Introduction to SQL with SQLite

Table of Contents

Introduction

Overview of Why SQLite Matters

https://youtu.be/q-CzqO_wdxo

SQLite is a Relational Database that is portable, fast, reliable and full featured SQL Database Engine.

It gives you the power of SQL, can store up to 240TB of data in a format that can be copy/pasted and does not have built in security.

SQLite is the most used database systems in the world, and comes with Python3.

We will use SQLite is a store for records in our projects. Whether we are writing blog posts, recording values from sensors, or creating a cache for AI requests and responses we will use SQLite as a store for the data to be able to save and read from.

Generally you will use a back end programming language such as Python to communicate with SQLite to store and access records

Bias

https://youtu.be/pjMdFTk6-wo

For this series there is no specific bias on SQLite. For Silicon Dojo we like to use SQLite because it gives the utility of a relational database with the administrative overhead of a text file.

Database files can literally be copy/pasted to other locations and there are no security or privileges to keep track of. This is good for learning SQL and storing non secure data, but if you have application that needs to protect it's data you should use MySQL or Postgress.

Prerequisites

https://youtu.be/xT3_p_097sk

To learn SQL and SQLite you do not technically need to know other programming languages, or have other technology skills. SQL (Structured Query Language) is a self contained language and you can actually build something useful just with SQLite.

From a practical standpoint you should know a “back end” language to actually be able to build projects with SQLite. We use Python for Silicon Dojo and will use SQLite as a store for records when we are building projects.

To use SQLite3 you simply need it installed on your system. It is installed by default when you install Python3.

What is SQL and SQLite

What is a Database

https://youtu.be/7YkvrNUnCv0

A database stores records for your application. Generally databases only store text based data such as text, ints, floats and boolean values.

Many databases can store multimedia files in a data type called “blob”, but this is generally a bad idea. It is better to store a pointer to a file location within a database so that the file is saved in a “file store’, and relevant data such as “author”, “publication date”, “tags”, “description” and file location are stored in the database.

There used to be only a few widely used databases such as MSSQL, Oracle, MySQL and Postgres. Currently there are a huge number of databases that allow you to store and retrieve all kinds of data. The database that you will end up using will be determined by your use case, resources, and ability to maintain the system.

SQL vs NoSQL Databases

SQL was the standard way of interacting with databases. Systems that used SQL were considered reliable, and stable but also could be slow and hard to scale up.

NoSQL databases allow for faster operations and can scale easily, but may not be as reliable.

SQL is used for transactions where data cannot be lost, NoSQL systems are used when some data loss is acceptable. SQL for Inventory Control, NoSQL for Tweets.

ACID – Relational Databases

Atomicity – A transaction is treated as a whole unit. If any part fails the whole transaction fails

Consistency – The transaction must maintain the consistency of the database

Isolation – Transactions are executed individually

Durability – Once transaction completes the changes are permenant

BASE – NoSQL

Basically Available – The database will always provide a response, but may not be the most up to date data

Soft State – The system may be in a state of change even when no transactions are being made

Eventual Consistency – Data in database will eventually be updated

IaaS Changed Things

SQL databases used to be much more difficult to administer. To scale and provide reliability they used to be turned into clusters of servers and the servers would replicate changes in data. Additionally there may have been servers designed to be closer to the end user that would perform specific functions. This is where the concept of read, write, read/write and master servers came from.

With the rise of AWS and Azure you can now point to their database services and they will automatically do the work to scale and provide reliability and redundancy.

What is a Relational Database

https://youtu.be/8pGDhkgVqMU

A relational database stores records within a predefined schema. The schema for a table defines what data is to be collected and its data type. Tables can be modified, but it requires a specific statement and is considered a significant risk.

For relational databases tables are used for specific containers of records. Different tables are then tied together using a JOIN statement. So for when creating an invoice you might JOIN a transaction table to a parts table. The transaction table will connect to a part id on the parts table and by joining the tables together you will have access to the transaction, and the information about the parts.

In this example we created a part table and a vendor table. In the part table we reference the vendors with a vendor_id column. This allows us to separate the vendor information from the part information in the tables, and then we can use a join statement to connect the two tables in SQL.

Notice that we use an auto incrementing primary key in the vendor table so that we have an identifier for the vendors that won't be modified.

```
create table part(name,description,vendor_id);
```

```
create table vendor(vendor_id integer primary key autoincrement, name);
```

```
sqlite> select * from part;
widget|a thingy|3
sprocket|another thingy|2
ratchet|another thingy|1
washer|another thingy|2
doo dad|another thingy|
```

```
sqlite> select * from vendor;
1|cool company
2|better company
3|amazeballs inc
```

```
sqlite> select part.name, part.description, vendor.name from part
            inner join vendor on part.vendor_id = vendor.vendor_id;
widget|a thingy|amazeballs inc
sprocket|another thingy|better company
ratchet|another thingy|cool company
washer|another thingy|better company
```

What is SQL

https://youtu.be/mpciEDAgCHk

SQL stands for Structured Query Language. It is the standard language used to interact with Relational Databases.

SQL was set as an ISO standard in 1987. As a standard most of the syntax is the same whether you interact with SQLite, MySQL, Postgres of MSSQL.

For new users, and simple projects basic SQL should work on all database systems. For complex tasks there may be specific syntax for different systems.

SQL uses a semi colon as the delimiter. Until you add a ; the statement will not execute when you hit the return button.

```
select * from table_name;
```

```
insert into table_name(name,age,size)values(‘bob’,22,’small’);
```

ORM – Object Relationship Mapper

When using some frameworks such as Django you may be advises to use an ORM. ORM’s are ways to abstract the communication with the database away from the raw SQL. We will not discuss these in this series, and you don’t need to use an ORM in general, but you should be aware they exist.

```
students = Student.objects.all()
for student in students:
    print(student.name, student.age, student.size)
```

What is SQLite

https://youtu.be/hSaHfr-YrLI

SQLite is a lightweight Relational Database Engine. It is fast, and can store a database of up to 281TB in size.

It is portable so that the entire file can simply be copy and pasted to another system, and there is no security. So it good for data that is not private, or could be an issue if compromised.

SQLite does not require a data type, and takes data types as only a suggestion unless a table is created in strict mode.

It automatically creates a ROWID column for a table. This can be used as a unique identifier for records.   If a record is deleted the ROWID number may be automatically reused.

SQLite comes with the Python3 installation so if Python is installed SQLite should be too.

Sqlite Version 3 came out in 2004.

https://www.sqlite.org/index.html

Using SQLite

https://youtu.be/9FjJVu4urlA

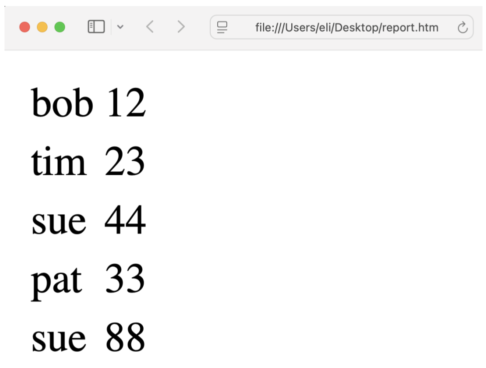
SQLite Shell

SQLite can be used from the terminal. You can enter the SQLite3 shell by using the SQLite3 command.

Python and SQLite

You can connect to SQLite using the sqlite3 module and send SQL Statements to interact with the database.

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()
cursor.execute('CREATE TABLE IF NOT EXISTS people(name,age)')
conn.commit()
conn.close()

while True:
    name = input('Name: ')
    age = input('Age: ')

    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO people(name,age) VALUES(?,?)",
                   (name, age))
    conn.commit()
    conn.close()

    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM people")
    rows = cursor.fetchall()
    conn.close()

    with open('report.htm', 'w') as file:
        file.write('<table>')
        for person in rows:
            file.write(f'''<tr>
                       <td>{person[0]}</td>
                       <td>{person[1]}</td>
                       </tr>''')
        file.write('</table>')
```



bob	12
tim	23
sue	44
pat	33
sue	88

SQLite Basic Tasks and Special Commands

Installing SQLite3

https://youtu.be/ZGOTn8ZdSvU

SQLite3 will be installed on your system if you have installed Python3. To determine if it has been installed run the –version command on the terminal.

```
sqlite3 --version
```

If sqlite3 is not installed then install Python3 onto your system.

https://www.python.org

SQLite Shell

https://youtu.be/w5AXdd35ZBc

You can interact with the SQLite database directly from the terminal. The SQLite shell is like other database shells for standard SQL. To control the server itself there are commands that start with a period.

https://www.sqlite.org/cli.html

To Create a Temporary Database (This database ceases to exist when you exit)

```
sqlite3
```

Save Temporary Database, or Create a Backup

```
.backup database_name.db
```

To Open a Specific Database or Create One

```
sqlite3 example.db
```

To Exit

```
.exit
```

For List of Commands

```
.help
```

To Show the Tables in the Database

```
.tables
```

```
sqlite> .tables
people
```

To View the Schema of the Tables in the Database

```
.schema
```

```
sqlite> .schema
CREATE TABLE people(name,age);
```

Show Column Headers when Doing Select Statements

```
.headers on
```

```
sqlite> .headers on
sqlite> select * from people;
name|age
bob|12
tim|23
sue|44
pat|33
sue|88
```

View SQLite3 Version

```
.version
```

```
sqlite> .version
SQLite 3.43.2 2023-10-10 13:08:14
1b37c146ee9ebb7acd0160c0ab1fd11017a419fa8a3187386ed8cb32b709aapl
zlib version 1.2.12
clang-16.0.0 (64-bit)
```

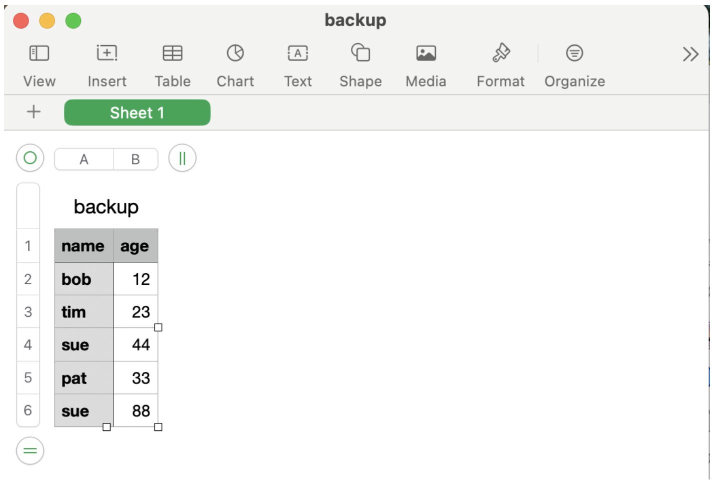
Export Data to CSV

Change Mode to CSV

Change Output to a .csv file

Run SELECT statements

```
sqlite> .mode csv
sqlite> .output backup.csv
sqlite> select * from people;
```



name	age
bob	12
tim	23
sue	44
pat	33
sue	88

Change Output Back to Shell

```
sqlite> .output
```

SQL – Structured Query Language

Create Tables and Data Types

https://youtu.be/NKp9W5GM-WI

SQLite is built to be a very simple to use database system. One of the XXXX of this is that you don't need to assign data types to values, and by default even if you do SQLite does not enforce the data type. So you can insert 'bob' into an age column.

To have SQLite enforce data types you have to set the table to strict mode when you create it.

Create a Table

```
create table student(name,age,size);
```

Create a Table and State Data Types

```
create table student2(name text, age integer, size text);
```

Create a Table and Strictly Enforce Data Types

```
create table student3(name text, age integer, size text) strict;

insert into student3(name,age,size)values('bob','tim','large');

Runtime error: cannot store TEXT value in INT column student3.age (19)
```

Data Types in SQLite

You can use INT as an Alias for INTEGER, but in strict mode SQLite might not properly deal with the number.

You can use FLOAT as an alias for REAL.

Type	Description
INTEGER	Stores whole numbers.
REAL	Stores floating-point numbers.
TEXT	Stores strings.
BLOB	Stores binary data.
NUMERIC	Stores numbers, including decimals.

Primary Keys and ROWID

You do not need to define a primary key with SQLite. By default every record will receive a ROWID.

ROWID is not shown when using a SELECT * statement. You have to explicitly request it.

The quirk of ROWID is that the value may be reused if a record is deleted. If you need a standard auto incrementing primary key do not rely on ROWID.

```
sqlite> select * from student;
name|age|size
timmy|33|small
sue|20|
```

```
sqlite> select rowid,name,age,size from student;
rowid|name|age|size
2|timmy|33|small
3|sue|20|
```

To Create an Auto Increment Primary Key you define it when you create the table.

```
sqlite> create table student4(id integer primary key autoincrement, name,
age,size);

sqlite> insert into student4(name,age,size)values('bob',22,'large');
sqlite> insert into student4(name,age,size)values('sue',20,'small');

sqlite> select * from student4;
id|name|age|size
1|bob|22|large
2|sue|20|small

sqlite> select rowid,id,name,age,size from student4;
id|id|name|age|size
1|1|bob|22|large
2|2|sue|20|small
```

Alter and Drop Tables

https://youtu.be/Cur9pDndt_g

Altering Tables

Altering tables that are already in use may have catastrophic consequences. Always backup your database before you do an alter statement, and the best time to change a schema is while its still on the whiteboard.

For these examples create a table named people in your database

```
sqlite3 example.db
```

```
create table people(name,age);
```

Rename Table

```
alter table old_name rename to new_name;
```

```
sqlite> .tables
people
sqlite> alter table people rename to records;
sqlite> .tables
records
```

Add a Column to a Table

```
alter table table_name add column column_name datatype;
```

```
sqlite> .schema
create table records(name,age);
sqlite> alter table records add column size text;
sqlite> .schema
create table records(name,age, size text);
```

Rename a Column

```
alter table table_name rename column column_name to new_name;
```

```
sqlite> .schema
create table records(name,age, size text);
sqlite> alter table records rename column size to shirt_size;
sqlite> .schema
create table records(name,age, shirt_size text);
```

Drop/ Delete Column

```
alter table table_name drop column column_name;
```

```
sqlite> alter table records drop column shirt_size;
sqlite> .schema
create table records(name,age);
```

Drop Tables

To Drop/ Delete a Table

```
sqlite> drop table records;
sqlite> .schema
sqlite>
```

Insert Record

https://youtu.be/VAV7sd0nQhw

To insert records into a table you use the INSERT statement, state which columns data should be added to, and then provide the values.

To Insert a Record

```
insert into table(column1,column2,column3)values('value',value,'value');
```

For these examples create a table named people in your database

```
sqlite3 example.db
```

```
create table people(name,age);
```

```
insert into people(name,age,size)values('timmy',33);

sqlite> select * from people;
name|age
timmy|33

insert into people(name)values('sue');

sqlite> select * from people;
name|age
timmy|33
sue
```

Select Records

https://youtu.be/oQQDZMqskXI

Selecting records is how you search for and access records from tables.

For these examples add a few more records into the people table that you created for the Insert session.

To Select values from All Columns use a * in Select Statement

```
sqlite> select * from people;
bob|12
tim|23
sue|44
pat|33
```

To Access Individual Columns and the ROWID value

```
sqlite> select rowid,name,age from people;
1|bob|12
2|tim|23
3|sue|44
4|pat|33
5|sue|88
```

```
sqlite> select name from people;
bob
tim
sue
pat
sue
```

To Find Records Based on a Conditional

```
sqlite> select * from people where age > '30';
sue|44
pat|33
sue|88
```

To Find Records Using Wildcard Queries

_ = Match single characters

% = Match any number of caharacters

```
sqlite> select * from people where name like '%b%';
bob|12
billy|22
brad|13
bess|5
```

Sort the Order of Records Returned

To order records ascending use the order by clause.

To order records descending add *desc* to the clause

```
sqlite> select * from people order by age;
bess|5
brad|13
billy|22
bob|12
tim|23
pat|33
sue|44
sue|88
```

```
sqlite> select * from people order by age desc;
sue|88
sue|44
pat|33
tim|23
bob|12
billy|22
brad|13
bess|5
```

Limit

To limit the number of records returned you add a limit clause to the statement

```
sqlite> select * from people limit 3;
bob|12
tim|23
sue|44
```

```
sqlite> select * from people order by age limit 3;
bess|5
brad|13
billy|22
```

Update Records

https://youtu.be/5dNE9U2--G8

You can edit records that have already been created using the UPDATE statement.

By default this will edit all records that match the statement. This is a reason to use a Primary Key, or at least a Row ID value to select a single record for editing.

```
update table_name set column_name = 'value' where name = 'value';
```

In this example we change a lower case name to being saved in uppercase.

```

sqlite> update people set name = 'BOB' where name = 'bob';

sqlite> select * from people;
BOB|12
tim|23
sue|44
pat|33
sue|88
```

```
sqlite> update people set age = '55' where name = 'BOB';

sqlite> select * from people;
BOB|55
tim|23
sue|44
pat|33
sue|88
```

You can use a LIKE clause to select records based on a wildcard statement. This may be good for cleaning up records where non standardized values have been stored.

```
update table_name set column_name = value where column_name like '%value%';
```

In this example we have a table where the value for ‘GIRL’ has been entered in many different ways. We update all records sex column to GIRL that have a ‘g’ in the column. This will update all records that match the pattern so be very careful.

```
sqlite> select * from roster;
bob|33|boy
bill|13|Boy
ben|21|BOY
sue|21|gIrl
sam|24|Girl
pat|42|G
tammy|12|gir

sqlite> update roster set sex = 'GIRL' where sex like '%g%';

sqlite> select * from roster;
```

```
bob|33|boy
bill|13|Boy
ben|21|BOY
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL
```

Delete Record

https://youtu.be/ouvGQ6QTenA

To delete records you can use the DELETE statement and define either a specific column value, or use a LIKE clause to delete records based off of wildcard values.

```
delete from table_name where column_name = 'value';
```

In this example we delete the record for ‘BOB’ from the table.

```
sqlite> select * from people;
BOB|12
tim|23
sue|44
pat|33
sue|88
billy|22
brad|13
bess|5

sqlite> delete from people where name = 'BOB';

sqlite> select * from people;
tim|23
sue|44
pat|33
sue|88
billy|22
brad|13
bess|5
```

To use wildcards to delete records use the LIKE clause

```
delete from table_name where column_name like '%value%';
```

In this example we delete any record that has ‘b’ in the name column.

```
sqlite> select * from people;
tim|23
sue|44
pat|33
sue|88
billy|22
brad|13
bess|5

sqlite> delete from people where name like '%b%';

sqlite> select * from people;
tim|23
sue|44
pat|33
```

Combining Statements with And, Or

https://youtu.be/77qDttmkIiA

You can test for multiple criteria using AND or OR operators with a SQL statement. This works for SELECT, INSERT, UPDATE and DELETE.

```
select * from table_name where column_name = 'value' and column_name > value;
```

```
select * from table_name where column_name = 'value' or column_name > value;
```

In this example we search for all records with ‘GIRL’ that are over the age of 22.

```
sqlite> select * from roster;
bob|33|boy
bill|13|Boy
ben|21|BOY
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL

sqlite> select * from roster where sex = 'GIRL';
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL

sqlite> select * from roster where sex = 'GIRL' and age > 22;
sam|24|GIRL
pat|42|GIRL
```

Capitalization

https://youtu.be/sSV1O8pZYnc

Capitalization in SQLite can be quirky. It is supposed to be case sensitive, but sometimes it's not.

If you make a query and need the results to be case insensitive there are a couple ways of doing it.

The first way is to use either the upper() or lower() SQL functions when calling the column and all values in the column will be treated as being that case.

In this example we change the values of sex to lowercase, and then test against the value of 'girl'. You see that when we do not use the lower() function no results are returned.

```
sqlite> select * from roster;
bob|33|boy
bill|13|Boy
ben|21|BOY
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL

sqlite> select * from roster where sex = 'girl';

sqlite> select * from roster where lower(sex) = 'girl';
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL
```

The second way to be case insensitive is to add COLLATE NOCASE at the end of the statement.

In this example we use the same table and search for records with 'girl' with COLLATE NOCASE added.

```
sqlite> select * from roster where sex = 'girl' collate nocase;
sue|21|GIRL
sam|24|GIRL
pat|42|GIRL
tammy|12|GIRL
```

This final way is to create the table with COLLATE NOCASE for the column that you want case insensitivity.

```
CREATE TABLE roster (
    name TEXT COLLATE NOCASE,
    age INTEGER,
        sex TEXT COLLATE NOCASE
);
```

Joins

https://youtu.be/fgUMBZAl1hc

Join in SQL combine 2 tables in a select statement. The idea is that each table should be used for a specific set of data, and then link to other tables using identifiers. In the examples below we use parts and vendors tables. The parts table records associate with the vendors table with a vendor_id value.

When creating tables that will be joined to you should generally create an auto incrementing create a primary key. This will allow you to join a value that should not change or be reused automatically by SQLite like a rowid might.

SQLite allows for Inner and Left Joins. It does not support Right Joins.

Note: Foreign Keys are disabled by default in SQLite.

For the Join examples we have created 2 tables and added a number of records to each.

```
sqlite> .schema
CREATE TABLE part(name,description,vendor_id);
CREATE TABLE vendor(vendor_id integer primary key autoincrement, name);
```

```
sqlite> select * from part;
name|description|vendor_id
widget|a thingy|3
sprocket|another thingy|2
ratchet|another thingy|1
washer|another thingy|2
doo dad|another thingy|
```

```
sqlite> select * from vendor;
vendor_id|name
1|cool company
2|better company
3|amazeballs inc
```

Inner Join

Inner Joins return results that match the Join criteria. Any records that do not Join to another table are not displayed. In the example a part record is not associated with a vendor and so it is not displayed.

```
select * from table1 inner join table2 on table1.column = table2.column;
```

```
sqlite> select * from part inner join vendor on part.vendor_id = vendor.vendor_id;
name|description|vendor_id|vendor_id|name
widget|a thingy|3|3|amazeballs inc
sprocket|another thingy|2|2|better company
ratchet|another thingy|1|1|cool company
washer|another thingy|2|2|better company
```

Left Join

Left Joins return all records from the initial table requested, and only records from matching records in the right table. So if a part is not associated with a vendor it will still be displayed.

```
select * from table1 left join table2 on table1.column = table2.column;
```

```
sqlite> select * from part left join vendor on part.vendor_id = vendor.vendor_id;
name|description|vendor_id|vendor_id|name
widget|a thingy|3|3|amazeballs inc
sprocket|another thingy|2|2|better company
ratchet|another thingy|1|1|cool company
washer|another thingy|2|2|better company
doo dad|another thingy|||
```

SQLite Functions

SQLite Functions

https://youtu.be/FPpGZWhj-io

SQLite Functions allow you to perform basic operations to data in the database using SQLite instead of relying on Python or your other programming languages. This is generally faster and uses fewer resources since the database system is doing the work.

In this example we show what happens when 'bob' is added to a record for the age of a person.

```
sqlite> select sum(age), avg(age), max(age), min(age) from roster;
166|23.7142857142857|42|12

sqlite> insert into roster(name,age,sex)values('timmy','bob','girl');

sqlite> select sum(age), avg(age), max(age), min(age) from roster;
166.0|20.75|bob|12
```

In this example we use the length() function to return the character length of names.

```
sqlite> select name, length(name) from roster;
bob|3
bill|4
ben|3
sue|3
sam|3
pat|3
tammy|5
timmy|5
```

In this example we return the names in a table in all capital letters using upper().

```
sqlite> select * from people;
bob|33
PAUL|23
tim|12
timmy|55
Sally|12
mack|22
sqlite> select upper(name) from people;
BOB
PAUL
TIM
TIMMY
SALLY
MACK
```

String Functions

Function	Description
`UPPER(str)`	Converts `str` to uppercase.
`LOWER(str)`	Converts `str` to lowercase.
`LENGTH(str)`	Returns the length of `str`.
`SUBSTR(str, start, length)`	Extracts a substring starting from `start` position with `length` characters.
`TRIM(str)`	Removes leading and trailing spaces from `str`.
`LTRIM(str)`	Removes leading spaces from `str`.
`RTRIM(str)`	Removes trailing spaces from `str`.
`REPLACE(str, old, new)`	Replaces occurrences of `old` with `new` in `str`.
`INSTR(str, substr)`	Returns the position of `substr` in `str`.
`PRINTF(format, args...)`	Formats a string like C’s `printf()`.

Numeric Functions

Function	Description
`ABS(x)`	Returns the absolute value of `x`.
`ROUND(x, y)`	Rounds `x` to `y` decimal places.
`CEIL(x) / CEILING(x)`	Rounds `x` up to the nearest integer.
`FLOOR(x)`	Rounds `x` down to the nearest integer.
`SQRT(x)`	Returns the square root of `x`.
`POWER(x, y)`	Raises `x` to the power of `y`.
`RANDOM()`	Returns a random integer.
`SIGN(x)`	Returns `-1` for negative, `0` for zero, `1` for positive.

Date & Time Functions

Function	Description
`DATE('now')`	Returns the current date (YYYY-MM-DD).
`TIME('now')`	Returns the current time (HH:MM:SS).
`DATETIME('now')`	Returns current date and time (YYYY-MM-DD HH:MM:SS).
`JULIANDAY('now')`	Returns Julian day number.
`STRFTIME('%Y-%m-%d', 'now')`	Formats the date/time as per given format.
`DATE('now', '+7 days')`	Adds 7 days to the current date.
`DATE('now', '-1 month')`	Subtracts 1 month from the current date.

Aggregate Functions

Function	Description
`COUNT(column)`	Returns the number of rows.
`SUM(column)`	Returns the sum of all values in `column`.
`AVG(column)`	Returns the average value of `column`.
`MAX(column)`	Returns the maximum value.
`MIN(column)`	Returns the minimum value.
`GROUP_CONCAT(column, separator)`	Concatenates values into a single string, separated by `separator`.

SQLite and Python

SQLite and Python

https://youtu.be/q83QDlz8pio

To use Python with SQLite there are a few steps to take.

- You have to Import the sqlite3 module
- Create a connection to the databases
- Create a Cursor for the Connection. The cursor is used to execute commands against the database.
- You then create a SQL Statement, and use the cursor to execute the statement.
- We then need to commit() the Statement for the results to be saved to the databse
- We then close the connection

Example SQL with Python

```
import sqlite3

conn = sqlite3.connect("example.db")
cursor = conn.cursor()

sql = ' #SQL Statement '
cursor.execute(sql)

conn.commit()
conn.close()
```

Create Table If Does Not Exist

This statement will create a table if it does not yet exist.

```
CREATE TABLE IF NOT EXISTS table_name (
    column1_name column1_datatype constraints,
    column2_name column2_datatype constraints,
    ...
);
```

In this example we create a table names people with the columns of name and age if the table does not yet exist. It it does exist nothing will happen.

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

sql = 'create table if not exists people(name,age)'
cursor.execute(sql)

conn.commit()
conn.close()
```

Parameterized Queries

When sending SQL Statements you can use f strings to dynamically create the SQL statements, but this creates vulnerabilities and the potential for injection attacks.

With parameterized queries you add a placeholder for values using question marks “?”, and then define the values when you call execute().

Note: If you only have a single parameter you need to end with a comma. The parameters are inputed as a tuple so a single value needs to end with a comma to show that it is a tuple data type.

Insert

This is how an insert statement is formmated

```
sql = 'insert into table(column1,column2)values(?,?)'
cursor.execute(sql,(value1,value2))
```

In this example we use a while True loop to allow us to add numerous names and ages to a table. We send the values to a function to insert the additions to the database.

```
import sqlite3

def add(name,age):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    sql = 'insert into people(name,age)values(?,?)'
    cursor.execute(sql,(name,age))

    conn.commit()
    conn.close()

while True:
    name = input('Name: ')
    age = input('Age: ')

    add(name,age)
```

```
sqlite> select * from people;
bob|33
sue|23
tim|12
timmy|55
sally|12
mack|22
```

Select

To read record from the database we need to add a fetch step. We can use fetchall() to get all records, fetchone() for a single record or fetchmany() for a specific number of records.

You do not have to use commit() for select statements.

```
sql = 'select * from people'
cursor.execute(sql)
results = cursor.fetchall()
conn.close()
```

```
sql = 'select * from people'
cursor.execute(sql)
results = cursor.fetchone()
conn.close()
```

```
sql = 'select * from people'
cursor.execute(sql)
results = cursor.fetchmany(20)
conn.close()
```

In this example we pull all of the records from the people table. We print out that results without modification, and then we loop through the list with a for loop to print out the values at each index.

```
import sqlite3

conn = sqlite3.connect('example.db')
cursor = conn.cursor()

sql = 'select * from people'
cursor.execute(sql)
results = cursor.fetchall()

conn.close()

print(results)

for x in results:
    print(x)
```

```
[('bob', '33'), ('sue', 23), ('tim', 12)]
('bob', '33')
('sue', 23)
('tim', 12)
```

Filters in Statements

When using Where and other filters in the SQL Statements you use a question mark as with inserts and add the value to be filtered agains in the execute().

```
sql = 'select * from table where column > ?'
cursor.execute(sql,(value,))
```

```
import sqlite3

def find(age):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    sql = 'select * from people where age > ?'
    cursor.execute(sql,(age,))
    results = cursor.fetchall()

    conn.close()

    return results

while True:
    age = input('Minimum Age: ')
    results = find(age)

    print(results)
```

```
Minimum Age: 12
[('bob', '33'), ('timmy', '55'), ('mack', '22')]
Minimum Age: 44
[('timmy', '55')]
Minimum Age: 30
[('bob', '33'), ('timmy', '55')]
Minimum Age: 10
[('bob', '33'), ('timmy', '55'), ('sally', '12'), ('mack', '22')]
```

Update

Updating a record is like using a Filter in a Select statement, but the record will be modified. It is important that you only modify the proper record or records. We can use the ROWID or the Primary Key to act as a unique identifier for a record.

This example fetches the rowid, name, and age columns. It then prints the results to the screen. We then use the input() function to ask which record should be updated, and then asked for a new name. We then send those values to the update() function.

```
import sqlite3

def find():
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    sql = 'select rowid,name,age from people'
    cursor.execute(sql)
    results = cursor.fetchall()

    conn.close()

    return results

def update(id,name):
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    sql = 'update people set name = ? where rowid = ?'
    cursor.execute(sql,(name,id))

    conn.commit()
    conn.close()

while True:
    results = find()
    print(results)

    id = input('ID to Change: ')
    name = input('New Name: ')

    update(id,name)
```

```
[(1, 'bob', '33'), (2, 'philly', 23), (3, 'tim', 12), (4, 'timmy', '55'), (5,
'Sally', '12'), (6, 'mack', '22')]
ID to Change: 2
New Name: PAUL
[(1, 'bob', '33'), (2, 'PAUL', 23), (3, 'tim', 12), (4, 'timmy', '55'), (5,
'Sally', '12'), (6, 'mack', '22')]
```

Deleting a Record

Deleting records is like the other filtered statements. It is recommended you use a ROWID or Primary Key as an identifier for the record to be deleted.

```
    conn = sqlite3.connect('example.db')
    cursor = conn.cursor()

    sql = 'delete from table where rowid = ?'
    cursor.execute(sql,(id, ))

    conn.commit()
    conn.close()
```