**Databases**

**Database (DB)**

A database is a collection of [information](https://searchsqlserver.techtarget.com/definition/information) that is organized so that it can be easily accessed, managed and updated.

Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it. Computer databases typically contain aggregations of data records or [files](https://searchsqlserver.techtarget.com/definition/flat-file), such as sales transactions, product catalogs and inventories, and customer profiles.

Typically, a database manager provides users with the ability to control read/write access, specify report generation and analyze usage. Some databases offer [ACID](https://searchsqlserver.techtarget.com/definition/ACID) (atomicity, consistency, isolation and durability) compliance to guarantee that data is consistent and that transactions are complete.

Databases are prevalent in large [mainframe](https://searchdatacenter.techtarget.com/definition/mainframe) systems, but are also present in smaller distributed [workstations](https://searchmobilecomputing.techtarget.com/definition/workstation) and midrange systems, such as IBM's AS/400 and personal computers.

**Evolution of databases**

Databases have evolved since their inception in the 1960s, beginning with hierarchical and network databases, through the 1980s with [object-oriented databases](https://searchoracle.techtarget.com/definition/object-oriented-database-management-system), and today with SQL and NoSQL databases and [cloud databases](https://searchcloudapplications.techtarget.com/definition/cloud-database).

In one view, databases can be classified according to content type: bibliographic, full text, numeric and images. In computing, databases are sometimes classified according to their organizational approach. There are many different kinds of databases, ranging from the most prevalent approach, the relational database, to a [distributed database](https://searchoracle.techtarget.com/definition/distributed-database), [cloud database](https://searchcloudapplications.techtarget.com/definition/cloud-database) or NoSQL database.

**Relational database**

A [relational database](https://searchsqlserver.techtarget.com/definition/relational-database), invented by [E.F. Codd](https://searchoracle.techtarget.com/definition/E-F-Codd) at IBM in 1970, is a tabular database in which data is defined so that it can be reorganized and accessed in a number of different ways.

Relational databases are made up of a set of tables with data that fits into a predefined category. Each table has at least one data category in a column, and each row has a certain data instance for the categories which are defined in the columns.

The [Structured Query Language](https://searchsqlserver.techtarget.com/definition/SQL) (SQL) is the standard user and application program interface for a relational database. Relational databases are easy to extend, and a new data category can be added after the original database creation without requiring that you modify all the existing applications.

**Distributed database**

A distributed database is a database in which portions of the database are stored in multiple physical locations, and in which processing is dispersed or replicated among different points in a network.

Distributed databases can be homogeneous or heterogeneous. All the physical locations in a homogeneous [distributed database system](https://searchsqlserver.techtarget.com/definition/DDBMS) have the same underlying hardware and run the same operating systems and database applications. The hardware, operating systems or database applications in a heterogeneous distributed database may be different at each of the locations.

**Cloud database**

A cloud database is a database that has been optimized or built for a virtualized environment, either in a hybrid cloud, public cloud or private cloud. Cloud databases provide benefits such as the ability to pay for storage capacity and bandwidth on a per-use basis, and they provide scalability on demand, along with [high availability](https://searchdatacenter.techtarget.com/definition/high-availability).

A cloud database also gives enterprises the opportunity to support business applications in a [software-as-a-service](https://searchcloudcomputing.techtarget.com/definition/Software-as-a-Service) deployment.

**NoSQL database**

[NoSQL databases](https://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL) are useful for large sets of distributed data.

NoSQL databases are effective for big data performance issues that relational databases aren't built to solve. They are most effective when an organization must analyze large chunks of [unstructured](https://searchbusinessanalytics.techtarget.com/definition/unstructured-data) data or data that's stored across multiple [virtual servers](https://searchnetworking.techtarget.com/definition/virtual-server) in the cloud.

**Object-oriented database**

Items created using [object-oriented programming languages](https://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP) are often stored in relational databases, but object-oriented databases are well-suited for those items.

An object-oriented database is organized around objects rather than actions, and data rather than logic. For example, a multimedia record in a relational database can be a definable data object, as opposed to an alphanumeric value.

**Graph database**

A graph-oriented database, or [graph database](https://whatis.techtarget.com/definition/graph-database), is a type of NoSQL database that uses [graph theory](https://whatis.techtarget.com/definition/graph-theory) to store, map and query relationships. Graph databases are basically collections of nodes and edges, where each node represents an entity, and each edge represents a connection between nodes.

Graph databases are growing in popularity for analyzing interconnections. For example, companies might use a graph database to [mine data](https://searchsqlserver.techtarget.com/definition/data-mining) about customers from [social media](https://whatis.techtarget.com/definition/social-media).

**Accessing the database: DBMS and RDBMS**

A [database management system](https://searchsqlserver.techtarget.com/definition/database-management-system) (DBMS) is a type of software that allows you to define, manipulate, retrieve and manage data stored within a database. A [relational database management system](https://searchsqlserver.techtarget.com/definition/relational-database-management-system) (RDBMS) is a type of database management software that was developed in the 1970s, based on the relational model, and is still the most popular way to manage a database.

**Sqlite**

SQLite is an in-process library that implements a [self-contained](https://www.sqlite.org/selfcontained.html), [serverless](https://www.sqlite.org/serverless.html), [zero-configuration](https://www.sqlite.org/zeroconf.html), [transactional](https://www.sqlite.org/transactional.html) SQL database engine. The code for SQLite is in the [public domain](https://www.sqlite.org/copyright.html) and is thus free for use for any purpose, commercial or private. SQLite is the [most widely deployed](https://www.sqlite.org/mostdeployed.html) database in the world with more applications than we can count, including several [high-profile projects.](https://www.sqlite.org/famous.html)

Basic Example: (<https://github.com/Satyam-Bhalla/Acadview-Python/blob/master/Database/Sqllite.py>)

## **Connecting to an SQLite database**

The sqlite3 that we will be using throughout this tutorial is part of the Python Standard Library and is a nice and easy interface to SQLite databases: There are no server processes involved, no configurations required, and no other obstacles we have to worry about.

In general, the only thing that needs to be done before we can perform any operation on a SQLite database via Python’s sqlite3 module, is to open a connection to an SQLite database file:

import sqlite3

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

where the database file (sqlite\_file) can reside anywhere on our disk, e.g.,

sqlite\_file **=** '/Users/Sebastian/Desktop/my\_db.sqlite'

Conveniently, a new database file (.sqlite file) will be created automatically the first time we try to connect to a database. However, we have to be aware that it won’t have a table, yet. In the following section, we will take a look at some example code of how to create a new SQLite database files with tables for storing some data.

To round up this section about connecting to a SQLite database file, there are two more operations that are worth mentioning. If we are finished with our operations on the database file, we have to close the connection via the .close() method:

conn**.**close()

And if we performed any operation on the database other than sending queries, we need to commit those changes via the .commit() method before we close the connection:

conn**.**commit()

conn**.**close()

## **Creating a new SQLite database**

Let us have a look at some example code to create a new SQLite database file with two tables: One with and one without a PRIMARY KEY column (don’t worry, there is more information about PRIMARY KEYs further down in this section).

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite' *# name of the sqlite database file*

table\_name1 **=** 'my\_table\_1' *# name of the table to be created*

table\_name2 **=** 'my\_table\_2' *# name of the table to be created*

new\_field **=** 'my\_1st\_column' *# name of the column*

field\_type **=** 'INTEGER' *# column data type*

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# Creating a new SQLite table with 1 column*

c**.**execute('CREATE TABLE {tn} ({nf} {ft})'**.**format(tn**=**table\_name1, nf**=**new\_field, ft**=**field\_type))

*# Creating a second table with 1 column and set it as PRIMARY KEY*

*# note that PRIMARY KEY column must consist of unique values!*

c**.**execute('CREATE TABLE {tn} ({nf} {ft} PRIMARY KEY)'**.**format(tn**=**table\_name2, nf**=**new\_field, ft**=**field\_type))

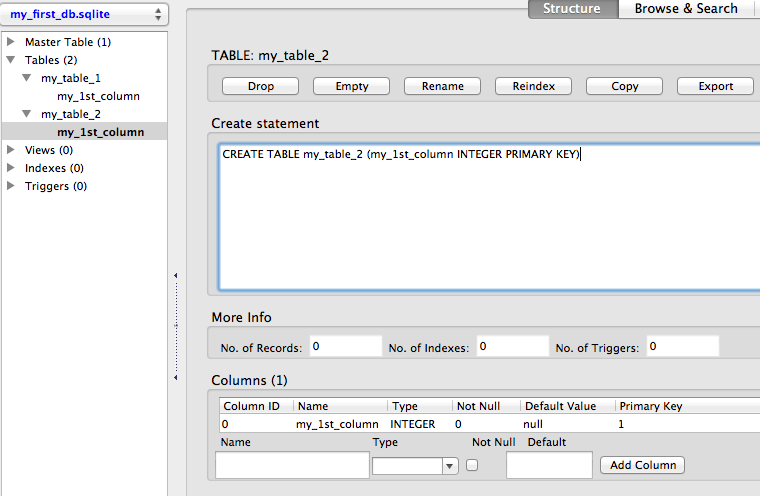
*# Committing changes and closing the connection to the database file*

conn**.**commit()

conn**.**close()

Download the script at: [create\_new\_db.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/create_new_db.py)

**Tip:**   
A handy tool to visualize and access SQLite databases is the free FireFox [SQLite Manager](https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/?src) add-on Or You can also use the db viewer for windows with the basic example given above. Throughout this article, I will use this tool to provide screenshots of the database structures that we created below the corresponding code sections.



Using the code above, we created a new .sqlite database file with 2 tables. Each table consists of currently one column only, which is of type INTEGER.

**Here is a quick overview of all data types that are supported by SQLite 3:**

* INTEGER: A signed integer up to 8 bytes depending on the magnitude of the value.
* REAL: An 8-byte floating point value.
* TEXT: A text string, typically UTF-8 encoded (depending on the database encoding).
* BLOB: A blob of data (binary large object) for storing binary data.
* NULL: A NULL value, represents missing data or an empty cell.

Looking at the table above, You might have noticed that SQLite 3 has no designated Boolean data type. However, this should not be an issue, since we could simply re-purpose the INTEGER type to represent Boolean values (0 = false, 1 = true).

**A quick word on PRIMARY KEYS:**

In our example code above, we set our 1 column in the second table to PRIMARY KEY. The advantage of a PRIMARY KEY index is a significant performance gain if we use the PRIMARY KEY column as query for accessing rows in the table. Every table can only have max. 1 PRIMARY KEY (single or multiple column(s)), and the values in this column MUST be unique! But more on column indexing in the a [later section](https://sebastianraschka.com/Articles/2014_sqlite_in_python_tutorial.html#unique_indexes).

## **Adding new columns**

If we want to add a new column to an existing SQLite database table, we can either leave the cells for each row empty (NULL value), or we can set a default value for each cell, which is pretty convenient for certain applications.  
Let’s have a look at some code:

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite' *# name of the sqlite database file*

table\_name **=** 'my\_table\_2' *# name of the table to be created*

id\_column **=** 'my\_1st\_column' *# name of the PRIMARY KEY column*

new\_column1 **=** 'my\_2nd\_column' *# name of the new column*

new\_column2 **=** 'my\_3nd\_column' *# name of the new column*

column\_type **=** 'TEXT' *# E.g., INTEGER, TEXT, NULL, REAL, BLOB*

default\_val **=** 'Hello World' *# a default value for the new column rows*

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# A) Adding a new column without a row value*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}' {ct}"**.**format(tn**=**table\_name, cn**=**new\_column1, ct**=**column\_type))

*# B) Adding a new column with a default row value*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}' {ct} DEFAULT '{df}'"**.**format(tn**=**table\_name, cn**=**new\_column2, ct**=**column\_type, df**=**default\_val))

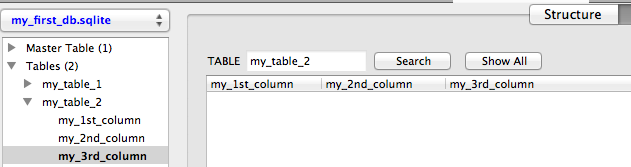
*# Committing changes and closing the connection to the database file*

conn**.**commit()

conn**.**close()

Download the script:

[add\_new\_column.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/add_new_column.py)



We just added 2 more columns (my\_2nd\_column and my\_3rd\_column) to my\_table\_2 of our SQLite database next to the PRIMARY KEY column my\_1st\_column.  
The difference between the two new columns is that we initialized my\_3rd\_column with a default value (here:’Hello World’), which will be inserted for every existing cell under this column and for every new row that we are going to add to the table if we don’t insert or update it with a different value.

## **Inserting and updating rows**

Inserting and updating rows into an existing SQLite database table - next to sending queries - is probably the most common database operation. The Structured Query Language has a convenient UPSERT function, which is basically just a merge between UPDATE and INSERT: It inserts new rows into a database table with a value for the PRIMARY KEY column if it does not exist yet, or updates a row for an existing PRIMARY KEY value. Unfortunately, this convenient syntax is not supported by the more compact SQLite database implementation that we are using here. However, there are some workarounds. But let us first have a look at the example code:

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite'

table\_name **=** 'my\_table\_2'

id\_column **=** 'my\_1st\_column'

column\_name **=** 'my\_2nd\_column'

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# A) Inserts an ID with a specific value in a second column*

**try**:

c**.**execute("INSERT INTO {tn} ({idf}, {cn}) VALUES (123456, 'test')"**.**format(tn**=**table\_name, idf**=**id\_column, cn**=**column\_name))

**except** sqlite3**.**IntegrityError:

**print**('ERROR: ID already exists in PRIMARY KEY column {}'**.**format(id\_column))

*# B) Tries to insert an ID (if it does not exist yet)*

*# with a specific value in a second column*

c**.**execute("INSERT OR IGNORE INTO {tn} ({idf}, {cn}) VALUES (123456, 'test')"**.f**ormat(tn**=**table\_name, idf**=**id\_column, cn**=**column\_name))

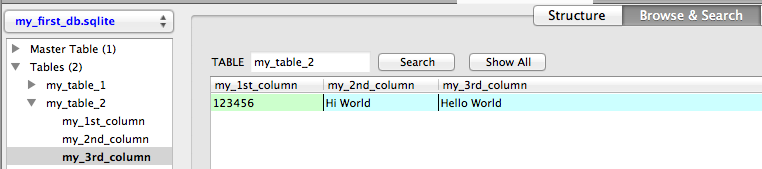
*# C) Updates the newly inserted or pre-existing entry*

c**.**execute("UPDATE {tn} SET {cn}=('Hi World') WHERE {idf}=(123456)"**.**format(tn**=**table\_name, cn**=**column\_name, idf**=**id\_column))

conn**.**commit()

conn**.**close()

Download the script:  
[update\_or\_insert\_records.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/update_or_insert_records.py)



Both A) INSERT and B) INSERT OR IGNORE have in common that they append new rows to the database if a given PRIMARY KEY does not exist in the database table, yet. However, if we’d try to append a PRIMARY KEY value that is not unique, a simple INSERT would raise ansqlite3.IntegrityError exception, which can be either captured via a try-except statement (case A) or circumvented by the SQLite call INSERT OR IGNORE (case B). This can be pretty useful if we want to construct an UPSERT equivalent in SQLite. E.g., if we want to add a dataset to an existing database table that contains a mix between existing and new IDs for our PRIMARY KEY column.

## **Creating unique indexes**

Just like hashtable-datastructures, indexes function as direct pointers to our data in a table for a particular column (i.e., the indexed column). For example, the PRIMARY KEY column would have such an index by default. The downside of indexes is that every row value in the column must be unique. However, it is recommended and pretty useful to index certain columns if possible, since it rewards us with a significant performance gain for the data retrieval.  
The example code below shows how to add such an unique index to an existing column in an SQLite database table. And if we should decide to insert non-unique values into a indexed column later, there is also a convenient way to drop the index, which is also shown in the code below.

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite' *# name of the sqlite database file*

table\_name **=** 'my\_table\_2' *# name of the table to be created*

id\_column **=** 'my\_1st\_column' *# name of the PRIMARY KEY column*

new\_column **=** 'unique\_names' *# name of the new column*

column\_type **=** 'TEXT' *# E.g., INTEGER, TEXT, NULL, REAL, BLOB*

index\_name **=** 'my\_unique\_index' *# name for the new unique index*

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# Adding a new column and update some record*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}' {ct}"**.**format(tn**=**table\_name, cn**=**new\_column, ct**=**column\_type))

c**.**execute("UPDATE {tn} SET {cn}='sebastian\_r' WHERE {idf}=123456"**.**format(tn**=**table\_name, idf**=**id\_column, cn**=**new\_column))

*# Creating an unique index*

c**.**execute('CREATE INDEX {ix} on {tn}({cn})'**.**format(ix**=**index\_name, tn**=**table\_name, cn**=**new\_column))

*# Dropping the unique index*

*# E.g., to avoid future conflicts with update/insert functions*

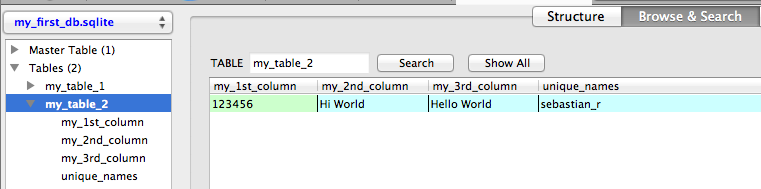
c**.**execute('DROP INDEX {ix}'**.**format(ix**=**index\_name))

*# Committing changes and closing the connection to the database file*

conn**.**commit()

conn**.**close()

Download the script:  
[create\_unique\_index.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/create_unique_index.py)



## **Querying the database - Selecting rows**

After we learned about how to create and modify SQLite databases, it’s about time for some data retrieval. The code below illustrates how we can retrieve row entries for all or some columns if they match certain criteria.

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite' *# name of the sqlite database file*

table\_name **=** 'my\_table\_2' *# name of the table to be queried*

id\_column **=** 'my\_1st\_column'

some\_id **=** 123456

column\_2 **=** 'my\_2nd\_column'

column\_3 **=** 'my\_3rd\_column'

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# 1) Contents of all columns for row that match a certain value in 1 column*

c**.**execute('SELECT \* FROM {tn} WHERE {cn}="Hi World"'**.**format(tn**=**table\_name, cn**=**column\_2))

all\_rows **=** c**.**fetchall()

**print**('1):', all\_rows)

*# 2) Value of a particular column for rows that match a certain value in column\_1*

c**.**execute('SELECT ({coi}) FROM {tn} WHERE {cn}="Hi World"'**.**format(coi**=**column\_2, tn**=**table\_name, cn**=**column\_2))

all\_rows **=** c**.**fetchall()

**print**('2):', all\_rows)

*# 3) Value of 2 particular columns for rows that match a certain value in 1 column*

c**.**execute('SELECT {coi1},{coi2} FROM {tn} WHERE {coi1}="Hi World"'**.**format(coi1**=**column\_2, coi2**=**column\_3, tn**=**table\_name, cn**=**column\_2))

all\_rows **=** c**.**fetchall()

**print**('3):', all\_rows)

*# 4) Selecting only up to 10 rows that match a certain value in 1 column*

c**.**execute('SELECT \* FROM {tn} WHERE {cn}="Hi World" LIMIT 10'**.**format(tn**=**table\_name, cn**=**column\_2))

ten\_rows **=** c**.**fetchall()

**print**('4):', ten\_rows)

*# 5) Check if a certain ID exists and print its column contents*

c**.**execute("SELECT \* FROM {tn} WHERE {idf}={my\_id}"**.**format(tn**=**table\_name, cn**=**column\_2, idf**=**id\_column, my\_id**=**some\_id))

id\_exists **=** c**.**fetchone()

**if** id\_exists:

**print**('5): {}'**.**format(id\_exists))

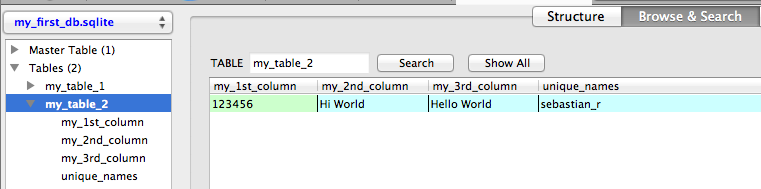
**else**:

**print**('5): {} does not exist'**.**format(some\_id))

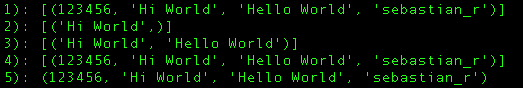
*# Closing the connection to the database file*

conn**.**close()

Download the script:  
[selecting\_entries.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/selecting_entries.py)



if we use the .fetchall() method, we return a list of tuples from the database query, where each tuple represents one row entry. The print output for the 5 different cases shown in the code above would look like this (note that we only have a table with 1 row here):\



## **Security and injection attacks**

So far, we have been using Python’s string formatting method to insert parameters like table and column names into the c.execute() functions. This is fine if we just want to use the database for ourselves. However, this leaves our database vulnerable to injection attacks. For example, if our database would be part of a web application, it would allow hackers to directly communicate with the database in order to bypass login and password verification and steal data.  
In order to prevent this, it is recommended to use ? place holders in the SQLite commands instead of the % formatting expression or the .format() method, which we have been using in this tutorial.  
For example, instead of using

*# 5) Check if a certain ID exists and print its column contents*

c**.**execute("SELECT \* FROM {tn} WHERE {idf}={my\_id}"**.**format(tn**=**table\_name, cn**=**column\_2, idf**=**id\_column, my\_id**=**some\_id))

in the [Querying the database - Selecting rows](https://sebastianraschka.com/Articles/2014_sqlite_in_python_tutorial.html#querying) section above, we would want to use the ? placeholder for the queried column value and include the variable(s) (here: 123456), which we want to insert, as tuple at the end of the c.execute() string.

*# 5) Check if a certain ID exists and print its column contents*

c**.**execute("SELECT \* FROM {tn} WHERE {idf}=?"**.**format(tn**=**table\_name, cn**=**column\_2, idf**=**id\_column), (123456,))

However, the problem with this approach is that it would only work for values, not for column or table names. So what are we supposed to do with the rest of the string if we want to protect ourselves from injection attacks? The easy solution would be to refrain from using variables in SQLite queries whenever possible, and if it cannot be avoided, we would want to use a function that strips all non-alphanumerical characters from the stored content of the variable, e.g.,

**def** **clean\_name**(some\_var):

**return** ''**.**join(char **for** char **in** some\_var **if** char**.**isalnum())

## **Date and time operations**

SQLite inherited the convenient date and time operations from SQL, which are one of my favorite features of the Structured Query Language: It does not only allow us to insert dates and times in various different formats, but we can also perform simple + and - arithmetic, for example to look up entries that have been added xxx days ago.

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite' *# name of the sqlite database file*

table\_name **=** 'my\_table\_3' *# name of the table to be created*

id\_field **=** 'id' *# name of the ID column*

date\_col **=** 'date' *# name of the date column*

time\_col **=** 'time'*# name of the time column*

date\_time\_col **=** 'date\_time' *# name of the date & time column*

field\_type **=** 'TEXT' *# column data type*

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# Creating a new SQLite table with 1 column*

c**.**execute('CREATE TABLE {tn} ({fn} {ft} PRIMARY KEY)'**.**format(tn**=**table\_name, fn**=**id\_field, ft**=**field\_type))

*# A) Adding a new column to save date insert a row with the current date*

*# in the following format: YYYY-MM-DD*

*# e.g., 2014-03-06*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}'"**.**format(tn**=**table\_name, cn**=**date\_col))

*# insert a new row with the current date and time, e.g., 2014-03-06*

c**.**execute("INSERT INTO {tn} ({idf}, {cn}) VALUES('some\_id1', DATE('now'))".format(tn**=**table\_name, idf**=**id\_field, cn**=**date\_col))

*# B) Adding a new column to save date and time and update with the current time*

*# in the following format: HH:MM:SS*

*# e.g., 16:26:37*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}'"**.**format(tn**=**table\_name, cn**=**time\_col))

*# update row for the new current date and time column, e.g., 2014-03-06 16:26:37*

c**.**execute("UPDATE {tn} SET {cn}=TIME('now') WHERE {idf}='some\_id1'"**.**format(tn**=**table\_name, idf**=**id\_field, cn**=**time\_col))

*# C) Adding a new column to save date and time and update with current date-time*

*# in the following format: YYYY-MM-DD HH:MM:SS*

*# e.g., 2014-03-06 16:26:37*

c**.**execute("ALTER TABLE {tn} ADD COLUMN '{cn}'"**.**format(tn**=**table\_name, cn**=**date\_time\_col))

*# update row for the new current date and time column, e.g., 2014-03-06 16:26:37*

c**.**execute("UPDATE {tn} SET {cn}=(CURRENT\_TIMESTAMP) WHERE {idf}='some\_id1'"**.**format(tn**=**table\_name, idf**=**id\_field, cn**=**date\_time\_col))

*# The database should now look like this:*

*# id date time date\_time*

*# "some\_id1" "2014-03-06" "16:42:30" "2014-03-06 16:42:30"*

*# 4) Retrieve all IDs of entries between 2 date\_times*

c**.**execute("SELECT {idf} FROM {tn} WHERE {cn} BETWEEN '2013-03-06 10:10:10' AND '2015-03-06 10:10:10'"**.**format(idf**=**id\_field, tn**=**table\_name, cn**=**date\_time\_col))

all\_date\_times **=** c**.**fetchall()

**print**('4) all entries between ~2013 - 2015:', all\_date\_times)

*# 5) Retrieve all IDs of entries between that are older than 1 day and 12 hrs*

c**.**execute("SELECT {idf} FROM {tn} WHERE DATE('now') - {dc} >= 1 AND DATE('now') - {tc} >= 12"**.**format(idf**=**id\_field, tn**=**table\_name, dc**=**date\_col, tc**=**time\_col))

all\_1day12hrs\_entries **=** c**.**fetchall()

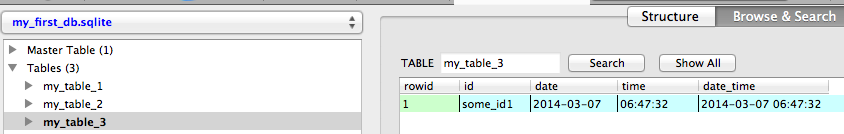
**print**('5) entries older than 1 day:', all\_1day12hrs\_entries)

*# Committing changes and closing the connection to the database file*

conn**.**commit()

conn**.**close()

Download the script:  
[date\_time\_ops.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/date_time_ops.py)



Some of the really convenient functions that return the current time and date are:

DATE('now') *# returns current date, e.g., 2014-03-06*

TIME('now') *# returns current time, e.g., 10:10:10*

CURRENT\_TIMESTAMP *# returns current date and time, e.g., 2014-03-06 16:42:30*

*# (or alternatively: DATETIME('now'))*

The screenshot below shows the print outputs of the code that we used to query for entries that lie between a specified date interval using

BETWEEN '2013-03-06 10:10:10' AND '2015-03-06 10:10:10'

and entries that are older than 1 day via

WHERE DATE('now') **-** some\_date

Note that we don’t have to provide the complete time stamps here, the same syntax applies to simple dates or simple times only, too.

5_sqlite3_date_time_2.png\

#### **Update Mar 16, 2014:**

If’d we are interested to calculate the hours between two DATETIME() timestamps, we can could use the handy STRFTIME() function like this

SELECT (STRFTIME('%s','2014-03-14 14:51:00') **-** STRFTIME('%s','2014-03-16 14:51:00'))

**/** **-**3600

which would calculate the difference in hours between two dates in this particular example above (here: 48) in this case.  
And to calculate the difference in hours between the current DATETIME and a given DATETIME string, we could use the following SQLite syntax:

SELECT (STRFTIME('%s',DATETIME('now')) **-** STRFTIME('%s','2014-03-15 14:51:00')) **/** 3600

## **Retrieving column names**

In the previous two sections we have seen how we query SQLite databases for data contents. Now let us have a look at how we retrieve its metadata (here: column names):

import sqlite3

sqlite\_file **=** 'my\_first\_db.sqlite'

table\_name **=** 'my\_table\_3'

*# Connecting to the database file*

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

*# Retrieve column information Every column will be represented by a tuple with the following attributes: (id, name, type, notnull, default\_value, primary\_key)*

c**.**execute('PRAGMA TABLE\_INFO({})'**.**format(table\_name))

*# collect names in a list*

names **=** [tup[1] **for** tup **in** c**.**fetchall()]

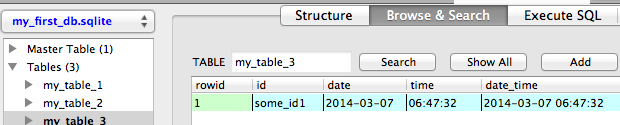
**print**(names)

*# e.g., ['id', 'date', 'time', 'date\_time']*

*# Closing the connection to the database file*

conn**.**close()

Download the script:  
[get\_columnnames.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/get_columnnames.py)



Since we haven’t created a PRIMARY KEY column for my\_table\_3, SQLite automatically provides an indexed rowid column with unique ascending integer values, which will be ignored in our case. Using the PRAGMA TABLE\_INFO() function on our table, we return a list of tuples, where each tuple contains the following information about every column in the table: (id, name, type, notnull, default\_value, primary\_key).So, in order to get the names of every column in our table, we only have to grab the 2nd value in each tuple of the returned list, which can be done by

names **=** [tup[1] **for** tup **in** c**.**fetchall()]

after the PRAGMA TABLE\_INFO() call. If we would print the contents of the variable names now, the output would look like this:

7_sqlite3_get_colnames_2.png

## **Printing a database summary**

I hope we covered most of the basics about SQLite database operations in the previous sections, and by now we should be well equipped to get some serious work done using SQLite in Python.  
Let me conclude this tutorial with an obligatory “last but not least” and a convenient script to print a nice overview of SQLite database tables:

import sqlite3

**def** **connect**(sqlite\_file):

""" Make connection to an SQLite database file """

conn **=** sqlite3**.**connect(sqlite\_file)

c **=** conn**.**cursor()

**return** conn, c

**def** **close**(conn):

""" Commit changes and close connection to the database """

*# conn.commit()*

conn**.**close()

**def** **total\_rows**(cursor, table\_name, print\_out**=**False):

""" Returns the total number of rows in the database """

cursor**.**execute('SELECT COUNT(\*) FROM {}'**.**format(table\_name))

count **=** cursor**.**fetchall()

**if** print\_out:

**print**('\nTotal rows: {}'**.**format(count[0][0]))

**return** count[0][0]

**def** **table\_col\_info**(cursor, table\_name, print\_out**=**False):

""" Returns a list of tuples with column informations:

(id, name, type, notnull, default\_value, primary\_key)

"""

cursor**.**execute('PRAGMA TABLE\_INFO({})'**.**format(table\_name))

info **=** cursor**.**fetchall()

**if** print\_out:

**print**("\nColumn Info:\nID, Name, Type, NotNull, DefaultVal, PrimaryKey")

**for** col **in** info:

**print**(col)

**return** info

**def** **values\_in\_col**(cursor, table\_name, print\_out**=**True):

""" Returns a dictionary with columns as keys

and the number of not-null entries as associated values.

"""

cursor**.**execute('PRAGMA TABLE\_INFO({})'**.**format(table\_name))

info **=** cursor**.**fetchall()

col\_dict **=** dict()

**for** col **in** info:

col\_dict[col[1]] **=** 0

**for** col **in** col\_dict:

c**.**execute('SELECT ({0}) FROM {1} '

'WHERE {0} IS NOT NULL'**.**format(col, table\_name))

*# In my case this approach resulted in better performance than using COUNT*

number\_rows **=** len(c**.**fetchall())

col\_dict[col] **=** number\_rows

**if** print\_out:

**print**("\nNumber of entries per column:")

**for** i **in** col\_dict**.**items():

**print**('{}: {}'**.**format(i[0], i[1]))

**return** col\_dict

**if** \_\_name\_\_ **==** '\_\_main\_\_':

sqlite\_file **=** 'my\_first\_db.sqlite'

table\_name **=** 'my\_table\_3'

conn, c **=** connect(sqlite\_file)

total\_rows(c, table\_name, print\_out**=**True)

table\_col\_info(c, table\_name, print\_out**=**True)

*# next line might be slow on large databases*

values\_in\_col(c, table\_name, print\_out**=**True)

close(conn)

Download the script:  
[print\_db\_info.py](https://github.com/rasbt/python_reference/blob/master/tutorials/sqlite3_howto/code/print_db_info.py)

