SQLite cheat sheet v5.1

To start the SQLite3 command line

Download the SQLite executable for your platform. Make sure your terminal/shell is in the directory with that executable, then just run:

```
$ sqlite3 nameOfDBFile.sqlite
```

It will create this database file if it doesn't exist.

or on Windows, double-click `sqlite3.exe` and then:

```
sqlite> .open nameOfDBFile.sqlite
```

SQLite settings

Show all settings:

```
sqlite> .show
```

Make tables look pretty (requires SQLite 3.33 or above):

```
sqlite> .mode table
```

Import/export

As SQL commands

Dump *commands* to reproduce an entire database:

```
sqlite> .output nameOfDumpFile.sql
sqlite> .dump
sqlite> .output
```

Read SQL *commands* from a file:

```
sqlite> .read nameOfFile.sql
```

As CSV data

Output query results in CSV format:

```
sqlite> .mode csv
sqlite> .output nameOfNewCsvFile.csv
sqlite> (whatever query you want to run)
sqlite> .output
```

Import *data* from CSV file

```
sqlite> .import nameOfFile.csv tableName
```

Meta browsing

List all tables:

```
sqlite> .tables
```

Find tables whose name matches a string:

```
sqlite> .tables %stringToMatch%
```

Show the `CREATE TABLE` command for a table:

```
sqlite> .schema tableName
```

Data types

We'll only use three data types in MySQL:

- `integer` (like `int`)
- `real` (like `double`)
- `text` (like `String`)

Creating tables

Basic

```
CREATE TABLE flavor (
    name text primary key,
    costpercarton real,
    freezerId integer
);
```

Multiple and composite keys

If you have multiple keys, specify all but the primary as `unique`. If you have a key with more than one attribute, specify it as a separate line with `primary key` (or `unique`) and the attribute list in bananas.

```
CREATE TABLE section (
    crn text,
    prefix text,
    number integer,
    secnum integer NOT NULL,
    days text,
    times text,
    primary key (crn),
    unique (prefix, number, secnum)
);
```

"NOT NULL" constraint

In the above example, `NOT NULL` means the database is instructed not to allow any row into the `section` table if it's missing a value for `secnum`.

Inserting data

```
INSERT INTO flavor (name, costpercarton, freezerId) VALUES
```

```
    ('Cherry',.35,19),
    ('Vanilla',.28,21),
    ('Dark Chocolate',.33,8);
```

You can omit the list of attributes if you are inserting *all* of them, in the order they were listed in the `CREATE TABLE` statement. (You can't omit `NOT NULL` values.)

Viewing the contents of a table

For now, all you need to know is:

```
SELECT * FROM tablename;
```

(Don't forget to `.mode table` if you want it to look pretty.)

Foreign keys

To enable (for *each* connection):

```
sqlite> pragma foreign_keys = on;
```

Then, a `FOREIGN KEY` statement will be honored. Btw, in SQLite3 you can only FK to a "`UNIQUE`" attribute set (or primary key):

```
sqlite> CREATE TABLE person (
    name text UNIQUE,
    favcolor text);
sqlite> CREATE TABLE item (
    name text,
    weight integer,
    owner text,
    FOREIGN KEY (owner) REFERENCES person(name)
);
```

Foreign key policy

You can also tell SQLite3 how you want it to handle attempted FK violations. Your choices here are `RESTRICT` (to outright disallow them), `SET NULL` (to allow them, but set the referencing table's values to `NULL` in affected row(s)), or `CASCADE` (to propagate value changes to the referencing table). And you can choose any of the three policies for the two separate types of violations: violations by attempted update (*e.g.*, change CPSC 350 to CPSC 351 in a Course table, screwing up CPSC 350 sections in the Sections table) or attempted delete (*e.g.*, delete CPSC 350 entirely from a Course table, similarly screwing up CPSC 350 sections.) The syntax is simply:

```
sqlite> CREATE TABLE item (
    name text,
    weight integer,
    owner text,
    FOREIGN KEY (owner) REFERENCES person(name) ON UPDATE CASCADE ON DELETE SET NULL
);
```

The SQL SELECT statement

Overall architecture

```
SELECT stuff
    FROM stuff
    WHERE stuff
    GROUP BY stuff
    HAVING stuff
    ORDER BY stuff
    LIMIT stuff
    OFFSET stuff;
```

Examples

Simple SELECT-FROM-WHEREs

```
select * from assemblyLine;
select * from assemblyLine where mixins='no';
select sum(capacity) from assemblyLine where mixins='yes';

select * from recipe;
select * from recipe where flavorName='vanilla' and cartonsOrdered > 150;
select name as popular from recipe where flavorName in
    ('vanilla','chocolate','cherry') and cartonsOrdered > 150;
```

Wildcard match with LIKE:

```
select * from recipe where name like '%fudge%';
```

Cartesian product joins (using WHERE clause)

```
select * from recipe, flavor;
select * from recipe, flavor where flavorName=flavor.name;
select cartonsOrdered from recipe, flavor where flavorName=flavor.name
    and recipe.name='Chunky Monkey';
```

The join operator (using FROM clause)

```
select * from recipe join flavor on flavorName=flavor.name;
select cartonsOrdered from recipe join flavor on flavorName=flavor.name
    where recipe.name='Chunky Monkey';
```

"Special" joins

Natural join

Match on columns in the two tables which have identical names, and omit the duplicate column in the result:

```
select * from children natural join costumes;
```

Outer join

Ensure that every row of both the left and the right side of the `JOIN` is present in the result, with `NULL` values necessary if a row has no match:

```
select * from recipe outer join ingredients on recipe.name =
    ingredients.recipe_name;
```

Left/right join

Same as an outer join, except only include non-matching rows from the left (or right) table.

```
select * from recipe left join ingredients on recipe.name =
    ingredients.recipe_name;
```

Grouping (using `GROUP BY` and `HAVING` clauses)

In the `SELECT` clause, include one attribute and one aggregate operator; in the `GROUP BY` clause, group by the attribute.

```
select flavorName, count() from recipe group by flavorName;
```

The `HAVING` clause applies filters to the result *after* grouping has been performed. (The `WHERE` clause applies *before*.)

```
select flavorName, sum(cartonsOrdered) as total from recipe
    group by flavorName
    having total > 10;
```

Sorting (using `ORDER BY` clause)

Specify a list of attributes, each optionally followed by `DESC` (for **desc**ending order), which will be used to sort rows in the result. Attributes after the first one in the list are successive tie-breakers.

```
select flavorName, max(cartonsOrdered) as maxorder, count() from recipe
    group by flavorName
    having maxorder > 50
    order by count() desc, maxorder desc;
```

Paging (using `LIMIT` and `OFFSET` clauses)

Ask for a certain number of result rows, starting from a certain position within the entire result set, so that a large result set can be retrieved in piecemeal fashion.

```
select * from recipe limit 10;

select * from recipe limit 10 offset 30;
```

Updating and deleting

In addition to the `INSERT` statement mentioned above, two other ways to modify the database's contents are `UPDATE` and `DELETE`.

The `UPDATE` statement

This is pretty easy once you understand the `SELECT` statement, because one of the two big parts of an `UPDATE` statement is precisely *the `WHERE` clause* from SELECT. This is how you tell SQLite exactly which rows you do (and by implication, which rows you do not) want to change.

The first part has key/value-pair-type things, separated by commas, that specify what to put in the fields. You can put literal values, or computed values based on some existing value in that field or others. Example:

```
UPDATE girlScoutCookies set paid='yes' where ordernum='67811';
UPDATE girlScoutCookies set numThinMints += 2, numTagalongs += 1 where ordernum='67811';
```

The `DELETE` statement

This one is even easier, since the *only* thing of interest is the `WHERE` clause. For example:

```
DELETE FROM girlScoutCookies where numTrefoils >= 1;
```

Python connectivity

A **connection** is a pathway through you can talk to the database and issue commands. Many simultaneous connections can be open at once, to/from different users and programs.

A **cursor** is a paging mechanism which lets you "cursor" (iterate) through the results in a result list.

Reading (SQL SELECT statement)

```
import sqlite3
conn = sqlite3.connect("nameOfDBFile.db")
results = conn.execute("SQL SELECT statement goes here").fetchall()
```

The above code loads the entire result table into memory, puts a *list of tuples* into the `results` variable. Each tuple in the list is a row of the result, and each element of the tuple, from 0 to n-1, is a column of that row. You can then pick apart the results at your leisure and do whatever you want with them.

Updating (SQL INSERT/UPDATE/DELETE statement)

In this case, you don't even need to store a return value from `.execute()` since it's a write operation, not a read.

```
import sqlite3
conn = sqlite3.connect("nameOfDBFile.db")
conn.execute("SQL INSERT/UPATE/DELETE statement goes here")
conn.commit()
```

Without the `.commit()`, the changes made are only to the local connection's *view* of the database, not propagated to the database proper where they are visible to other connections. `.commit()` does that.

In multi-user/program environments, your `.commit()`s will sometimes fail and throw exceptions. You can catch these in a `try/except` structure, and in the failure case, issue a `.rollback()` on the connection to sort of go back in time and try again. (The point in time to which you'll go back is the line of code where you ran `conn.execute("BEGIN TRANSACTION")`.)