 3.12+ `sqlite3.connect(..., autocommit=True)` or `autocommit=False` options, raising `sqlite_utils.db.TransactionError`. On those connections every write the library made was silently discarded when the connection closed.

Packaging changes

- `sqlite-utils` now uses `pyproject.toml` in place of `setup.py`.
- `pip` is now a runtime dependency, used by the `sqlite-utils install` and `uninstall` commands.

New features to be aware of

Not breaking changes, but new in 4.0 and worth knowing about when you upgrade:

- A *database migrations system*, incorporating the functionality of the `sqlite-migrate` plugin. If you used that plugin, the built-in system reads the same `_sqlite_migrations` table - your applied migrations will not run again. Update your migration files to use `from sqlite_utils import Migrations`.
- *db.atomic()* for nested transaction support.
- `table.insert_all()` and `table.upsert_all()` accept an iterator of lists or tuples as an alternative to dictionaries - see *Inserting data from a list or tuple iterator*.

1.8.2 Upgrading from 2.x to 3.0

The 3.0 release redesigned search. The breaking changes were minor:

- `table.search()` returns a generator of dictionaries, sorted by relevance. It previously returned a list of tuples sorted by `rowid`.
- The `-c` shortcut for `--csv` and the `-f` shortcut for `--fmt` were removed from the CLI - use the full option names.

1.8.3 Upgrading from 1.x to 2.0

The 2.0 release changed the meaning of *upsert*. In 1.x, `table.upsert()` and `table.upsert_all()` actually performed `INSERT OR REPLACE` operations - entirely replacing the existing row. Since 2.0 an upsert updates only the columns you provide, leaving other columns untouched.

If you want the 1.x behavior, use `table.insert(..., replace=True)` or `table.insert_all(..., replace=True)` instead.

1.9 Contributing

Development of `sqlite-utils` takes place in the sqlite-utils GitHub repository.

All improvements to the software should start with an issue. Read How I build a feature for a detailed description of the recommended process for building bug fixes or enhancements.

1.9.1 Obtaining the code

To work on this library locally, first checkout the code:

```
git clone git@github.com:simonw/sqlite-utils
cd sqlite-utils
```

Use `uv run` to run the development version of the tool:

```
uv run sqlite-utils --help
```

1.9.2 Running the tests

Use `uv run` to run the tests:

```
uv run pytest
```

1.9.3 Building the documentation

To build the documentation run this command:

```
uv run make livehtml --directory docs
```

This will start a server on port 8000 that will serve the documentation and live-reload any time you make an edit to a `.rst` file.

The cog tool is used to maintain portions of the documentation. You can run it like so:

```
uv run cog -r docs/*.rst
```

1.9.4 Linting and formatting

`sqlite-utils` uses Black for code formatting, and flake8 and mypy for linting and type checking:

```
uv run black .
```

Linting tools can be run like this:

```
uv run flake8
uv run mypy sqlite_utils
```

All three of these tools are run by our CI mechanism against every commit and pull request.

1.9.5 Using Just

If you install Just you can use it to manage your local development environment.

To run all of the tests and linters:

```
just
```

To run tests, or run a specific test module or test by name:

```
just test # All tests
just test tests/test_cli_memory.py # Just this module
just test -k test_memory_no_detect_types # Just this test
```

To run just the linters:

```
just lint
```

To apply Black to your code:

```
just black
```

To update documentation using Cog:

```
just cog
```

To run the live documentation server (this will run Cog first):

```
just docs
```

And to list all available commands:

```
just -l
```

1.9.6 Release process

Releases are performed using tags. When a new release is published on GitHub, a GitHub Actions workflow will perform the following:

- Run the unit tests against all supported Python versions. If the tests pass...
- Build a wheel bundle of the underlying Python source code
- Push that new wheel up to PyPI: https://pypi.org/project/sqlite-utils/

To deploy new releases you will need to have push access to the GitHub repository.

`sqlite-utils` follows Semantic Versioning:

```
major.minor.patch
```

We increment `major` for backwards-incompatible releases.

We increment `minor` for new features.

We increment `patch` for bugfix releass.

To release a new version, first create a commit that updates the version number in `pyproject.toml` and the *the changelog* with highlights of the new version. An example commit can be seen here:

```
# Update changelog
git commit -m " Release 3.29

Refs #423, #458, #467, #469, #470, #471, #472, #475" -a
git push
```

Referencing the issues that are part of the release in the commit message ensures the name of the release shows up on those issue pages, e.g. here.

You can generate the list of issue references for a specific release by copying and pasting text from the release notes or GitHub changes-since-last-release view into this Extract issue numbers from pasted text tool.

To create the tag for the release, create a new release on GitHub matching the new version number. You can convert the release notes to Markdown by copying and pasting the rendered HTML into this Paste to Markdown tool.

1.10 Changelog

1.10.1 Unreleased

- `sqlite-utils query` can now read the SQL query from standard input by passing `-` in place of the query, for example `echo "select * from dogs" | sqlite-utils query dogs.db -`. (#765)
- `sqlite-utils insert` and `sqlite-utils upsert` now accept a `--code` option for *providing a block of Python code* (or a path to a `.py` file) that defines a `rows()` function or `rows` iterable of rows to insert, as an alternative to importing from a file. (#684)

1.10.2 4.0 (2026-07-07)

The 4.0 release includes some minor backwards-incompatible fixes (hence the major version number bump) and introduces three major new features:

- *Database migrations*, providing a structured mechanism for evolving a project's schema over time. (#752)
- *Nested transaction support* via `db.atomic()`, plus numerous improvements to how transactions work across the library. (#755)
- Support for *compound foreign keys*, including creation, transformation and introspection through *table.foreign_keys*. (#594)

Other notable changes include:

- Upserts now use SQLite's `INSERT ... ON CONFLICT ... DO UPDATE SET` syntax, detect existing table primary keys automatically and reject records that are missing required primary key values. (#652)
- `db.query()` now executes immediately and rejects statements that do not return rows; use `db.execute()` for writes and DDL.

- CSV and TSV imports now detect column types by default, while inserts into existing tables preserve those tables' column types. (#679)
- Foreign key handling now preserves `ON DELETE`/`ON UPDATE` actions during transforms and resolves referenced primary keys more accurately. (#530)
- Column names passed to Python API methods are now matched case-insensitively, mirroring SQLite's own identifier behavior. (#760)
- The command-line tool now emits UTF-8 JSON output by default, with `--ascii` available to restore escaped output. (#625)
- `table.extract()` and `extracts=` no longer create lookup table records for all-`null` values. (#186)

See *Upgrading from 3.x to 4.0* for details on backwards-incompatible changes.

The detailed release notes for the features and fixes shipped during the 4.0 pre-release cycle are available in *4.0a0*, *4.0a1*, *4.0rc1*, *4.0rc2*, *4.0rc3* and *4.0rc4*.

Bug fixes since 4.0rc4

- Fixed 4.0 regressions in `insert`/`upsert` against tables that use SQLite's implicit `rowid` primary key. Passing `pk="rowid"`, `pk="_rowid_"` or `pk="oid"` now works again for rowid tables, and `last_pk` is set correctly. (#781)
- Fixed `insert(..., ignore=True)` and `insert_all(..., ignore=True)` so an ignored insert that conflicts with an existing primary key row now reports that existing row in `last_rowid` and `last_pk` where possible. This also works for compound primary keys and list-mode inserts. (#783)

1.10.3 4.0rc4 (2026-07-06)

- **Breaking change**: `table.extract()` - and the `sqlite-utils extract` command - no longer extract rows where every extracted column is `null`. Those rows now keep a `null` value in the new foreign key column instead of pointing at an all-`null` record in the lookup table. When extracting multiple columns, rows are still extracted if at least one of the columns has a value. (#186)
- The `extracts=` option to `table.insert()` and friends no longer creates a lookup table record for `None` values - the column value stays `null`. Previously every batch of inserted rows containing a `None` value would add a duplicate `null` record to the lookup table.
- Fixed a bug where `table.lookup()` inserted a duplicate row on every call if any of the lookup values were `None`. Lookup values are now compared using `IS` so that `None` values match existing rows correctly.
- JSON output from the command-line tool no longer escapes non-ASCII characters, so `sqlite-utils data.db "select '' as text"` now outputs `[{"text": ""}]`. This matches how values were already stored by `insert` and how CSV/TSV output already behaved. A new `--ascii` option restores the previous behavior of escaping non-ASCII characters, for output destinations that cannot handle UTF-8 - see *Unicode characters in JSON*. The option is available on the `query`, `rows`, `search`, `tables`, `views`, `triggers`, `indexes` and `memory` commands. The `convert --multi --dry-run` preview and `plugins` output also no longer escape non-ASCII characters. (#625)
- `--no-headers` now omits the header row from `--fmt` and `--table` output, not just CSV and TSV output. (#566)
- `table.insert_all(..., pk=...)` now raises `InvalidColumns` if `pk=` names columns that do not exist in an existing table. Previously this behaved inconsistently, with single-row inserts raising a `KeyError` while other row counts succeeded. (#732)
- Fixed an `IndexError` from `table.insert(..., pk=..., ignore=True)` when an ignored insert followed writes to another table on the same connection. `last_pk` is now populated from the explicit primary key value instead of looking up a stale `lastrowid`. (#554)

- Fixed a bug where a failed write statement executed with `db.execute()` left the driver's implicit transaction open. Every subsequent write then joined that phantom transaction, which nothing committed, so their work was silently rolled back when the connection was closed. The implicit transaction opened by a failed statement is now rolled back before the exception is raised. A failed write inside a transaction opened with `db.begin()` or `db.atomic()` leaves that transaction open and untouched, as before.
- Fixed a bug where transaction-control statements prefixed with an empty statement - `db.query("; COMMIT")` - or a UTF-8 byte order mark slipped past the check that rejects them, committing the caller's open transaction before raising a confusing `OperationalError`. The keyword scanner used by `db.query()` and `db.execute()` now skips leading `;` and byte order marks, matching what the `sqlite3` driver tolerates before the first token, so these statements are rejected with a `ValueError` without being executed. The same fix means `db.execute("; BEGIN")` no longer auto-commits the transaction it just opened.
- Documented a limitation of `db.query()`: a `PRAGMA` statement that returns no rows raises a `ValueError` but still takes effect, because PRAGMA statements run outside the savepoint guard used to roll back other rejected statements. Use `db.execute()` for row-less PRAGMA statements.
- Fixed exception masking when a statement destroys the enclosing transaction. An error such as a `RAISE(ROLLBACK)` trigger or `INSERT OR ROLLBACK` conflict rolls back the whole transaction, destroying every savepoint - the cleanup in `db.atomic()` and `db.query()` then failed with `OperationalError: no such savepoint` (or `cannot rollback - no transaction is active`), hiding the original `IntegrityError` from code that tried to catch it. Cleanup now checks whether a transaction is still open first, so the original exception propagates.
- `sqlite-utils migrate --list` is now read-only even when the migrations file uses the legacy `sqlite_migrate.Migrations` class, whose listing methods create the `_sqlite_migrations` table as a side effect. The listing now runs inside a transaction that is rolled back.
- `sqlite-utils insert ... --pk <missing column>` and `sqlite-utils extract <missing column>` now show a clean `Error:` message instead of a raw Python traceback. The `extract` command also shows a clean error when pointed at a view.
- Fixed a bug where running `table.extract()` more than once against the same lookup table inserted duplicate rows for values containing `null` - SQLite unique indexes treat `NULL` values as distinct, so `INSERT OR IGNORE` alone could not dedupe them. Each repeat extract added another copy that nothing referenced. The insert now uses an `IS`-based `NOT EXISTS` guard so `null`-containing rows match existing lookup rows.
- `db.add_foreign_keys()` no longer silently ignores requested `ON DELETE/ON UPDATE` actions when a foreign key with the same columns already exists - it raises `AlterError` suggesting `table.transform()`, since the actions of an existing foreign key cannot be changed in place. Exact duplicates, including actions, are still skipped so repeated calls stay idempotent. The method also now validates that compound foreign keys have the same number of columns on both sides, instead of silently discarding the extra columns.
- `db.ensure_autocommit_on()` now raises `TransactionError` if called while a transaction is open. Assigning `isolation_level` commits any pending transaction as a side effect, so entering the block silently committed the caller's open transaction and made a later `rollback()` a no-op.
- `sqlite-utils migrate --stop-before` now exits with an error if the named migration has already been applied. Previously the name passed validation but was only checked against pending migrations, so every migration after it was silently applied - the exact outcome `--stop-before` exists to prevent. `Migrations.apply(db, stop_before=...)` raises `ValueError` in the same situation, before applying anything.
- Fixed a regression where `table.insert(..., pk=..., alter=True)` raised `InvalidColumns` if the primary key column did not exist in the table yet. With `alter=True` the check now waits until the record keys are known, so a pk column supplied by the records is added by the alter as it was in 3.x. A pk column found in neither the table nor the records still raises `InvalidColumns`.
- Fixed a bug where inserting CSV or TSV data into an existing table rewrote that table's column types to match the incoming file. Type detection is the default in 4.0, so `sqlite-utils insert data.db places places.csv`

`--csv` against a table with a `TEXT` zip code column would convert the column to `INTEGER` and corrupt values with leading zeros - `"01234"` became `1234`. Detected types are now only applied when the `insert` or `upsert` command creates the table.

- Fixed `pks_and_rows_where()` raising `AttributeError` when called on a view, and no longer double-quotes the synthesized `rowid` column in its generated SQL - SQLite turns a double-quoted identifier that does not resolve into a string literal, which on a view produced a confusing `KeyError` instead of the `OperationalError` raised in 3.x. Compound primary keys returned by this method now follow `PRIMARY KEY` declaration order.
- The `foreign_keys=` argument to `create()` and `insert()` accepts a mixed list of `ForeignKey` objects, tuples and column name strings again. In 4.0 pre-releases mixing `ForeignKey` objects with tuples raised a `ValueError` - a regression from 3.x, where `ForeignKey` was a `namedtuple` and passed the tuple checks.
- `ForeignKey` objects are hashable again. The 4.0 change from `namedtuple` to dataclass accidentally made them unhashable, breaking patterns like `set(table.foreign_keys)` that worked in 3.x. `ForeignKey` is now a frozen dataclass - immutable and hashable, like the namedtuple was.
- Fixed a bug where compound primary key columns were returned in table column order instead of `PRIMARY KEY` declaration order. For a table declared as `CREATE TABLE other (b TEXT, a TEXT, PRIMARY KEY (a, b))` an implicit `FOREIGN KEY (x, y) REFERENCES other` was introspected as referencing `(b, a)` when SQLite resolves it as `(a, b)` - running `transform()` on such a table then rewrote the schema with the inverted column order, silently reversing the meaning of the constraint and causing foreign key errors on valid data. `table.pks`, compound foreign key guessing and `transform()` now all use the primary key declaration order, and `transform()` no longer reorders a compound `PRIMARY KEY (b, a)` into table column order.

1.10.4 4.0rc3 (2026-07-05)

Breaking changes

- *table.foreign_keys* now returns `ForeignKey` objects that are dataclasses rather than `namedtuple` instances, so they can no longer be unpacked or indexed as `(table, column, other_table, other_column)` tuples - access their fields by name instead. Compound (multi-column) foreign keys are now represented as a single `ForeignKey` with `is_compound=True` and populated `columns`/`other_columns` tuples, where `column` and `other_column` are `None`. Previously they were returned as one `ForeignKey` per column, misleadingly suggesting several independent foreign keys. See *Upgrading from 3.x to 4.0* for details. (#594)
- Removed support for using `sqlean.py` as a drop-in replacement for the Python standard library `sqlite3` module. `sqlite-utils` will now use `pysqlite3` if it is installed, otherwise it will use `sqlite3` from the standard library.
- The `db.ensure_autocommit_off()` context manager has been renamed to `db.ensure_autocommit_on()`, because the old name described the opposite of what it did. The method temporarily puts the connection into driver-level autocommit mode - by setting `isolation_level = None` - so that statements such as `PRAGMA journal_mode=wal` can run outside of an implicit transaction. (#705)

Compound foreign key support

- Tables can now be created with *compound foreign keys*, by passing tuples of column names in `foreign_keys=`: `foreign_keys=[(("campus_name", "dept_code"), "departments")]`. The referenced columns default to the compound primary key of the other table. Compound keys are rendered as table-level `FOREIGN KEY` constraints in the generated schema.
- `table.transform()` now preserves compound foreign keys, applying any column renames to them. Dropping a column that is part of a compound foreign key drops the whole constraint, matching the existing single-column behavior. `drop_foreign_keys=` accepts a bare column name - dropping any foreign key that column participates in - or a tuple of columns to target a compound key precisely.
- `table.add_foreign_key()` and `db.add_foreign_keys()` accept tuples of column names to add a compound foreign key to an existing table.

- `db.index_foreign_keys()` creates a single composite index for a compound foreign key.

Other foreign key improvements

- `ForeignKey` now exposes `on_delete` and `on_update` fields reflecting the foreign key's `ON DELETE`/`ON UPDATE` actions, and `table.transform()` preserves those actions. Previously a transform silently stripped clauses such as `ON DELETE CASCADE` from the table schema.
- `table.add_foreign_key()` accepts new `on_delete=` and `on_update=` parameters for creating foreign keys with actions, e.g. `table.add_foreign_key("author_id", "authors", "id", on_delete="CASCADE")`. (#530)
- Foreign keys declared as `REFERENCES other_table` with no explicit column are now resolved to the other table's primary key by `table.foreign_keys`, instead of reporting `other_column=None`.
- Fixed a `TypeError` when sorting `ForeignKey` objects where some were compound.

Case-insensitive column matching

Column names passed to Python API methods are now matched against the table schema case-insensitively, mirroring how SQLite itself treats identifiers. Previously many methods accepted mixed-case identifiers in the SQL they generated but then failed - or silently did nothing - when performing Python-side comparisons against the schema. (#760) Fixes include:

- `table.insert()` and `table.upsert()` now populate `table.last_pk` correctly when the `pk=` argument uses different casing to the table schema or the record keys - previously this raised a `KeyError` after the row had already been written.
- Upserts no longer raise or misbehave when the casing of `pk=` differs from the casing of the record keys. The primary key columns are correctly excluded from the generated `DO UPDATE SET` clause.
- `table.transform()` arguments `types=`, `rename=`, `drop=`, `pk=`, `not_null=`, `defaults=`, `column_order=` and `drop_foreign_keys=` all resolve column names case-insensitively. Previously options like `rename={"name": "title"}` against a column called `Name` were silently ignored.
- `db.create_table(..., transform=True)` now recognizes existing columns that differ only by case, instead of attempting to add them again and failing with `duplicate column name`. The casing used in the existing schema is preserved.
- `table.lookup()` returns the primary key value even if `pk=` casing differs from the schema, and recognizes existing unique indexes case-insensitively instead of creating redundant ones.
- `table.extract()` and `table.convert()` - including `multi=True` and `output=` - accept column names in any casing.
- Foreign key columns are validated and recorded using the casing of the actual schema columns, in `foreign_keys=` when creating tables, `db.add_foreign_keys()`, `table.add_foreign_key()` and `table.add_column(fk_col=...)`. Duplicate foreign key detection is also case-insensitive.
- `table.create()` with `pk=`, `not_null=`, `defaults=` or `column_order=` referencing columns using different casing no longer creates an unwanted extra primary key column or raises a `ValueError`.

Everything else

- Fixed a bug where `table.transform()` could convert `DEFAULT TRUE`, `DEFAULT FALSE` and `DEFAULT NULL` column defaults into quoted string defaults when rebuilding a table. Thanks, Vincent Gao. (#764)

1.10.5 4.0rc2 (2026-07-04)

Breaking changes:

- Write statements executed with `db.execute()` are now committed automatically, unless a transaction is already open in which case they join it. Previously they opened an implicit transaction that stayed open until something committed it - writes appeared to work when read on the same connection but were silently rolled back when the connection closed. Code that relied on rolling back uncommitted `db.execute()` writes should use the new `db.begin()` method to open an explicit transaction first. The transaction model is documented in full at *Transactions and saving your changes*.
- `db.query()` now executes its SQL as soon as it is called, rather than waiting until the returned generator is first iterated. Rows are still fetched lazily during iteration. SQL errors are now raised at the call site, statements such as `INSERT ... RETURNING` are executed and committed immediately without needing to iterate over their results, and passing a statement that returns no rows - previously a silent no-op - now raises a `ValueError` recommending `db.execute()` instead. A statement rejected this way is rolled back before the error is raised, so it has no effect on the database.
- Python API validation errors now raise `ValueError` instead of `AssertionError`. Previously invalid arguments - such as `create_table()` with no columns, `transform()` on a table that does not exist, or passing both `ignore=True` and `replace=True` - were rejected using bare `assert` statements, which are silently skipped when Python runs with the `-O` flag. Code that caught `AssertionError` for these cases should catch `ValueError` instead.
- `table.upsert()` and `table.upsert_all()` now raise `PrimaryKeyRequired` if a record is missing a value for any primary key column, or has a value of `None` for one. Previously such records - which can never match an existing row - were quietly inserted as brand new rows, or triggered a confusing `KeyError` after the insert had already taken place.
- `db.enable_wal()` and `db.disable_wal()` now raise a `sqlite_utils.db.TransactionError` if called while a transaction is open. Previously they would silently commit the open transaction as a side effect of changing the journal mode, breaking the rollback guarantee of `db.atomic()` and of user-managed transactions.
- The `View` class no longer has an `enable_fts()` method. It existed only to raise `NotImplementedError`, since full-text search is not supported for views - calling it now raises `AttributeError` instead, and the method no longer appears in the API reference. The `sqlite-utils enable-fts` command shows a clean error when pointed at a view.
- The no-op `-d/--detect-types` flag has been removed from the `insert` and `upsert` commands. Type detection has been the default for CSV/TSV data since 4.0a1, so the flag did nothing - invocations using it should simply drop it. `--no-detect-types` remains available to disable detection.
- `Database()` now raises a `sqlite_utils.db.TransactionError` if passed a connection created with the Python 3.12+ `sqlite3.connect(..., autocommit=True)` or `autocommit=False` options. `commit()` and `rollback()` behave differently on those connections, which previously caused every write made by the library to be silently discarded when the connection closed.

Everything else:

- Fixed a bug where `table.delete_where()`, `table.optimize()` and `table.rebuild_fts()` did not commit their changes, leaving the connection inside an open transaction. Their work - and any subsequent writes - could then be silently rolled back when the connection was closed. All three now use `db.atomic()`, consistent with the other write methods.
- The `sqlite-utils drop-table` command now refuses to drop a view, and `drop-view` refuses to drop a table. Previously each would silently drop the wrong type of object if the name matched. Both now exit with an error suggesting the correct command to use.
- Migrations applied by the new *migrations system* now run inside a transaction, together with the record of the migration having been applied. If a migration raises an exception its changes are rolled back and it stays pend-

ing, so it can be safely re-applied after the error is fixed. Migrations that cannot run inside a transaction, such as those executing `VACUUM`, can opt out using `@migrations(transactional=False)` - see *Migrations and transactions*.

- `table.upsert()` and `table.upsert_all()` now detect the primary key or compound primary key of an existing table, so the `pk=` argument is no longer required when upserting into a table that already has a primary key.
- `db.table(table_name).insert({})` can now be used to insert a row consisting entirely of default values into an existing table, using `INSERT INTO ... DEFAULT VALUES`. (#759)
- Improvements to the `sqlite-utils migrate` command: `--stop-before` values that do not match any known migration are now an error instead of being silently ignored, `--stop-before` now works correctly with migration files that still use the older `sqlite_migrate.Migrations` class, and `--list` is now a read-only operation that no longer creates the database file or the migrations tracking table. `migrations.applied()` now returns migrations in the order they were applied.
- New `db.begin()`, `db.commit()` and `db.rollback()` methods for taking manual control of transactions, as an alternative to the `db.atomic()` context manager.
- New documentation: *Transactions and saving your changes* describes how transactions work and when changes are committed, and a new *Upgrading* page details the changes needed to move between major versions.

1.10.6 4.0rc1 (2026-06-21)

- New *database migrations system*, incorporating functionality that was previously provided by the separate sqlite-migrate plugin. Define migration sets using the new `sqlite_utils.Migrations` class and apply them using the `sqlite-utils migrate` command or the *migrations Python API*. (#752)
- New `db.atomic()` *context manager providing nested transaction support* using SQLite transactions and savepoints. Internal multi-step operations such as `table.transform()` now use this mechanism to avoid unexpectedly committing an existing transaction. (#755)
- `Database` objects can now be *used as context managers*, automatically closing the connection when the `with` block exits. The CLI also now closes database and file handles more reliably, resolving a number of `ResourceWarning` warnings. (#692)
- The `sqlite-utils convert` command can now accept a direct callable reference such as `r.parsedate` or `json.loads --import json` as the conversion code, as an alternative to calling it explicitly with `r.parsedate(value)`. (#686)
- Fixed a bug where CSV or TSV files with only a header row could crash `sqlite-utils insert` and `sqlite-utils memory` when type detection was enabled. Thanks, Rami Abdelrazzaq. (#702, #707)
- Fixed a bug where installed plugins could be loaded while running the test suite, despite the test-mode safeguard that disables plugin loading. Thanks, Rami Abdelrazzaq. (#713, #719)
- `table.detect_fts()` now recognizes legacy FTS virtual tables that quote the `content=` table name using square brackets, allowing `table.enable_fts(..., replace=True)` to replace them correctly. (#694)
- Now depends on Click 8.3.1 or later, removing compatibility workarounds for Click's `Sentinel` default values. (#666)
- Improved type annotations throughout the package, with `ty` now run in CI. (#697)
- Development tooling now uses `uv` dependency groups, with separate `dev` and `docs` groups. (#691)
- The test suite now runs against Python 3.15-dev. (#738)

1.10.7 3.39 (2025-11-24)

- Fixed a bug with `sqlite-utils install` when the tool had been installed using `uv`. (#687)
- The `--functions` argument now optionally accepts a path to a Python file as an alternative to a string full of code, and can be specified multiple times - see *Defining custom SQL functions*. (#659)
- `sqlite-utils` now requires Python 3.10 or higher.

1.10.8 4.0a1 (2025-11-23)

- **Breaking change**: The `db.table(table_name)` method now only works with tables. To access a SQL view use `db.view(view_name)` instead. (#657)
- The `table.insert_all()` and `table.upsert_all()` methods can now accept an iterator of lists or tuples as an alternative to dictionaries. The first item should be a list/tuple of column names. See *Inserting data from a list or tuple iterator* for details. (#672)
- **Breaking change**: The default floating point column type has been changed from `FLOAT` to `REAL`, which is the correct SQLite type for floating point values. This affects auto-detected columns when inserting data. (#645)
- Now uses `pyproject.toml` in place of `setup.py` for packaging. (#675)
- Tables in the Python API now do a much better job of remembering the primary key and other schema details from when they were first created. (#655)
- **Breaking change**: The `table.convert()` and `sqlite-utils convert` mechanisms no longer skip values that evaluate to `False`. Previously the `--skip-false` option was needed, this has been removed. (#542)
- **Breaking change**: Tables created by this library now wrap table and column names in `"double-quotes"` in the schema. Previously they would use `[square-braces]`. (#677)
- The `--functions` CLI argument now accepts a path to a Python file in addition to accepting a string full of Python code. It can also now be specified multiple times. (#659)
- **Breaking change:** Type detection is now the default behavior for the `insert` and `upsert` CLI commands when importing CSV or TSV data. Previously all columns were treated as `TEXT` unless the `--detect-types` flag was passed. Use the new `--no-detect-types` flag to restore the old behavior. The `SQLITE_UTILS_DETECT_TYPES` environment variable has been removed. (#679)

1.10.9 4.0a0 (2025-05-08)

- Upsert operations now use SQLite's `INSERT ... ON CONFLICT SET` syntax on all SQLite versions later than 3.23.1. This is a very slight breaking change for apps that depend on the previous `INSERT OR IGNORE` followed by `UPDATE` behavior. (#652)
- Python library users can opt-in to the previous implementation by passing `use_old_upsert=True` to the `Database()` constructor, see *Alternative upserts using INSERT OR IGNORE*.
- Dropped support for Python 3.8, added support for Python 3.13. (#646)
- `sqlite-utils tui` is now provided by the sqlite-utils-tui plugin. (#648)
- Test suite now also runs against SQLite 3.23.1, the last version (from 2018-04-10) before the new `INSERT ... ON CONFLICT SET` syntax was added. (#654)

1.10.10 3.38 (2024-11-23)

- Plugins can now reuse the implementation of the `sqlite-utils memory` CLI command with the new `return_db=True` parameter. (#643)

- `table.transform()` now recreates indexes after transforming a table. A new `sqlite_utils.db.TransformError` exception is raised if these indexes cannot be recreated due to conflicting changes to the table such as a column rename. Thanks, Mat Miller. (#633)
- `table.search()` now accepts a `include_rank=True` parameter, causing the resulting rows to have a `rank` column showing the calculated relevance score. Thanks, liunux4odoo. (#628)
- Fixed an error that occurred when creating a strict table with at least one floating point column. These `FLOAT` columns are now correctly created as `REAL` as well, but only for strict tables. (#644)

1.10.11 3.37 (2024-07-18)

- The `create-table` and `insert-files` commands all now accept multiple `--pk` options for compound primary keys. (#620)
- Now tested against Python 3.13 pre-release. (#619)
- Fixed a crash that can occur in environments with a broken `numpy` installation, producing a `module 'numpy' has no attribute 'int8'`. (#632)

1.10.12 3.36 (2023-12-07)

- **Support for creating tables in SQLite STRICT mode. Thanks, Taj Khattra. (#344)**
  - CLI commands `create-table`, `insert` and `upsert` all now accept a `--strict` option.
  - Python methods that can create a table - `table.create()` and `insert/upsert/insert_all/upsert_all` all now accept an optional `strict=True` parameter.
  - The `transform` command and `table.transform()` method preserve strict mode when transforming a table.
- The `sqlite-utils create-table` command now accepts `str`, `int` and `bytes` as aliases for `text`, `integer` and `blob` respectively. (#606)

1.10.13 3.35.2 (2023-11-03)

- The `--load-extension=spatialite` option and *find_spatialite()* utility function now both work correctly on `arm64` Linux. Thanks, Mike Coats. (#599)
- Fix for bug where `sqlite-utils insert` could cause your terminal cursor to disappear. Thanks, Luke Plant. (#433)
- `datetime.timedelta` values are now stored as `TEXT` columns. Thanks, Harald Nezbeda. (#522)
- Test suite is now also run against Python 3.12.

1.10.14 3.35.1 (2023-09-08)

- Fixed a bug where *table.transform()* would sometimes re-assign the `rowid` values for a table rather than keeping them consistent across the operation. (#592)

1.10.15 3.35 (2023-08-17)

Adding foreign keys to a table no longer uses `PRAGMA writable_schema = 1` to directly manipulate the `sqlite_master` table. This was resulting in errors in some Python installations where the SQLite library was compiled in a way that prevented this from working, in particular on macOS. Foreign keys are now added using the *table transformation* mechanism instead. (#577)

This new mechanism creates a full copy of the table, so it is likely to be significantly slower for large tables, but will no longer trigger `table sqlite_master may not be modified` errors on platforms that do not support `PRAGMA writable_schema = 1`.

A new plugin, sqlite-utils-fast-fks, is now available for developers who still want to use that faster but riskier implementation.

Other changes:

- The *table.transform() method* has two new parameters: `foreign_keys=` allows you to replace the foreign key constraints defined on a table, and `add_foreign_keys=` lets you specify new foreign keys to add. These complement the existing `drop_foreign_keys=` parameter. (#577)
- The *sqlite-utils transform* command has a new `--add-foreign-key` option which can be called multiple times to add foreign keys to a table that is being transformed. (#585)
- *sqlite-utils convert* now has a `--pdb` option for opening a debugger on the first encountered error in your conversion script. (#581)
- Fixed a bug where `sqlite-utils install -e '.[test]'` option did not work correctly.

1.10.16 3.34 (2023-07-22)

This release introduces a new *plugin system*. Read more about this in sqlite-utils now supports plugins. (#567)

- Documentation describing *how to build a plugin*.
- Plugin hook: *register_commands(cli)*, for plugins to add extra commands to `sqlite-utils`. (#569)
- Plugin hook: *prepare_connection(conn)*. Plugins can use this to help prepare the SQLite connection to do things like registering custom SQL functions. Thanks, Alex Garcia. (#574)
- `sqlite_utils.Database(..., execute_plugins=False)` option for disabling plugin execution. (#575)
- `sqlite-utils install -e path-to-directory` option for installing editable code. This option is useful during the development of a plugin. (#570)
- `table.create(...)` method now accepts `replace=True` to drop and replace an existing table with the same name, or `ignore=True` to silently do nothing if a table already exists with the same name. (#568)
- `sqlite-utils insert ... --stop-after 10` option for stopping the insert after a specified number of records. Works for the `upsert` command as well. (#561)
- The `--csv` and `--tsv` modes for `insert` now accept a `--empty-null` option, which causes empty strings in the CSV file to be stored as `null` in the database. (#563)
- New `db.rename_table(table_name, new_name)` method for renaming tables. (#565)
- `sqlite-utils rename-table my.db table_name new_name` command for renaming tables. (#565)
- The `table.transform(...)` method now takes an optional `keep_table=new_table_name` parameter, which will cause the original table to be renamed to `new_table_name` rather than being dropped at the end of the transformation. (#571)
- Documentation now notes that calling `table.transform()` without any arguments will reformat the SQL schema stored by SQLite to be more aesthetically pleasing. (#564)

1.10.17 3.33 (2023-06-25)

- `sqlite-utils` will now use sqlean.py in place of `sqlite3` if it is installed in the same virtual environment. This is useful for Python environments with either an outdated version of SQLite or with restrictions on SQLite such as disabled extension loading or restrictions resulting in the `sqlite3.OperationalError: table sqlite_master may not be modified` error. (#559)

- New `with db.ensure_autocommit_off()` context manager, which ensures that the database is in autocommit mode for the duration of a block of code. This is used by `db.enable_wal()` and `db.disable_wal()` to ensure they work correctly with `pysqlite3` and `sqlean.py`.
- New `db.iterdump()` method, providing an iterator over SQL strings representing a dump of the database. This uses `sqlite-dump` if it is available, otherwise falling back on the `conn.iterdump()` method from `sqlite3`. Both `pysqlite3` and `sqlean.py` omit support for `iterdump()` - this method helps paper over that difference.

1.10.18 3.32.1 (2023-05-21)

- Examples in the *CLI documentation* can now all be copied and pasted without needing to remove a leading `$`. (#551)
- Documentation now covers *Setting up shell completion* for `bash` and `zsh`. (#552)

1.10.19 3.32 (2023-05-21)

- New experimental `sqlite-utils tui` interface for interactively building command-line invocations, powered by Trogon. This requires an optional dependency, installed using `sqlite-utils install trogon`. (#545)
- `sqlite-utils analyze-tables` command (*documentation*) now has a `--common-limit 20` option for changing the number of common/least-common values shown for each column. (#544)
- `sqlite-utils analyze-tables --no-most` and `--no-least` options for disabling calculation of most-common and least-common values.
- If a column contains only `null` values, `analyze-tables` will no longer attempt to calculate the most common and least common values for that column. (#547)
- Calling `sqlite-utils analyze-tables` with non-existent columns in the `-c/--column` option now results in an error message. (#548)
- The `table.analyze_column()` method (*documented here*) now accepts `most_common=False` and `least_common=False` options for disabling calculation of those values.

1.10.20 3.31 (2023-05-08)

- Dropped support for Python 3.6. Tests now ensure compatibility with Python 3.11. (#517)
- Automatically locates the SpatiaLite extension on Apple Silicon. Thanks, Chris Amico. (#536)
- New `--raw-lines` option for the `sqlite-utils query` and `sqlite-utils memory` commands, which outputs just the raw value of the first column of every row. (#539)
- Fixed a bug where `table.upsert_all()` failed if the `not_null=` option was passed. (#538)
- Fixed a `ResourceWarning` when using `sqlite-utils insert`. (#534)
- Now shows a more detailed error message when `sqlite-utils insert` is called with invalid JSON. (#532)
- `table.convert(..., skip_false=False)` and `sqlite-utils convert --no-skip-false` options, for avoiding a misfeature where the *convert()* mechanism skips rows in the database with a falsey value for the specified column. Fixing this by default would be a backwards-incompatible change and is under consideration for a 4.0 release in the future. (#527)
- Tables can now be created with self-referential foreign keys. Thanks, Scott Perry. (#537)
- `sqlite-utils transform` no longer breaks if a table defines default values for columns. Thanks, Kenny Song. (#509)
- Fixed a bug where repeated calls to `table.transform()` did not work correctly. Thanks, Martin Carpenter. (#525)

- Improved error message if `rows_from_file()` is passed a non-binary-mode file-like object. (#520)

1.10.21 3.30 (2022-10-25)

- Now tested against Python 3.11. (#502)
- New `table.search_sql(include_rank=True)` option, which adds a `rank` column to the generated SQL. Thanks, Jacob Chapman. (#480)
- Progress bars now display for newline-delimited JSON files using the `--nl` option. Thanks, Mischa Untaga. (#485)
- New `db.close()` method. (#504)
- Conversion functions passed to *table.convert(...)* can now return lists or dictionaries, which will be inserted into the database as JSON strings. (#495)
- `sqlite-utils install` and `sqlite-utils uninstall` commands for installing packages into the same virtual environment as `sqlite-utils`, *described here*. (#483)
- New *sqlite_utils.utils.flatten()* utility function. (#500)
- Documentation on *using Just* to run tests, linters and build documentation.
- Documentation now covers the *Release process* for this package.

1.10.22 3.29 (2022-08-27)

- The `sqlite-utils query`, `memory` and `bulk` commands now all accept a new `--functions` option. This can be passed a string of Python code, and any callable objects defined in that code will be made available to SQL queries as custom SQL functions. See *Defining custom SQL functions* for details. (#471)
- `db[table].create(...)` method now accepts a new `transform=True` parameter. If the table already exists it will be *transformed* to match the schema configuration options passed to the function. This may result in columns being added or dropped, column types being changed, column order being updated or not null and default values for columns being set. (#467)
- Related to the above, the `sqlite-utils create-table` command now accepts a `--transform` option.
- New introspection property: `table.default_values` returns a dictionary mapping each column name with a default value to the configured default value. (#475)
- The `--load-extension` option can now be provided a path to a compiled SQLite extension module accompanied by the name of an entrypoint, separated by a colon - for example `--load-extension ./lines0:sqlite3_lines0_noread_init`. This feature is modelled on code first contributed to Datasette by Alex Garcia. (#470)
- Functions registered using the *db.register_function()* method can now have a custom name specified using the new `db.register_function(fn, name=...)` parameter. (#458)
- *sqlite-utils rows* has a new `--order` option for specifying the sort order for the returned rows. (#469)
- All of the CLI options that accept Python code blocks can now all be used to define functions that can access modules imported in that same block of code without needing to use the `global` keyword. (#472)
- Fixed bug where `table.extract()` would not behave correctly for columns containing null values. Thanks, Forest Gregg. (#423)
- New tutorial: Cleaning data with sqlite-utils and Datasette shows how to use `sqlite-utils` to import and clean an example CSV file.
- Datasette and `sqlite-utils` now have a Discord community. Join the Discord here.

1.10.23 3.28 (2022-07-15)

- New *table.duplicate(new_name)* method for creating a copy of a table with a matching schema and row contents. Thanks, David. (#449)
- New `sqlite-utils duplicate data.db table_name new_name` CLI command for *Duplicating tables*. (#454)
- `sqlite_utils.utils.rows_from_file()` is now a *documented API*. It can be used to read a sequence of dictionaries from a file-like object containing CSV, TSV, JSON or newline-delimited JSON. It can be passed an explicit format or can attempt to detect the format automatically. (#443)
- `sqlite_utils.utils.TypeTracker` is now a documented API for detecting the likely column types for a sequence of string rows, see *Detecting column types using TypeTracker*. (#445)
- `sqlite_utils.utils.chunks()` is now a documented API for *splitting an iterator into chunks*. (#451)
- `sqlite-utils enable-fts` now has a `--replace` option for replacing the existing FTS configuration for a table. (#450)
- The `create-index`, `add-column` and `duplicate` commands all now take a `--ignore` option for ignoring errors should the database not be in the right state for them to operate. (#450)

1.10.24 3.27 (2022-06-14)

See also the annotated release notes for this release.

- Documentation now uses the Furo Sphinx theme. (#435)
- Code examples in documentation now have a "copy to clipboard" button. (#436)
- `sqlite_utils.utils.utils.rows_from_file()` is now a documented API, see *Reading rows from a file*. (#443)
- `rows_from_file()` has two new parameters to help handle CSV files with rows that contain more values than are listed in that CSV file's headings: `ignore_extras=True` and `extras_key="name-of-key"`. (#440)
- `sqlite_utils.utils.maximize_csv_field_size_limit()` helper function for increasing the field size limit for reading CSV files to its maximum, see *Setting the maximum CSV field size limit*. (#442)
- `table.search(where=, where_args=)` parameters for adding additional `WHERE` clauses to a search query. The `where=` parameter is available on `table.search_sql(...)` as well. See *Searching with table.search()*. (#441)
- Fixed bug where `table.detect_fts()` and other search-related functions could fail if two FTS-enabled tables had names that were prefixes of each other. (#434)

1.10.25 3.26.1 (2022-05-02)

- Now depends on click-default-group-wheel, a pure Python wheel package. This means you can install and use this package with Pyodide, which can run Python entirely in your browser using WebAssembly. (#429)

  Try that out using the Pyodide REPL:

```
>>> import micropip
>>> await micropip.install("sqlite-utils")
>>> import sqlite_utils
>>> db = sqlite_utils.Database(memory=True)
>>> list(db.query("select 3 * 5"))
[{'3 * 5': 15}]
```

1.10.26 3.26 (2022-04-13)

- New `errors=r.IGNORE/r.SET_NULL` parameter for the `r.parsedatetime()` and `r.parsedate()` *convert recipes*. (#416)
- Fixed a bug where `--multi` could not be used in combination with `--dry-run` for the *convert* command. (#415)
- New documentation: *Defining a convert() function*. (#420)
- More robust detection for whether or not `deterministic=True` is supported. (#425)

1.10.27 3.25.1 (2022-03-11)

- Improved display of type information and parameters in the *API reference documentation*. (#413)

1.10.28 3.25 (2022-03-01)

- New `hash_id_columns=` parameter for creating a primary key that's a hash of the content of specific columns - see *Setting an ID based on the hash of the row contents* for details. (#343)
- New *db.sqlite_version* property, returning a tuple of integers representing the version of SQLite, for example `(3, 38, 0)`.
- Fixed a bug where *register_function(deterministic=True)* caused errors on versions of SQLite prior to 3.8.3. (#408)
- New documented *hash_record(record, keys=...)* function.

1.10.29 3.24 (2022-02-15)

- SpatiaLite helpers for the `sqlite-utils` command-line tool - thanks, Chris Amico. (#398)
  - *sqlite-utils create-database* `--init-spatialite` option for initializing SpatiaLite on a newly created database.
  - *sqlite-utils add-geometry-column* command for adding geometry columns.
  - *sqlite-utils create-spatial-index* command for adding spatial indexes.
- `db[table].create(..., if_not_exists=True)` option for *creating a table* only if it does not already exist. (#397)
- `Database(memory_name="my_shared_database")` parameter for creating a *named in-memory database* that can be shared between multiple connections. (#405)
- Documentation now describes *how to add a primary key to a rowid table* using `sqlite-utils transform`. (#403)

1.10.30 3.23 (2022-02-03)

This release introduces four new utility methods for working with SpatiaLite. Thanks, Chris Amico. (#385)

- `sqlite_utils.utils.find_spatialite()` *finds the location of the SpatiaLite module* on disk.
- `db.init_spatialite()` *initializes SpatiaLite* for the given database.
- `table.add_geometry_column(...)` *adds a geometry column* to an existing table.
- `table.create_spatial_index(...)` *creates a spatial index* for a column.
- `sqlite-utils batch` now accepts a `--batch-size` option. (#392)

1.10.31 3.22.1 (2022-01-25)

- All commands now include example usage in their `--help` - see *CLI reference*. (#384)
- Python library documentation has a new *Getting started* section. (#387)
- Documentation now uses Plausible analytics. (#389)

1.10.32 3.22 (2022-01-11)

- New *CLI reference* documentation page, listing the output of `--help` for every one of the CLI commands. (#383)
- `sqlite-utils rows` now has `--limit` and `--offset` options for paginating through data. (#381)
- `sqlite-utils rows` now has `--where` and `-p` options for filtering the table using a `WHERE` query, see *Returning all rows in a table*. (#382)

1.10.33 3.21 (2022-01-10)

CLI and Python library improvements to help run ANALYZE after creating indexes or inserting rows, to gain better performance from the SQLite query planner when it runs against indexes.

Three new CLI commands: `create-database`, `analyze` and `bulk`.

More details and examples can be found in the annotated release notes.

- New `sqlite-utils create-database` command for creating new empty database files. (#348)
- New Python methods for running `ANALYZE` against a database, table or index: `db.analyze()` and `table.analyze()`, see *Optimizing index usage with ANALYZE*. (#366)
- New *sqlite-utils analyze command* for running `ANALYZE` using the CLI. (#379)
- The `create-index`, `insert` and `upsert` commands now have a new `--analyze` option for running `ANALYZE` after the command has completed. (#379)
- New *sqlite-utils bulk command* which can import records in the same way as `sqlite-utils insert` (from JSON, CSV or TSV) and use them to bulk execute a parametrized SQL query. (#375)
- The CLI tool can now also be run using `python -m sqlite_utils`. (#368)
- Using `--fmt` now implies `--table`, so you don't need to pass both options. (#374)
- The `--convert` function applied to rows can now modify the row in place. (#371)
- The *insert-files command* supports two new columns: `stem` and `suffix`. (#372)
- The `--nl` import option now ignores blank lines in the input. (#376)
- Fixed bug where streaming input to the `insert` command with `--batch-size 1` would appear to only commit after several rows had been ingested, due to unnecessary input buffering. (#364)

1.10.34 3.20 (2022-01-05)

- `sqlite-utils insert ... --lines` to insert the lines from a file into a table with a single `line` column, see *Inserting unstructured data with --lines and --text*.
- `sqlite-utils insert ... --text` to insert the contents of the file into a table with a single `text` column and a single row.
- `sqlite-utils insert ... --convert` allows a Python function to be provided that will be used to convert each row that is being inserted into the database. See *Applying conversions while inserting data*, including details on special behavior when combined with `--lines` and `--text`. (#356)
- `sqlite-utils convert` now accepts a code value of `-` to read code from standard input. (#353)

- `sqlite-utils convert` also now accepts code that defines a named `convert(value)` function, see *Converting data in columns*.
- `db.supports_strict` property showing if the database connection supports SQLite strict tables.
- `table.strict` property (see *.strict*) indicating if the table uses strict mode. (#344)
- Fixed bug where `sqlite-utils upsert ... --detect-types` ignored the `--detect-types` option. (#362)

1.10.35 3.19 (2021-11-20)

- The *table.lookup() method* now accepts keyword arguments that match those on the underlying `table.insert()` method: `foreign_keys=`, `column_order=`, `not_null=`, `defaults=`, `extracts=`, `conversions=` and `columns=`. You can also now pass `pk=` to specify a different column name to use for the primary key. (#342)

1.10.36 3.18 (2021-11-14)

- The `table.lookup()` method now has an optional second argument which can be used to populate columns only the first time the record is created, see *Working with lookup tables*. (#339)
- `sqlite-utils memory` now has a `--flatten` option for *flattening nested JSON objects* into separate columns, consistent with `sqlite-utils insert`. (#332)
- `table.create_index(..., find_unique_name=True)` parameter, which finds an available name for the created index even if the default name has already been taken. This means that `index-foreign-keys` will work even if one of the indexes it tries to create clashes with an existing index name. (#335)
- Added `py.typed` to the module, so mypy should now correctly pick up the type annotations. Thanks, Andreas Longo. (#331)
- Now depends on `python-dateutil` instead of depending on `dateutils`. Thanks, Denys Pavlov. (#324)
- `table.create()` (see *Explicitly creating a table*) now handles `dict`, `list` and `tuple` types, mapping them to `TEXT` columns in SQLite so that they can be stored encoded as JSON. (#338)
- Inserted data with square braces in the column names (for example a CSV file containing a `item[price]`) column now have the braces converted to underscores: `item_price_`. Previously such columns would be rejected with an error. (#329)
- Now also tested against Python 3.10. (#330)

1.10.37 3.17.1 (2021-09-22)

- *sqlite-utils memory* now works if files passed to it share the same file name. (#325)
- *sqlite-utils query* now returns `[]` in JSON mode if no rows are returned. (#328)

1.10.38 3.17 (2021-08-24)

- The *sqlite-utils memory* command has a new `--analyze` option, which runs the equivalent of the *analyze-tables* command directly against the in-memory database created from the incoming CSV or JSON data. (#320)
- *sqlite-utils insert-files* now has the ability to insert file contents in to `TEXT` columns in addition to the default `BLOB`. Pass the `--text` option or use `content_text` as a column specifier. (#319)

1.10.39 3.16 (2021-08-18)

- Type signatures added to more methods, including `table.resolve_foreign_keys()`, `db.create_table_sql()`, `db.create_table()` and `table.create()`. (#314)
- New `db.quote_fts(value)` method, see *Quoting characters for use in search* - thanks, Mark Neumann. (#246)
- `table.search()` now accepts an optional `quote=True` parameter. (#296)
- CLI command `sqlite-utils search` now accepts a `--quote` option. (#296)
- Fixed bug where `--no-headers` and `--tsv` options to *sqlite-utils insert* could not be used together. (#295)
- Various small improvements to *API reference* documentation.

1.10.40 3.15.1 (2021-08-10)

- Python library now includes type annotations on almost all of the methods, plus detailed docstrings describing each one. (#311)
- New *API reference* documentation page, powered by those docstrings.
- Fixed bug where `.add_foreign_keys()` failed to raise an error if called against a `View`. (#313)
- Fixed bug where `.delete_where()` returned a `[]` instead of returning `self` if called against a non-existent table. (#315)

1.10.41 3.15 (2021-08-09)

- `sqlite-utils insert --flatten` option for *flattening nested JSON objects* to create tables with column names like `topkey_nestedkey`. (#310)
- Fixed several spelling mistakes in the documentation, spotted using codespell.
- Errors that occur while using the `sqlite-utils` CLI tool now show the responsible SQL and query parameters, if possible. (#309)

1.10.42 3.14 (2021-08-02)

This release introduces the new *sqlite-utils convert command* (#251) and corresponding *table.convert(...)* Python method (#302). These tools can be used to apply a Python conversion function to one or more columns of a table, either updating the column in place or using transformed data from that column to populate one or more other columns.

This command-line example uses the Python standard library textwrap module to wrap the content of the `content` column in the `articles` table to 100 characters:

```
$ sqlite-utils convert content.db articles content \
    '"\n".join(textwrap.wrap(value, 100))' \
    --import=textwrap
```

The same operation in Python code looks like this:

```
import sqlite_utils, textwrap

db = sqlite_utils.Database("content.db")
db["articles"].convert("content", lambda v: "\n".join(textwrap.wrap(v, 100)))
```

See the full documentation for the *sqlite-utils convert command* and the *table.convert(...)* Python method for more details.

Also in this release:

- The new `table.count_where(...)` method, for counting rows in a table that match a specific SQL `WHERE` clause. (#305)
- New `--silent` option for the *sqlite-utils insert-files command* to hide the terminal progress bar, consistent with the `--silent` option for `sqlite-utils convert`. (#301)

1.10.43 3.13 (2021-07-24)

- `sqlite-utils schema my.db table1 table2` command now accepts optional table names. (#299)
- `sqlite-utils memory --help` now describes the `--schema` option.

1.10.44 3.12 (2021-06-25)

- New *db.query(sql, params)* method, which executes a SQL query and returns the results as an iterator over Python dictionaries. (#290)
- This project now uses `flake8` and has started to use `mypy`. (#291)
- New documentation on *contributing* to this project. (#292)

1.10.45 3.11 (2021-06-20)

- New `sqlite-utils memory data.csv --schema` option, for outputting the schema of the in-memory database generated from one or more files. See *--schema, --analyze, --dump and --save*. (#288)
- Added *installation instructions*. (#286)

1.10.46 3.10 (2021-06-19)

This release introduces the `sqlite-utils memory` command, which can be used to load CSV or JSON data into a temporary in-memory database and run SQL queries (including joins across multiple files) directly against that data.

Also new: `sqlite-utils insert --detect-types`, `sqlite-utils dump`, `table.use_rowid` plus some smaller fixes.

sqlite-utils memory

This example of `sqlite-utils memory` retrieves information about the all of the repositories in the Dogsheep organization on GitHub using this JSON API, sorts them by their number of stars and outputs a table of the top five (using `-t`):

```
$ curl -s 'https://api.github.com/users/dogsheep/repos' \
  | sqlite-utils memory - '
      select full_name, forks_count, stargazers_count
      from stdin order by stargazers_count desc limit 5
    ' -t
full_name                          forks_count    stargazers_count
-------------------------------  -------------  ------------------
dogsheep/twitter-to-sqlite                  12                 225
dogsheep/github-to-sqlite                   14                 139
dogsheep/dogsheep-photos                     5                 116
dogsheep/dogsheep.github.io                  7                  90
dogsheep/healthkit-to-sqlite                 4                  85
```

The tool works against files on disk as well. This example joins data from two CSV files:

```
$ cat creatures.csv
species_id,name
1,Cleo
2,Bants
2,Dori
2,Azi
$ cat species.csv
id,species_name
1,Dog
2,Chicken
$ sqlite-utils memory species.csv creatures.csv '
  select * from creatures join species on creatures.species_id = species.id
'
[{"species_id": 1, "name": "Cleo", "id": 1, "species_name": "Dog"},
 {"species_id": 2, "name": "Bants", "id": 2, "species_name": "Chicken"},
 {"species_id": 2, "name": "Dori", "id": 2, "species_name": "Chicken"},
 {"species_id": 2, "name": "Azi", "id": 2, "species_name": "Chicken"}]
```

Here the `species.csv` file becomes the `species` table, the `creatures.csv` file becomes the `creatures` table and the output is JSON, the default output format.

You can also use the `--attach` option to attach existing SQLite database files to the in-memory database, in order to join data from CSV or JSON directly against your existing tables.

Full documentation of this new feature is available in *Querying data directly using an in-memory database*. (#272)

sqlite-utils insert --detect-types

The *sqlite-utils insert* command can be used to insert data from JSON, CSV or TSV files into a SQLite database file. The new `--detect-types` option (shortcut `-d`), when used in conjunction with a CSV or TSV import, will automatically detect if columns in the file are integers or floating point numbers as opposed to treating everything as a text column and create the new table with the corresponding schema. See *Inserting CSV or TSV data* for details. (#282)

Other changes

- **Bug fix**: `table.transform()`, when run against a table without explicit primary keys, would incorrectly create a new version of the table with an explicit primary key column called `rowid`. (#284)
- New `table.use_rowid` introspection property, see *.use_rowid*. (#285)
- The new `sqlite-utils dump file.db` command outputs a SQL dump that can be used to recreate a database. (#274)
- `-h` now works as a shortcut for `--help`, thanks Loren McIntyre. (#276)
- Now using pytest-cov and Codecov to track test coverage - currently at 96%. (#275)
- SQL errors that occur when using `sqlite-utils query` are now displayed as CLI errors.

1.10.47 3.9.1 (2021-06-12)

- Fixed bug when using `table.upsert_all()` to create a table with only a single column that is treated as the primary key. (#271)

1.10.48 3.9 (2021-06-11)

- New `sqlite-utils schema` command showing the full SQL schema for a database, see *Showing the schema (CLI)*. (#268)
- `db.schema` introspection property exposing the same feature to the Python library, see *Showing the schema (Python library)*.

1.10.49 3.8 (2021-06-02)

- New `sqlite-utils indexes` command to list indexes in a database, see *Listing indexes*. (#263)
- `table.xindexes` introspection property returning more details about that table's indexes, see *.xindexes*. (#261)

1.10.50 3.7 (2021-05-28)

- New `table.pks_and_rows_where()` method returning `(primary_key, row_dictionary)` tuples - see *Listing rows with their primary keys*. (#240)
- Fixed bug with `table.add_foreign_key()` against columns containing spaces. (#238)
- `table_or_view.drop(ignore=True)` option for avoiding errors if the table or view does not exist. (#237)
- `sqlite-utils drop-view --ignore` and `sqlite-utils drop-table --ignore` options. (#237)
- Fixed a bug with inserts of nested JSON containing non-ascii strings - thanks, Dylan Wu. (#257)
- Suggest `--alter` if an error occurs caused by a missing column. (#259)
- Support creating indexes with columns in descending order, see *API documentation* and *CLI documentation*. (#260)
- Correctly handle CSV files that start with a UTF-8 BOM. (#250)

1.10.51 3.6 (2021-02-18)

This release adds the ability to execute queries joining data from more than one database file - similar to the cross database querying feature introduced in Datasette 0.55.

- The `db.attach(alias, filepath)` Python method can be used to attach extra databases to the same connection, see *db.attach() in the Python API documentation*. (#113)
- The `--attach` option attaches extra aliased databases to run SQL queries against directly on the command-line, see *attaching additional databases in the CLI documentation*. (#236)

1.10.52 3.5 (2021-02-14)

- `sqlite-utils insert --sniff` option for detecting the delimiter and quote character used by a CSV file, see *Alternative delimiters and quote characters*. (#230)
- The `table.rows_where()`, `table.search()` and `table.search_sql()` methods all now take optional `offset=` and `limit=` arguments. (#231)
- New `--no-headers` option for `sqlite-utils insert --csv` to handle CSV files that are missing the header row, see *CSV files without a header row*. (#228)
- Fixed bug where inserting data with extra columns in subsequent chunks would throw an error. Thanks @nieuwenhoven for the fix. (#234)
- Fixed bug importing CSV files with columns containing more than 128KB of data. (#229)
- Test suite now runs in CI against Ubuntu, macOS and Windows. Thanks @nieuwenhoven for the Windows test fixes. (#232)

1.10.53 3.4.1 (2021-02-05)

- Fixed a code import bug that slipped in to 3.4. (#226)

1.10.54 3.4 (2021-02-05)

- `sqlite-utils insert --csv` now accepts optional `--delimiter` and `--quotechar` options. See *Alternative delimiters and quote characters*. (#223)

1.10.55 3.3 (2021-01-17)

- The `table.m2m()` method now accepts an optional `alter=True` argument to specify that any missing columns should be added to the referenced table. See *Working with many-to-many relationships*. (#222)

1.10.56 3.2.1 (2021-01-12)

- Fixed a bug where `.add_missing_columns()` failed to take case insensitive column names into account. (#221)

1.10.57 3.2 (2021-01-03)

This release introduces a new mechanism for speeding up `count(*)` queries using cached table counts, stored in a `_counts` table and updated by triggers. This mechanism is described in *Cached table counts using triggers*, and can be enabled using Python API methods or the new `enable-counts` CLI command. (#212)

- `table.enable_counts()` method for enabling these triggers on a specific table.
- `db.enable_counts()` method for enabling triggers on every table in the database. (#213)
- New `sqlite-utils enable-counts my.db` command for enabling counts on all or specific tables, see *Enabling cached counts*. (#214)
- New `sqlite-utils triggers` command for listing the triggers defined for a database or specific tables, see *Listing triggers*. (#218)
- New `db.use_counts_table` property which, if `True`, causes `table.count` to read from the `_counts` table. (#215)
- `table.has_counts_triggers` property revealing if a table has been configured with the new `_counts` database triggers.
- `db.reset_counts()` method and `sqlite-utils reset-counts` command for resetting the values in the `_counts` table. (#219)
- The previously undocumented `db.escape()` method has been renamed to `db.quote()` and is now covered by the documentation: *Quoting strings for use in SQL*. (#217)
- New `table.triggers_dict` and `db.triggers_dict` introspection properties. (#211, #216)
- `sqlite-utils insert` now shows a more useful error message for invalid JSON. (#206)

1.10.58 3.1.1 (2021-01-01)

- Fixed failing test caused by `optimize` sometimes creating larger database files. (#209)
- Documentation now lives on https://sqlite-utils.datasette.io/
- README now includes `brew install sqlite-utils` installation method.

1.10.59 3.1 (2020-12-12)

- New command: `sqlite-utils analyze-tables my.db` outputs useful information about the table columns in the database, such as the number of distinct values and how many rows are null. See *Analyzing tables* for documentation. (#207)
- New `table.analyze_column(column)` Python method used by the `analyze-tables` command - see *Analyzing a column*.
- The `table.update()` method now correctly handles values that should be stored as JSON. Thanks, Andreas Madsack. (#204)

1.10.60 3.0 (2020-11-08)

This release introduces a new `sqlite-utils search` command for searching tables, see *Executing searches*. (#192)

The `table.search()` method has been redesigned, see *Searching with table.search()*. (#197)

The release includes minor backwards-incompatible changes, hence the version bump to 3.0. Those changes, which should not affect most users, are:

- The `-c` shortcut option for outputting CSV is no longer available. The full `--csv` option is required instead.
- The `-f` shortcut for `--fmt` has also been removed - use `--fmt`.
- The `table.search()` method now defaults to sorting by relevance, not sorting by `rowid`. (#198)
- The `table.search()` method now returns a generator over a list of Python dictionaries. It previously returned a list of tuples.

Also in this release:

- The `query`, `tables`, `rows` and `search` CLI commands now accept a new `--tsv` option which outputs the results in TSV. (#193)
- A new `table.virtual_table_using` property reveals if a table is a virtual table, and returns the upper case type of virtual table (e.g. `FTS4` or `FTS5`) if it is. It returns `None` if the table is not a virtual table. (#196)
- The new `table.search_sql()` method returns the SQL for searching a table, see *Building SQL queries with table.search_sql()*.
- `sqlite-utils rows` now accepts multiple optional `-c` parameters specifying the columns to return. (#200)

Changes since the 3.0a0 alpha release:

- The `sqlite-utils search` command now defaults to returning every result, unless you add a `--limit 20` option.
- The `sqlite-utils search -c` and `table.search(columns=[])` options are now fully respected. (#201)

1.10.61 2.23 (2020-10-28)

- `table.m2m(other_table, records)` method now takes any iterable, not just a list or tuple. Thanks, Adam Wolf. (#189)
- `sqlite-utils insert` now displays a progress bar for CSV or TSV imports. (#173)
- New `@db.register_function(deterministic=True)` option for registering deterministic SQLite functions in Python 3.8 or higher. (#191)

1.10.62 2.22 (2020-10-16)

- New `--encoding` option for processing CSV and TSV files that use a non-utf-8 encoding, for both the `insert` and `update` commands. (#182)
- The `--load-extension` option is now available to many more commands. (#137)
- `--load-extension=spatialite` can be used to load SpatiaLite from common installation locations, if it is available. (#136)
- Tests now also run against Python 3.9. (#184)
- Passing `pk=["id"]` now has the same effect as passing `pk="id"`. (#181)

1.10.63 2.21 (2020-09-24)

- `table.extract()` and `sqlite-utils extract` now apply much, much faster - one example operation reduced from twelve minutes to just four seconds! (#172)
- `sqlite-utils extract` no longer shows a progress bar, because it's fast enough not to need one.
- New `column_order=` option for `table.transform()` which can be used to alter the order of columns in a table. (#175)
- `sqlite-utils transform --column-order=` option (with a `-o` shortcut) for changing column order. (#176)
- The `table.transform(drop_foreign_keys=)` parameter and the `sqlite-utils transform --drop-foreign-key` option have changed. They now accept just the name of the column rather than requiring all three of the column, other table and other column. This is technically a backwards-incompatible change but I chose not to bump the major version number because the transform feature is so new. (#177)
- The table `.disable_fts()`, `.rebuild_fts()`, `.delete()`, `.delete_where()` and `.add_missing_columns()` methods all now `return self`, which means they can be chained together with other table operations.

1.10.64 2.20 (2020-09-22)

This release introduces two key new capabilities: **transform** (#114) and **extract** (#42).

Transform

SQLite's ALTER TABLE has several documented limitations. The `table.transform()` Python method and `sqlite-utils transform` CLI command work around these limitations using a pattern where a new table with the desired structure is created, data is copied over to it and the old table is then dropped and replaced by the new one.

You can use these tools to change column types, rename columns, drop columns, add and remove `NOT NULL` and defaults, remove foreign key constraints and more. See the *transforming tables (CLI)* and *transforming tables (Python library)* documentation for full details of how to use them.

Extract

Sometimes a database table - especially one imported from a CSV file - will contain duplicate data. A `Trees` table may include a `Species` column with only a few dozen unique values, when the table itself contains thousands of rows.

The `table.extract()` method and `sqlite-utils extract` commands can extract a column - or multiple columns - out into a separate lookup table, and set up a foreign key relationship from the original table.

The Python library *extract() documentation* describes how extraction works in detail, and *Extracting columns into a separate table* in the CLI documentation includes a detailed example.

Other changes

- The `@db.register_function` decorator can be used to quickly register Python functions as custom SQL functions, see *Registering custom SQL functions*. (#162)
- The `table.rows_where()` method now accepts an optional `select=` argument for specifying which columns should be selected, see *Listing rows*.

1.10.65 2.19 (2020-09-20)

- New `sqlite-utils add-foreign-keys` command for *Adding multiple foreign keys at once*. (#157)
- New `table.enable_fts(..., replace=True)` argument for replacing an existing FTS table with a new configuration. (#160)
- New `table.add_foreign_key(..., ignore=True)` argument for ignoring a foreign key if it already exists. (#112)

1.10.66 2.18 (2020-09-08)

- `table.rebuild_fts()` method for rebuilding a FTS index, see *Rebuilding a full-text search table*. (#155)
- `sqlite-utils rebuild-fts data.db` command for rebuilding FTS indexes across all tables, or just specific tables. (#155)
- `table.optimize()` method no longer deletes junk rows from the `*_fts_docsize` table. This was added in 2.17 but it turns out running `table.rebuild_fts()` is a better solution to this problem.
- Fixed a bug where rows with additional columns that are inserted after the first batch of records could cause an error due to breaking SQLite's maximum number of parameters. Thanks, Simon Wiles. (#145)

1.10.67 2.17 (2020-09-07)

This release handles a bug where replacing rows in FTS tables could result in growing numbers of unnecessary rows in the associated `*_fts_docsize` table. (#149)

- `PRAGMA recursive_triggers=on` by default for all connections. You can turn it off with `Database(recursive_triggers=False)`. (#152)
- `table.optimize()` method now deletes unnecessary rows from the `*_fts_docsize` table. (#153)
- New tracer method for tracking underlying SQL queries, see *Tracing queries*. (#150)
- Neater indentation for schema SQL. (#148)
- Documentation for `sqlite_utils.AlterError` exception thrown by in `add_foreign_keys()`.

1.10.68 2.16.1 (2020-08-28)

- `insert_all(..., alter=True)` now works for columns introduced after the first 100 records. Thanks, Simon Wiles! (#139)
- Continuous Integration is now powered by GitHub Actions. (#143)

1.10.69 2.16 (2020-08-21)

- `--load-extension` option for `sqlite-utils query` for loading SQLite extensions. (#134)
- New `sqlite_utils.utils.find_spatialite()` function for finding SpatiaLite in common locations. (#135)

1.10.70 2.15.1 (2020-08-12)

- Now available as a `sdist` package on PyPI in addition to a wheel. (#133)

1.10.71 2.15 (2020-08-10)

- New `db.enable_wal()` and `db.disable_wal()` methods for enabling and disabling Write-Ahead Logging for a database file - see *WAL mode* in the Python API documentation.
- Also `sqlite-utils enable-wal file.db` and `sqlite-utils disable-wal file.db` commands for doing the same thing on the command-line, see *WAL mode (CLI)*. (#132)

1.10.72 2.14.1 (2020-08-05)

- Documentation improvements.

1.10.73 2.14 (2020-08-01)

- The *insert-files command* can now read from standard input: `cat dog.jpg | sqlite-utils insert-files dogs.db pics - --name=dog.jpg`. (#127)
- You can now specify a full-text search tokenizer using the new `tokenize=` parameter to *enable_fts()*. This means you can enable Porter stemming on a table by running `db["articles"].enable_fts(["headline", "body"], tokenize="porter")`. (#130)
- You can also set a custom tokenizer using the *sqlite-utils enable-fts* CLI command, via the new `--tokenize` option.

1.10.74 2.13 (2020-07-29)

- `memoryview` and `uuid.UUID` objects are now supported. `memoryview` objects will be stored using `BLOB` and `uuid.UUID` objects will be stored using `TEXT`. (#128)

1.10.75 2.12 (2020-07-27)

The theme of this release is better tools for working with binary data. The new `insert-files` command can be used to insert binary files directly into a database table, and other commands have been improved with better support for BLOB columns.

- `sqlite-utils insert-files my.db gifs *.gif` can now insert the contents of files into a specified table. The columns in the table can be customized to include different pieces of metadata derived from the files. See *Inserting data from files*. (#122)
- `--raw` option to `sqlite-utils query` - for outputting just a single raw column value - see *Returning raw data, such as binary content*. (#123)
- JSON output now encodes BLOB values as special base64 objects - see *Returning JSON*. (#125)
- The same format of JSON base64 objects can now be used to insert binary data - see *Inserting JSON data*. (#126)
- The `sqlite-utils query` command can now accept named parameters, e.g. `sqlite-utils :memory: "select :num * :num2" -p num 5 -p num2 6` - see *Returning JSON*. (#124)

1.10.76 2.11 (2020-07-08)

- New `--truncate` option to `sqlite-utils insert`, and `truncate=True` argument to `.insert_all()`. Thanks, Thomas Sibley. (#118)
- The `sqlite-utils query` command now runs updates in a transaction. Thanks, Thomas Sibley. (#120)

1.10.77 2.10.1 (2020-06-23)

- Added documentation for the `table.pks` introspection property. (#116)

1.10.78 2.10 (2020-06-12)

- The `sqlite-utils` command now supports UPDATE/INSERT/DELETE in addition to SELECT. (#115)

1.10.79 2.9.1 (2020-05-11)

- Added custom project links to the PyPI listing.

1.10.80 2.9 (2020-05-10)

- New `sqlite-utils drop-table` command, see *Dropping tables*. (#111)
- New `sqlite-utils drop-view` command, see *Dropping views*.
- Python `decimal.Decimal` objects are now stored as `FLOAT`. (#110)

1.10.81 2.8 (2020-05-03)

- New `sqlite-utils create-table` command, see *Creating tables*. (#27)
- New `sqlite-utils create-view` command, see *Creating views*. (#107)

1.10.82 2.7.2 (2020-05-02)

- `db.create_view(...)` now has additional parameters `ignore=True` or `replace=True`, see *Creating views*. (#106)

1.10.83 2.7.1 (2020-05-01)

- New `sqlite-utils views my.db` command for listing views in a database, see *Listing views*. (#105)
- `sqlite-utils tables` (and `views`) has a new `--schema` option which outputs the table/view schema, see *Listing tables*. (#104)
- Nested structures containing invalid JSON values (e.g. Python bytestrings) are now serialized using `repr()` instead of throwing an error. (#102)

1.10.84 2.7 (2020-04-17)

- New `columns=` argument for the `.insert()`, `.insert_all()`, `.upsert()` and `.upsert_all()` methods, for over-riding the auto-detected types for columns and specifying additional columns that should be added when the table is created. See *Custom column order and column types*. (#100)

1.10.85 2.6 (2020-04-15)

- New `table.rows_where(..., order_by="age desc")` argument, see *Listing rows*. (#76)

1.10.86 2.5 (2020-04-12)

- Panda's Timestamp is now stored as a SQLite TEXT column. Thanks, b0b5h4rp13! (#96)
- `table.last_pk` is now only available for inserts or upserts of a single record. (#98)
- New `Database(filepath, recreate=True)` parameter for deleting and recreating the database. (#97)

1.10.87 2.4.4 (2020-03-23)

- Fixed bug where columns with only null values were not correctly created. (#95)

1.10.88 2.4.3 (2020-03-23)

- Column type suggestion code is no longer confused by null values. (#94)

1.10.89 2.4.2 (2020-03-14)

- `table.column_dicts` now works with all column types - previously it would throw errors on types other than `TEXT`, `BLOB`, `INTEGER` or `FLOAT`. (#92)
- Documentation for `NotFoundError` thrown by `table.get(pk)` - see *Retrieving a specific record*.

1.10.90 2.4.1 (2020-03-01)

- `table.enable_fts()` now works with columns that contain spaces. (#90)

1.10.91 2.4 (2020-02-26)

- `table.disable_fts()` can now be used to remove FTS tables and triggers that were created using `table.enable_fts(...)`. (#88)
- The `sqlite-utils disable-fts` command can be used to remove FTS tables and triggers from the command-line. (#88)
- Trying to create table columns with square braces ([or]) in the name now raises an error. (#86)
- Subclasses of `dict`, `list` and `tuple` are now detected as needing a JSON column. (#87)

1.10.92 2.3.1 (2020-02-10)

`table.create_index()` now works for columns that contain spaces. (#85)

1.10.93 2.3 (2020-02-08)

`table.exists()` is now a method, not a property. This was not a documented part of the API before so I'm considering this a non-breaking change. (#83)

1.10.94 2.2.1 (2020-02-06)

Fixed a bug where `.upsert(..., hash_id="pk")` threw an error (#84).

1.10.95 2.2 (2020-02-01)

New feature: `sqlite_utils.suggest_column_types([records])` returns the suggested column types for a list of records. See *Suggesting column types*. (#81).

This replaces the undocumented `table.detect_column_types()` method.

1.10.96 2.1 (2020-01-30)

New feature: `conversions={...}` can be passed to the `.insert()` family of functions to specify SQL conversions that should be applied to values that are being inserted or updated. See *Converting column values using SQL functions* . (#77).

1.10.97 2.0.1 (2020-01-05)

The `.upsert()` and `.upsert_all()` methods now raise a `sqlite_utils.db.PrimaryKeyRequired` exception if you call them without specifying the primary key column using `pk=` (#73).

1.10.98 2.0 (2019-12-29)

This release changes the behaviour of `upsert`. It's a breaking change, hence `2.0`.

The `upsert` command-line utility and the `.upsert()` and `.upsert_all()` Python API methods have had their behaviour altered. They used to completely replace the affected records: now, they update the specified values on existing records but leave other columns unaffected.

See *Upserting data using the Python API* and *Upserting data using the CLI* for full details.

If you want the old behaviour - where records were completely replaced - you can use `$ sqlite-utils insert ... --replace` on the command-line and `.insert(..., replace=True)` and `.insert_all(..., replace=True)` in the Python API. See *Insert-replacing data using the Python API* and *Insert-replacing data using the CLI* for more.

For full background on this change, see issue #66.

1.10.99 1.12.1 (2019-11-06)

- Fixed error thrown when `.insert_all()` and `.upsert_all()` were called with empty lists (#52)

1.10.100 1.12 (2019-11-04)

Python library utilities for deleting records (#62)

- `db["tablename"].delete(4)` to delete by primary key, see *Deleting a specific record*
- `db["tablename"].delete_where("id > ?", [3])` to delete by a where clause, see *Deleting multiple records*

1.10.101 1.11 (2019-09-02)

Option to create triggers to automatically keep FTS tables up-to-date with newly inserted, updated and deleted records. Thanks, Amjith Ramanujam! (#57)

- `sqlite-utils enable-fts ... --create-triggers` - see *Configuring full-text search using the CLI*
- `db["tablename"].enable_fts(..., create_triggers=True)` - see *Configuring full-text search using the Python library*
- Support for introspecting triggers for a database or table - see *Introspecting tables and views* (#59)

1.10.102 1.10 (2019-08-23)

Ability to introspect and run queries against views (#54)

- `db.view_names()` method and and `db.views` property
- Separate `View` and `Table` classes, both subclassing new `Queryable` class
- `view.drop()` method

See *Listing views*.

1.10.103 1.9 (2019-08-04)

- `table.m2m(...)` method for creating many-to-many relationships: *Working with many-to-many relationships* (#23)

1.10.104 1.8 (2019-07-28)

- `table.update(pk, values)` method: *Updating a specific record* (#35)

1.10.105 1.7.1 (2019-07-28)

- Fixed bug where inserting records with 11 columns in a batch of 100 triggered a "too many SQL variables" error (#50)
- Documentation and tests for `table.drop()` method: *Dropping a table or view*

1.10.106 1.7 (2019-07-24)

Support for lookup tables.

- New `table.lookup({...})` utility method for building and querying lookup tables - see *Working with lookup tables* (#44)
- New `extracts=` table configuration option, see *Populating lookup tables automatically during insert/upsert* (#46)
- Use pysqlite3 if it is available, otherwise use `sqlite3` from the standard library
- Table options can now be passed to the new `db.table(name, **options)` factory function in addition to being passed to `insert_all(records, **options)` and friends - see *Table configuration options*
- In-memory databases can now be created using `db = Database(memory=True)`

1.10.107 1.6 (2019-07-18)

- `sqlite-utils insert` can now accept TSV data via the new `--tsv` option (#41)

1.10.108 1.5 (2019-07-14)

- Support for compound primary keys (#36)
  - Configure these using the CLI tool by passing `--pk` multiple times
  - In Python, pass a tuple of columns to the `pk=(..., ...)` argument: *Compound primary keys*
- New `table.get()` method for retrieving a record by its primary key: *Retrieving a specific record* (#39)

1.10.109 1.4.1 (2019-07-14)

- Assorted minor documentation fixes: changes since 1.4

1.10.110 1.4 (2019-06-30)

- Added `sqlite-utils index-foreign-keys` command (*docs*) and `db.index_foreign_keys()` method (*docs*) (#33)

1.10.111 1.3 (2019-06-28)

- New mechanism for adding multiple foreign key constraints at once: *db.add_foreign_keys() documentation* (#31)

1.10.112 1.2.2 (2019-06-25)

- Fixed bug where `datetime.time` was not being handled correctly

1.10.113 1.2.1 (2019-06-20)

- Check the column exists before attempting to add a foreign key (#29)

1.10.114 1.2 (2019-06-12)

- Improved foreign key definitions: you no longer need to specify the `column`, `other_table` AND `other_column` to define a foreign key - if you omit the `other_table` or `other_column` the script will attempt to guess the correct values by introspecting the database. See *Adding foreign key constraints* for details. (#25)
- Ability to set `NOT NULL` constraints and `DEFAULT` values when creating tables (#24). Documentation: *Setting defaults and not null constraints (Python API)*, *Setting defaults and not null constraints (CLI)*
- Support for `not_null_default=X` / `--not-null-default` for setting a `NOT NULL DEFAULT 'x'` when adding a new column. Documentation: *Adding columns (Python API)*, *Adding columns (CLI)*

1.10.115 1.1 (2019-05-28)

- Support for `ignore=True` / `--ignore` for ignoring inserted records if the primary key already exists (#21) - documentation: *Inserting data (Python API)*, *Inserting data (CLI)*
- Ability to add a column that is a foreign key reference using `fk=...` / `--fk` (#16) - documentation: *Adding columns (Python API)*, *Adding columns (CLI)*

1.10.116 1.0.1 (2019-05-27)

- `sqlite-utils rows data.db table --json-cols` - fixed bug where `--json-cols` was not obeyed

1.10.117 1.0 (2019-05-24)

- **Option to automatically add new columns if you attempt to insert or upsert data with extra fields:**
  `sqlite-utils insert ... --alter` - see *Adding columns automatically with the sqlite-utils CLI*

  `db["tablename"].insert(record, alter=True)` - see *Adding columns automatically using the Python API*
- New `--json-cols` option for outputting nested JSON, see *Nested JSON values*

1.10.118 0.14 (2019-02-24)

- Ability to create unique indexes: `db["mytable"].create_index(["name"], unique=True)`
- `db["mytable"].create_index(["name"], if_not_exists=True)`
- `$ sqlite-utils create-index mydb.db mytable col1 [col2...]`, see *Creating indexes*
- `table.add_column(name, type)` method, see *Adding columns*
- `$ sqlite-utils add-column mydb.db mytable nameofcolumn`, see *Adding columns* (CLI)
- `db["books"].add_foreign_key("author_id", "authors", "id")`, see *Adding foreign key constraints*

- `$ sqlite-utils add-foreign-key books.db books author_id authors id`, see *Adding foreign key constraints* (CLI)
- Improved (but backwards-incompatible) `foreign_keys=` argument to various methods, see *Specifying foreign keys*

1.10.119 0.13 (2019-02-23)

- New `--table` and `--fmt` options can be used to output query results in a variety of visual table formats, see *Table-formatted output*
- New `hash_id=` argument can now be used for *Setting an ID based on the hash of the row contents*
- Can now derive correct column types for numpy int, uint and float values
- `table.last_id` has been renamed to `table.last_rowid`
- `table.last_pk` now contains the last inserted primary key, if `pk=` was specified
- Prettier indentation in the `CREATE TABLE` generated schemas

1.10.120 0.12 (2019-02-22)

- Added `db[table].rows` iterator - see *Listing rows*
- Replaced `sqlite-utils json` and `sqlite-utils csv` with a new default subcommand called `sqlite-utils query` which defaults to JSON and takes formatting options `--nl`, `--csv` and `--no-headers` - see *Returning JSON* and *Returning CSV or TSV*
- New `sqlite-utils rows data.db name-of-table` command, see *Returning all rows in a table*
- `sqlite-utils table` command now takes options `--counts` and `--columns` plus the standard output format options, see *Listing tables*

1.10.121 0.11 (2019-02-07)

New commands for enabling FTS against a table and columns:

```
sqlite-utils enable-fts db.db mytable col1 col2
```

See *Configuring full-text search*.

1.10.122 0.10 (2019-02-06)

Handle `datetime.date` and `datetime.time` values.

New option for efficiently inserting rows from a CSV:

```
sqlite-utils insert db.db foo - --csv
```

1.10.123 0.9 (2019-01-27)

Improved support for newline-delimited JSON.

`sqlite-utils insert` has two new command-line options:

- `--nl` means "expect newline-delimited JSON". This is an extremely efficient way of loading in large amounts of data, especially if you pipe it into standard input.

- `--batch-size=1000` lets you increase the batch size (default is 100). A commit will be issued every X records. This also control how many initial records are considered when detecting the desired SQL table schema for the data.

In the Python API, the `table.insert_all(...)` method can now accept a generator as well as a list of objects. This will be efficiently used to populate the table no matter how many records are produced by the generator.

The `Database()` constructor can now accept a `pathlib.Path` object in addition to a string or an existing SQLite connection object.

1.10.124 0.8 (2019-01-25)

Two new commands: `sqlite-utils csv` and `sqlite-utils json`

These commands execute a SQL query and return the results as CSV or JSON. See *Returning CSV or TSV* and *Returning JSON* for more details.

```
$ sqlite-utils json --help
Usage: sqlite-utils json [OPTIONS] PATH SQL

  Execute SQL query and return the results as JSON

Options:
  --nl      Output newline-delimited JSON
  --arrays  Output rows as arrays instead of objects
  --help    Show this message and exit.

$ sqlite-utils csv --help
Usage: sqlite-utils csv [OPTIONS] PATH SQL

  Execute SQL query and return the results as CSV

Options:
  --no-headers  Exclude headers from CSV output
  --help        Show this message and exit.
```

1.10.125 0.7 (2019-01-24)

This release implements the `sqlite-utils` command-line tool with a number of useful subcommands.

- `sqlite-utils tables demo.db` lists the tables in the database
- `sqlite-utils tables demo.db --fts4` shows just the FTS4 tables
- `sqlite-utils tables demo.db --fts5` shows just the FTS5 tables
- `sqlite-utils vacuum demo.db` runs VACUUM against the database
- `sqlite-utils optimize demo.db` runs OPTIMIZE against all FTS tables, then VACUUM
- `sqlite-utils optimize demo.db --no-vacuum` runs OPTIMIZE but skips VACUUM

The two most useful subcommands are `upsert` and `insert`, which allow you to ingest JSON files with one or more records in them, creating the corresponding table with the correct columns if it does not already exist. See *Inserting JSON data* for more details.

- `sqlite-utils insert demo.db dogs dogs.json --pk=id` inserts new records from `dogs.json` into the `dogs` table

- `sqlite-utils upsert demo.db dogs dogs.json --pk=id` upserts records, replacing any records with duplicate primary keys

One backwards incompatible change: the `db["table"].table_names` property is now a method:

- `db["table"].table_names()` returns a list of table names
- `db["table"].table_names(fts4=True)` returns a list of just the FTS4 tables
- `db["table"].table_names(fts5=True)` returns a list of just the FTS5 tables

A few other changes:

- Plenty of updated documentation, including full coverage of the new command-line tool
- Allow column names to be reserved words (use correct SQL escaping)
- Added automatic column support for bytes and datetime.datetime

1.10.126 0.6 (2018-08-12)

- `.enable_fts()` now takes optional argument `fts_version`, defaults to `FTS5`. Use `FTS4` if the version of SQLite bundled with your Python does not support FTS5
- New optional `column_order=` argument to `.insert()` and friends for providing a partial or full desired order of the columns when a database table is created
- *New documentation* for `.insert_all()` and `.upsert()` and `.upsert_all()`

1.10.127 0.5 (2018-08-05)

- `db.tables` and `db.table_names` introspection properties
- `db.indexes` property for introspecting indexes
- `table.create_index(columns, index_name)` method
- `db.create_view(name, sql)` method
- Table methods can now be chained, plus added `table.last_id` for accessing the last inserted row ID

1.10.128 0.4 (2018-07-31)

- `enable_fts()`, `populate_fts()` and `search()` table methods

1.10.129 0.3.1 (2018-07-31)

- Documented related projects
- Added badges to the documentation

1.10.130 0.3 (2018-07-31)

- New `Table` class representing a table in the SQLite database

1.10.131 0.2 (2018-07-28)

- Initial release to PyPI

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X