# sqlite-utils documentation

*Release 2.13*

**Simon Willison**

**Jul 30, 2020**

# Contents


**1 Contents** **3**

1.1 sqlite-utils command-line tool . . . 3

1.1.1 Running queries and returning JSON . . . 3

1.1.2 Running queries and returning CSV . . . 5

1.1.3 Running queries and outputting a table . . . 5

1.1.4 Returning raw data from a query, such as binary content . . . 6

1.1.5 Returning all rows in a table . . . 6

1.1.6 Listing tables . . . 6

1.1.7 Listing views . . . 7

1.1.8 Inserting JSON data . . . 7

1.1.9 Inserting CSV or TSV data . . . 9

1.1.10 Insert-replacing data . . . 9

1.1.11 Upserting data . . . 9

1.1.12 Inserting binary data from files . . . 10

1.1.13 Creating tables . . . 11

1.1.14 Dropping tables . . . 12

1.1.15 Creating views . . . 12

1.1.16 Dropping views . . . 12

1.1.17 Adding columns . . . 13

1.1.18 Adding columns automatically on insert/update . . . 13

1.1.19 Adding foreign key constraints . . . 13

1.1.20 Setting defaults and not null constraints . . . 14

1.1.21 Creating indexes . . . 14

1.1.22 Configuring full-text search . . . 14

1.1.23 Vacuum . . . 15

1.1.24 Optimize . . . 15

1.2 Python API . . . 15

1.2.1 Connecting to or creating a database . . . 15

1.2.2 Listing tables . . . 16

1.2.3 Listing views . . . 16

1.2.4 Listing rows . . . 16

1.2.5 Retrieving a specific record . . . 17

1.2.6 Creating tables . . . 17

1.2.7 Table configuration options . . . 20

1.2.8 Setting defaults and not null constraints . . . 20

1.2.9 Bulk inserts . . . 21

1.2.10 Insert-replacing data . . . 22
1.2.11 Updating a specific record . . . 22
1.2.12 Deleting a specific record . . . 23
1.2.13 Deleting multiple records . . . 23
1.2.14 Upserting data . . . 23
1.2.15 Working with lookup tables . . . 24
1.2.16 Working with many-to-many relationships . . . 25
1.2.17 Adding columns . . . 26
1.2.18 Adding columns automatically on insert/update . . . 27
1.2.19 Adding foreign key constraints . . . 28
1.2.20 Dropping a table or view . . . 29
1.2.21 Setting an ID based on the hash of the row contents . . . 29
1.2.22 Creating views . . . 29
1.2.23 Storing JSON . . . 30
1.2.24 Converting column values using SQL functions . . . 30
1.2.25 Introspection . . . 31
1.2.26 Enabling full-text search . . . 33
1.2.27 Optimizing a full-text search table . . . 34
1.2.28 Creating indexes . . . 34
1.2.29 Vacuum . . . 34
1.2.30 Suggesting column types . . . 35
1.3 Changelog . . . 36
1.3.1 2.13 (2020-07-29) . . . 36
1.3.2 2.12 (2020-07-27) . . . 36
1.3.3 2.11 (2020-07-08) . . . 36
1.3.4 2.10.1 (2020-06-23) . . . 36
1.3.5 2.10 (2020-06-12) . . . 37
1.3.6 2.9.1 (2020-05-11) . . . 37
1.3.7 2.9 (2020-05-10) . . . 37
1.3.8 2.8 (2020-05-03) . . . 37
1.3.9 2.7.2 (2020-05-02) . . . 37
1.3.10 2.7.1 (2020-05-01) . . . 37
1.3.11 2.7 (2020-04-17) . . . 37
1.3.12 2.6 (2020-04-15) . . . 37
1.3.13 2.5 (2020-04-12) . . . 38
1.3.14 2.4.4 (2020-03-23) . . . 38
1.3.15 2.4.3 (2020-03-23) . . . 38
1.3.16 2.4.2 (2020-03-14) . . . 38
1.3.17 2.4.1 (2020-03-01) . . . 38
1.3.18 2.4 (2020-02-26) . . . 38
1.3.19 2.3.1 (2020-02-10) . . . 38
1.3.20 2.3 (2020-02-08) . . . 38
1.3.21 2.2.1 (2020-02-06) . . . 39
1.3.22 2.2 (2020-02-01) . . . 39
1.3.23 2.1 (2020-01-30) . . . 39
1.3.24 2.0.1 (2020-01-05) . . . 39
1.3.25 2.0 (2019-12-29) . . . 39
1.3.26 1.12.1 (2019-11-06) . . . 39
1.3.27 1.12 (2019-11-04) . . . 39
1.3.28 1.11 (2019-09-02) . . . 40
1.3.29 1.10 (2019-08-23) . . . 40
1.3.30 1.9 (2019-08-04) . . . 40
1.3.31 1.8 (2019-07-28) . . . 40
1.3.32 1.7.1 (2019-07-28) . . . 40

1.3.33 1.7 (2019-07-24) . . . 40
1.3.34 1.6 (2019-07-18) . . . 41
1.3.35 1.5 (2019-07-14) . . . 41
1.3.36 1.4.1 (2019-07-14) . . . 41
1.3.37 1.4 (2019-06-30) . . . 41
1.3.38 1.3 (2019-06-28) . . . 41
1.3.39 1.2.2 (2019-06-25) . . . 41
1.3.40 1.2.1 (2019-06-20) . . . 41
1.3.41 1.2 (2019-06-12) . . . 41
1.3.42 1.1 (2019-05-28) . . . 42
1.3.43 1.0.1 (2019-05-27) . . . 42
1.3.44 1.0 (2019-05-24) . . . 42
1.3.45 0.14 (2019-02-24) . . . 42
1.3.46 0.13 (2019-02-23) . . . 42
1.3.47 0.12 (2019-02-22) . . . 43
1.3.48 0.11 (2019-02-07) . . . 43
1.3.49 0.10 (2019-02-06) . . . 43
1.3.50 0.9 (2019-01-27) . . . 43
1.3.51 0.8 (2019-01-25) . . . 44
1.3.52 0.7 (2019-01-24) . . . 44
1.3.53 0.6 (2018-08-12) . . . 45
1.3.54 0.5 (2018-08-05) . . . 45
1.3.55 0.4 (2018-07-31) . . . 45

*Python utility functions 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.

CHAPTER 1

# Contents

## 1.1 sqlite-utils command-line tool

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

### 1.1.1 Running queries and returning JSON

You can execute a SQL query against a database and get the results back as JSON like this:

```
$ sqlite-utils query dogs.db "select * from dogs"
[{"id": 1, "age": 4, "name": "Cleo"},
 {"id": 2, "age": 2, "name": "Pancakes"}]
```

This is the default command for `sqlite-utils`, so you can instead use this:

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

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}]
```

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"}
```

You can use `--arrays` to request ararys 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"
    }
]
```

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

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

```
$ sqlite-utils dogs.db "update dogs set age = 5 where name = 'Cleo'"
[{"rows_affected": 1}]
```

You can run queries against a temporary in-memory database by passing `:memory:` as the filename:

```
$ sqlite-utils :memory: "select sqlite_version()"
[{"sqlite_version()": "3.29.0"}]
```

### 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\"}]"
```

(continues on next page)

(continued from previous page)

```
    }
]
```

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"
            }
        ]
    }
]
```

### 1.1.2 Running queries and returning CSV

You can use the `--csv` option (or `-c` shortcut) 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
```

### 1.1.3 Running queries and outputting a table

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` (or `-f`) option to specify different table formats, for example `rst` for reStructuredText:

```
$ sqlite-utils dogs.db "select * from dogs" --table --fmt rst
====  =====  ========
  id    age  name
====  =====  ========
```

(continues on next page)

(continued from previous page)

```
   1       4  Cleo
   2       2  Pancakes
====  =====  ========
```

For a full list of table format options, run `sqlite-utils query --help`.

### 1.1.4 Returning raw data from a query, 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
```

### 1.1.5 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`, `--no-headers`, `--table` and `--fmt`.

### 1.1.6 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` option:

```
$ 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)
dogs     CREATE TABLE [dogs] (
           [id] INTEGER,
           [age] INTEGER,
           [name] TEXT)
```

The `--nl`, `--csv` and `--table` options are all available.

### 1.1.7 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`
- `--table`

### 1.1.8 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
    },
    {
        "id": 3,
        "name": "Toby",
        "age": 6
    }
]
```

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"
        }
    }
]
```

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
```

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

```
$ sqlite-utils insert dogs.db dogs dogs.json --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 also import newline-delimited JSON using the `--nl` option. Since Datasette can export newline-delimited JSON, you can combine the two tools 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
```

This also means you pipe `sqlite-utils` together to easily 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
# This creates saved.db with a single table called trees:
$ 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
```

### 1.1.9 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 docs.csv --csv
```

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

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

### 1.1.10 Insert-replacing data

Insert-replacing works exactly like inserting, with the exception that if your data has a primary key that matches an already existing record that record will be replaced with the new data.

After running the above `dogs.json` example, try running this:

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

This will replace the record for id=2 (Pancakes) with a new record with an updated age.

### 1.1.11 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 id=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.

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.

### 1.1.12 Inserting binary data from files

SQLite `BLOB` columns can be used to store binary content. It can be useful to insert the contents of files into a SQLite table.

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

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
);
```

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] FLOAT
);
```

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

**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

### 1.1.13 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`.

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])
         )
```

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`.

### 1.1.14 Dropping tables

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

```
$ sqlite-utils drop-table mytable
```

### 1.1.15 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.

### 1.1.16 Dropping views

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

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

### 1.1.17 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. You can use one of `text`, `integer`, `float` or `blob`. If you leave it off, `text` will be used.

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
```

### 1.1.18 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
```

### 1.1.19 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
```

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

#### 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
```

### 1.1.20 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
```

### 1.1.21 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.

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

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

### 1.1.23 Vacuum

You can run VACUUM to optimize your database like so:

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

### 1.1.24 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 our 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
```

## 1.2 Python API

### 1.2.1 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"))
```

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

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

Tables are accessed using the indexing operator, like so:

```
table = db["my_table"]
```

If the table does not yet exist, it will be created the first time you attempt to insert or upsert data into it.

You can also access tables using the `.table()` method like so:

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

Using this factory function allows you to set *Table configuration options*.

### 1.2.2 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>]
```

### 1.2.3 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)`
- `drop()`

### 1.2.4 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["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["dogs"].rows_where("age > ?", [3]):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
```

To specify an order, use the `order_by=` argument:

```
>>> for row in db["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["dogs"].rows_where(order_by="age desc"):
...     print(row)
{'id': 1, 'age': 4, 'name': 'Cleo'}
{'id': 2, 'age': 2, 'name': 'Pancakes'}
```

### 1.2.5 Retrieving a specific record

You can retrieve a record by its primary key using `table.get()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db["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["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["dogs"].get(5)
except NotFoundError:
    print("Dog not found")
```

### 1.2.6 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(sqlite3.connect("/tmp/dogs.db"))
dogs = db["dogs"]
dogs.insert({
    "name": "Cleo",
```

(continues on next page)

(continued from previous page)

```
    "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:

```
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:

```
dogs.insert({
    "id": 1,
```

(continues on next page)

(continued from previous page)

```
    "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] FLOAT
)
```

### Explicitly creating a table

You can directly create a new table without inserting any data into it using the `.create()` method:

```
db["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.

### 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["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["authors"].insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
], pk="id")
db["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")
])
```

### 1.2.7 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_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`, `upsert`, `batch_size`, `hash_id`, `alter`, `ignore`. These are all documented below.

### 1.2.8 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["authors"].insert_all(
    [{"id": 1, "name": "Sally", "score": 2}],
    pk="id",
    not_null={"name", "score"},
    defaults={"score": 1},
)
db["authors"].insert({"name": "Dharma"})

list(db["authors"].rows)
# Outputs:
# [{'id': 1, 'name': 'Sally', 'score': 2},
#  {'id': 3, 'name': 'Dharma', 'score': 1}]
print(db["authors"].schema)
↪                                                          # Outputs:
# CREATE TABLE [authors] (
#     [id] INTEGER PRIMARY KEY,
#     [name] TEXT NOT NULL,
#     [score] INTEGER NOT NULL DEFAULT 1
# )
```

### 1.2.9 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:

```
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", 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 or missing columns in subsequent batches will be ignored.

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["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.

### 1.2.10 Insert-replacing data

If you want to insert a record or replace an existing record with the same primary key, using the `replace=True` argument to `.insert()` or `.insert_all()`:

```
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.x the `.upsert()` and `.upsert_all()` methods did this. See *Upserting data* for the new behaviour of those methods in 2.x.

### 1.2.11 Updating a specific record

You can update a record by its primary key using `table.update()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> print(db["dogs"].get(1))
{'id': 1, 'age': 4, 'name': 'Cleo'}
>>> db["dogs"].update(1, {"age": 5})
>>> print(db["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["compound_dogs"].update((5, 3), {"name": "Updated"})
```

The second argument is a dictonary 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["dogs"].update(1, {"breed": "Mutt"}, alter=True)
```

### 1.2.12 Deleting a specific record

You can delete a record using `table.delete()`:

```
>>> db = sqlite_utils.Database("dogs.db")
>>> db["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["compound_dogs"].delete((5, 3))
```

### 1.2.13 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["dogs"].delete_where("age < ?", [3]):
```

Calling `table.delete_where()` with no other arguments will delete every row in the table.

### 1.2.14 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:

```
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.

**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.

### 1.2.15 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["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["Trees"].insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": db["Species"].lookup({
        "common_name": "Common Juniper",
        "latin_name": "Juniperus communis"
    })
})
```

#### 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"
})
```

(continues on next page)

(continued from previous page)

```
# If you want the table to be called 'species', you can do this:
trees = db.table("Trees", extracts=["species"])

# Using .insert() directly
db["Trees"].insert({
    "latitude": 49.1265976,
    "longitude": 2.5496218,
    "species": "Common Juniper"
}, extracts={"species": "Species"})
```

### 1.2.16 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["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["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=`.

### 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["dogs"].rows)
[{'id': 1, 'name': 'Cleo'}]
>>> list(db["characteristics"].rows)
[{'id': 1, 'name': 'Playful'}, {'id': 2, 'name': 'Opinionated'}]
>>> list(db["characteristics_dogs"].rows)
[{'characteristics_id': 1, 'dogs_id': 1}, {'characteristics_id': 2, 'dogs_id': 1}]
>>> print(db["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])
)
```

## 1.2.17 Adding columns

You can add a new column to a table using the `.add_column(col_name, col_type)` method:

```
db["dogs"].add_column("instagram", str)
db["dogs"].add_column("weight", float)
db["dogs"].add_column("dob", datetime.date)
db["dogs"].add_column("image", "BLOB")
db["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"` or `"BLOB"`.

If you pass a Python type, it will be mapped to SQLite types as shown here:

```
float: "FLOAT"
int: "INTEGER"
bool: "INTEGER"
str: "TEXT"
bytes: "BLOB"
datetime.datetime: "TEXT"
datetime.date: "TEXT"
datetime.time: "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: "FLOAT"
np.float32: "FLOAT"
np.float64: "FLOAT"
```

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

```
db["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["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["dogs"].add_column("friends_count", int, not_null_default=0)
```

### 1.2.18 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["new_table"].insert({"name": "Gareth"})
# This will throw an exception:
db["new_table"].insert({"name": "Gareth", "age": 32})
# This will succeed and add a new "age" integer column:
db["new_table"].insert({"name": "Gareth", "age": 32}, alter=True)
# You can see confirm the new column like so:
print(db["new_table"].columns_dict)
# Outputs this:
# {'name': <class 'str'>, 'age': <class 'int'>}
```

(continues on next page)

(continued from previous page)

```
# This works too:
new_table = db.table("new_table", alter=True)
new_table.insert({"name": "Gareth", "age": 32, "shoe_size": 11})
```

### 1.2.19 Adding foreign key constraints

The SQLite `ALTER TABLE` statement doesn't have the ability to add foreign key references to an existing column.

It's possible to add these references through very careful manipulation of SQLite's `sqlite_master` table, using `PRAGMA writable_schema`.

`sqlite-utils` can do this for you, though there is a significant risk of data corruption if something goes wrong so it is advisable to create a fresh copy of your database file before attempting this.

Here's an example of this mechanism in action:

```
db["authors"].insert_all([
    {"id": 1, "name": "Sally"},
    {"id": 2, "name": "Asheesh"}
], pk="id")
db["books"].insert_all([
    {"title": "Hedgehogs of the world", "author_id": 1},
    {"title": "How to train your wolf", "author_id": 2},
])
db["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 ommit 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`

#### Adding multiple foreign key constraints at once

The final step in adding a new foreign key to a SQLite database is to run `VACUUM`, to ensure the new foreign key is available in future introspection queries.

`VACUUM` against a large (multi-GB) database can take several minutes or longer. If you are adding multiple foreign keys using `table.add_foreign_key(...)` these can quickly add up.

Instead, you can use `db.add_foreign_keys(...)` to add multiple foreign keys within a single transaction. 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")
])
```

#### 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()
```

### 1.2.20 Dropping a table or view

You can drop a table or view using the `.drop()` method:

```
db["my_table"].drop()
```

### 1.2.21 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["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["dogs"].upsert({
    "name": "Cleo",
    "twitter": "cleopaws"
}, hash_id="id").last_pk
# dog_id is now "f501265970505d9825d8d9f590bfab3519fb20b1"
```

### 1.2.22 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)
```

### 1.2.23 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["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]
    },
    "address": {
        "streetAddress": "5497 Highway 9",
        "addressLocality": "Felton, CA",
        "postalCode": "95018"
    }
})
db.conn.execute("""
    select json_extract(address, '$.addressLocality')
    from niche_museums
""").fetchall()
# Returns [('Felton, CA',)]
```

### 1.2.24 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["example"].insert({
    "name": "The Bigfoot Discovery Museum"
}, conversions={"name": "upper(?)"})

# list(db["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 requests

# Open a database and load the SpatiaLite extension:
import sqlite3

conn = sqlite3.connect("places.db")
conn.enable_load_extension(True)
conn.load_extension("/usr/local/lib/mod_spatialite.dylib")

# Use sqlite-utils to create a places table:
db = sqlite_utils.Database(conn)
places = db["places"].create({"id": int, "name": str,})

# Add a SpatiaLite 'geometry' column:
db.conn.execute("select InitSpatialMetadata(1)")
db.conn.execute(
    "SELECT AddGeometryColumn('places', 'geometry', 4326, 'MULTIPOLYGON', 2);"
)

# Fetch some GeoJSON from Who's On First:
geojson = requests.get(
    "https://data.whosonfirst.org/404/227/475/404227475.geojson"
).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["places"].insert(
    {"name": "Wales", "geometry": wkt},
    conversions={"geometry": "GeomFromText(?, 4326)"},
)
```

### 1.2.25 Introspection

If you have loaded an existing table or view, you can use introspection to find out more about it:

```
>>> db["PlantType"]
<Table PlantType (id, value)>
```

The `.exists()` method can be used to find out if a table exists or not:

```
>>> db["PlantType"].exists()
True
>>> db["PlantType2"].exists()
False
```

The `.count` property shows the current number of rows (`select count(*) from table`):

```
>>> db["PlantType"].count
3
>>> db["Street_Tree_List"].count
189144
```

The `.columns` property shows the columns in the table or view:

```
>>> db["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)]
```

The `.columns_dict` property returns a dictionary version of this with just the names and types:

```
>>> db["PlantType"].columns_dict
{'id': <class 'int'>, 'value': <class 'str'>}
```

The `.pks` property returns a list of strings naming the primary key columns for the table:

```
>>> db["PlantType"].pks
['id']
```

The `.foreign_keys` property shows if the table has any foreign key relationships. It is not available on views.

```
>>> db["Street_Tree_List"].foreign_keys
[ForeignKey(table='Street_Tree_List', column='qLegalStatus', other_table='qLegalStatus
↪', other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qCareAssistant', other_table=
↪'qCareAssistant', other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qSiteInfo', other_table='qSiteInfo',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qSpecies', other_table='qSpecies',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='qCaretaker', other_table='qCaretaker',␣
↪other_column='id'),
 ForeignKey(table='Street_Tree_List', column='PlantType', other_table='PlantType',␣
↪other_column='id')]
```

The `.schema` property outputs the table's schema as a SQL string:

```
>>> print(db["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,
```

(continues on next page)

(continued from previous page)

```
  "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))
```

The `.indexes` property shows you all indexes created for a table. It is not available on views.

```
>>> db["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'])]
```

The `.triggers` property lists database triggers. It can be used on both database and table objects.

```
>>> db["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["authors"].triggers
```

### 1.2.26 Enabling full-text search

You can enable full-text search on a table using `.enable_fts(columns)`:

```
dogs.enable_fts(["name", "twitter"])
```

You can then run searches using the `.search()` method:

```
rows = dogs.search("cleo")
```

If you insert additional records into the table you will need to refresh the search index using `populate_fts()`:

```
dogs.insert({
    "id": 2,
    "name": "Marnie",
    "twitter": "MarnieTheDog",
    "age": 16,
```

(continues on next page)

(continued from previous page)

```
    "is_good_dog": True,
}, pk="id")
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`:

```
dogs.enable_fts(["name", "twitter"], create_triggers=True)
```

`.enable_fts()` defaults to using FTS5. If you wish to use FTS4 instead, use the following:

```
dogs.enable_fts(["name", "twitter"], fts_version="FTS4")
```

To remove the FTS tables and triggers you created, use the `disable_fts()` table method:

```
dogs.disable_fts()
```

### 1.2.27 Optimizing a full-text search table

Once you have populated a FTS table you can optimize it to dramatically reduce its size like so:

```
dogs.optimize()
```

This runs the following SQL:

```
INSERT INTO dogs_fts (dogs_fts) VALUES ("optimize");
```

### 1.2.28 Creating indexes

You can create an index on a table using the `.create_index(columns)` method. The method takes a list of columns:

```
dogs.create_index(["is_good_dog"])
```

By default the index will be named `idx_{table-name}_{columns}` - if you want to customize the name of the created index you can pass the `index_name` parameter:

```
dogs.create_index(
    ["is_good_dog", "age"],
    index_name="good_dogs_by_age"
)
```

You can create a unique index by passing `unique=True`:

```
dogs.create_index(["name"], unique=True)
```

Use `if_not_exists=True` to do nothing if an index with that name already exists.

### 1.2.29 Vacuum

You can optimize your database by running VACUUM against it like so:

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

### 1.2.30 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["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["cats"].create(types, pk="id")
# Insert the records
db["cats"].insert_all(cats)

# list(db["cats"].rows) now returns:
# [{"id": 1, "name": "Snowflake", "age": None, "thumbnail": None}
#  {"id": 2, "name": "Crabtree", "age": 4, "thumbnail": None}]
```

(continues on next page)

(continued from previous page)

```
# The table schema looks like this:
# print(db["cats"].schema)
# CREATE TABLE [cats] (
#    [id] INTEGER PRIMARY KEY,
#    [name] TEXT,
#    [age] INTEGER,
#    [thumbnail] BLOB
# )
```

## 1.3 Changelog

### 1.3.1 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)

### 1.3.2 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 binary data from files*. (#122)
- `--raw` option to `sqlite-utils query` - for outputting just a single raw column value - see *Returning raw data from a query, such as binary content*. (#123)
- JSON output now encodes BLOB values as special base64 obects - see *Running queries and 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 *Running queries and returning JSON*. (#124)

### 1.3.3 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)

### 1.3.4 2.10.1 (2020-06-23)

- Added documentation for the `table.pks` introspection property. (#116)

### 1.3.5 2.10 (2020-06-12)

- The `sqlite-utils` command now supports UPDATE/INSERT/DELETE in addition to SELECT. (#115)

### 1.3.6 2.9.1 (2020-05-11)

- Added custom project links to the PyPI listing.

### 1.3.7 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)

### 1.3.8 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)

### 1.3.9 2.7.2 (2020-05-02)

- `db.create_view(...)` now has additional parameters `ignore=True` or `replace=True`, see *Creating views*. (#106)

### 1.3.10 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)

### 1.3.11 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)

### 1.3.12 2.6 (2020-04-15)

- New `table.rows_where(..., order_by="age desc")` argument, see *Listing rows*. (#76)

### 1.3.13 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)

### 1.3.14 2.4.4 (2020-03-23)

- Fixed bug where columns with only null values were not correctly created. (#95)

### 1.3.15 2.4.3 (2020-03-23)

- Column type suggestion code is no longer confused by null values. (#94)

### 1.3.16 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*.

### 1.3.17 2.4.1 (2020-03-01)

- `table.enable_fts()` now works with columns that contain spaces. (#90)

### 1.3.18 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)

### 1.3.19 2.3.1 (2020-02-10)

`table.create_index()` now works for columns that contain spaces. (#85)

### 1.3.20 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)

### 1.3.21 2.2.1 (2020-02-06)

Fixed a bug where `.upsert(..., hash_id="pk")` threw an error (#84).

### 1.3.22 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.

### 1.3.23 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).

### 1.3.24 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).

### 1.3.25 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.

### 1.3.26 1.12.1 (2019-11-06)

- Fixed error thrown when `.insert_all()` and `.upsert_all()` were called with empty lists (#52)

### 1.3.27 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*

### 1.3.28 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 *Introspection* (#59)

### 1.3.29 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*.

### 1.3.30 1.9 (2019-08-04)

- `table.m2m(...)` method for creating many-to-many relationships: *Working with many-to-many relationships* (#23)

### 1.3.31 1.8 (2019-07-28)

- `table.update(pk, values)` method: *Updating a specific record* (#35)

### 1.3.32 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*

### 1.3.33 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)`

### 1.3.34 1.6 (2019-07-18)

- `sqlite-utils insert` can now accept TSV data via the new `--tsv` option (#41)

### 1.3.35 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)

### 1.3.36 1.4.1 (2019-07-14)

- Assorted minor documentation fixes: changes since 1.4

### 1.3.37 1.4 (2019-06-30)

- Added `sqlite-utils index-foreign-keys` command (*docs*) and `db.index_foreign_keys()` method (*docs*) (#33)

### 1.3.38 1.3 (2019-06-28)

- New mechanism for adding multiple foreign key constraints at once: *db.add_foreign_keys() documentation* (#31)

### 1.3.39 1.2.2 (2019-06-25)

- Fixed bug where `datetime.time` was not being handled correctly

### 1.3.40 1.2.1 (2019-06-20)

- Check the column exists before attempting to add a foreign key (#29)

### 1.3.41 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 instrospecting 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)*

### 1.3.42 1.1 (2019-05-28)

- Support for `ignore=True` / `--ignore` for ignoring inserted records if the primary key alread 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)*

### 1.3.43 1.0.1 (2019-05-27)

- `sqlite-utils rows data.db table --json-cols` - fixed bug where `--json-cols` was not obeyed

### 1.3.44 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*

### 1.3.45 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*

### 1.3.46 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 *Running queries and outputting a table*
- 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

### 1.3.47 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 *Running queries and returning JSON* and *Running queries and returning CSV*
- 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*

### 1.3.48 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*.

### 1.3.49 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
```

### 1.3.50 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.

### 1.3.51 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 *Running queries and returning CSV* and *Running queries 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.
```

### 1.3.52 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

### 1.3.53 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()`

### 1.3.54 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

### 1.3.55 0.4 (2018-07-31)

- `enable_fts()`,`populate_fts()` and `search()` table methods

Take a look at this script for an example of this library in action.