Mid Journey V4 prompt: database, server, sunny, 8k, photo by Nikon --ar 16:9

Lecture 7-2: Database

Dr. James Chen, Animal Data Scientist, School of Animal Sciences

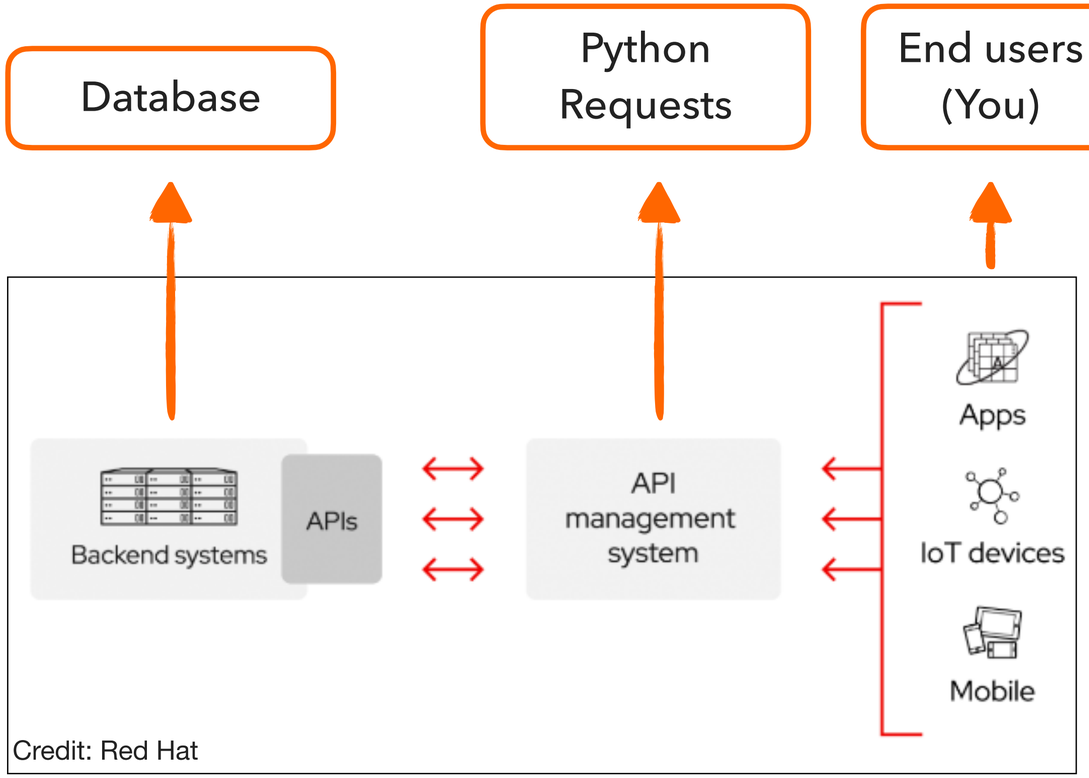
2023 APSC-5984 SS: Agriculture Data Science

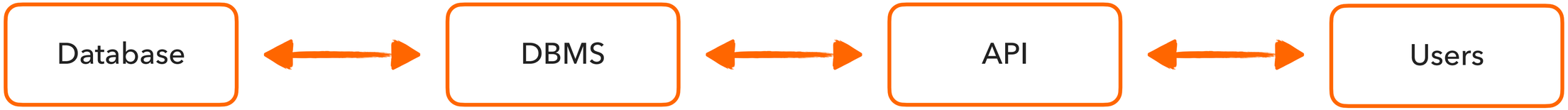
What Is Database?

Oracle https://www.oracle.com/database/what-is-database/

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

Data within the most common types of databases in operation today is typically modeled in rows and columns in a series of tables to make processing and data querying efficient. The data can then be easily accessed, managed, modified, updated, controlled, and organized. Most databases use structured query language (SQL) for writing and querying data



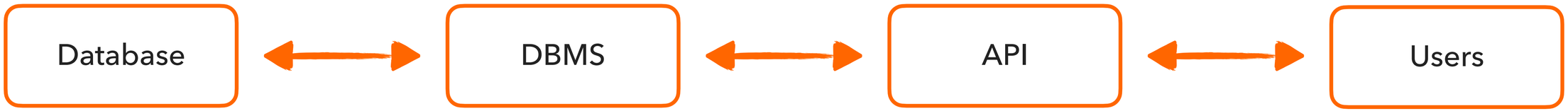


Database Management System (DBMS)

Oracle https://www.oracle.com/database/what-is-database/

A database typically requires a comprehensive database software program known as a database management system (DBMS). A DBMS serves as an interface between the database and its end users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized. A DBMS also facilitates oversight and control of databases, enabling a variety of administrative operations such as performance monitoring, tuning, and backup and recovery.

Some examples of popular database software or DBMSs include MySQL, Microsoft Access, Microsoft SQL Server, FileMaker Pro, Oracle Database, and dBASE.



SQLite

SQLite

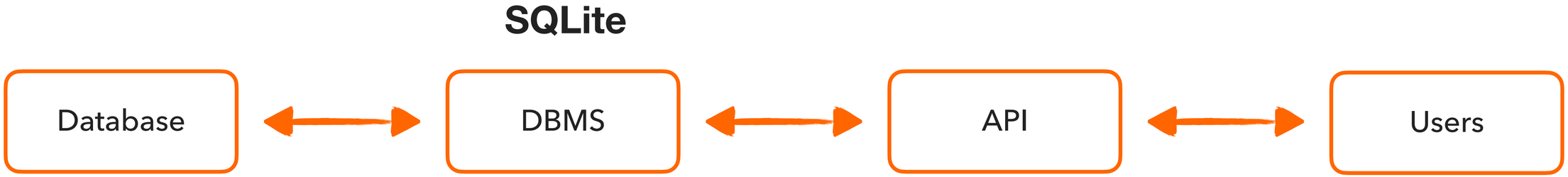
Home About Documentation Download License Support Purchase

What Is SQLite?

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. More Information...

The SQLite file format is stable, cross-platform, and backwards compatible and the developers pledge to keep it that way through the year 2050. SQLite database files are commonly used as containers to transfer rich content between systems [1] [2] [3] and as a long-term archival format for data [4]. There are over 1 trillion (1e12) SQLite databases in active use [5].

SQLite source code is in the public-domain and is free to everyone to use for any purpose.



Python Sqlite3

3. SQLite3

After we learn how to interact with an existing database through API, we can also build our own database using our own data. In this section, we will use `sqlite3` to build a database and query the data.

```
import sqlite3
```

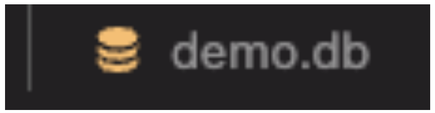
Create a database

Creating a database is as simple as creating a file. We can use `sqlite3.connect()` to create a database file.

```
conn = sqlite3.connect("demo.db") # conn stands for connection
```

```
import sqlite3
conn = sqlite3.connect("demo.db")
```

Open a file stream



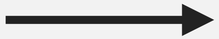
Create a Table

Create a table named ‘users’

Like we learned in the previous section, a database can contain multiple tables (or surveys, reports). To craete a table, we need to specify the name of the columns and the data type. We also need to specify the primary key, which is a unique identifier for each row. We can create a table named `users` with the following columns:

- `id` - INTEGER (PRIMARY KEY)
- `name` - TEXT
- `gender` - TEXT
- `age` - INTEGER

```
cur = conn.cursor() # cur stands for cursor
cur.execute(
    """
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        gender TEXT,
        age INTEGER
    )
    """
)
```



```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    gender TEXT,
    age INTEGER
)
```

Table name

Column names

Constraint

Data type

Inspect the Table

PRAGMA: from the Greek word meaning action

http://archive.adaic.com/standards/83rat/html/ratl-02-01.html

Check the table columns

Table name

```
cur.execute("PRAGMA table_info(users)")
cur.fetchall()
```

required primary key (index)

```
[(0, 'id', 'INTEGER', 0, None, 1),
 (1, 'name', 'TEXT', 1, None, 0),
 (2, 'gender', 'TEXT', 0, None, 0),
 (3, 'age', 'INTEGER', 0, None, 0)]
```

If you think the command is too complicated, we can use a function to simplify the process.

```
def list_cols(cur, table):
    cur.execute("PRAGMA table_info(%s)" % table)
    return display([x[1:3] for x in cur.fetchall()])

list_cols(cur, "users")
```

```
[('id', 'INTEGER'), ('name', 'TEXT'), ('gender', 'TEXT'), ('age', 'INTEGER')]
```

Save the Database

We can use `conn.commit()` to save the changes to the database. Like how we deal with a file, we need to close `conn.close()` the database after we are done with it.

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

Or we can use a `with` statement to automatically close the database after we are done with it.

```
with sqlite3.connect("demo.db") as conn:
    cur = conn.cursor()
    list_cols(cur, "users")
```

```
[('id', 'INTEGER'), ('name', 'TEXT'), ('gender', 'TEXT'), ('age', 'INTEGER')]
```

Function definition:

```
def list_cols(cur, table):
    cur.execute("PRAGMA table_info(%s)" % table)
    return display([x[1:3] for x in cur.fetchall()])
```

Add Information: Data Insertion

Insert data

There are two ways to insert data into a table:

id name gender age

Provide information for all columns

Table name

- `INSERT INTO users VALUES (1, 'John', 'M', 20)`

Provide information for some columns

Selected columns

- `INSERT INTO users (name, gender) VALUES ('John', 'M')`

```
cur.execute("INSERT INTO users VALUES (1, 'Mary', 'F', 25)")
cur.execute("INSERT INTO users (name) VALUES ('John')")
```

```
<sqlite3.Cursor at 0x28fdb7c70>
```

Check the data

every columns

Table name

```
cur.execute("SELECT * FROM users")
cur.fetchall()
```

```
[(1, 'Mary', 'F', 25), (2, 'John', None, None)]
```

Or wrap it up in a function

```
def print_table(cur, table):
    cur.execute("SELECT * FROM %s" % table)
    display(cur.fetchall())

print_table(cur, "users")
```

```
[(1, 'Mary', 'F', 25),
 (2, 'John', None, None),
 (3, 'Camille', 'female', 40),
 (4, 'Mike', 'male', 25),
 (5, 'Jason', 'male', 35),
 (6, 'Maria', 'female', 20)]
```

Add Information: Data Insertion (Pandas)

Table columns

```
(0, 'id', 'INTEGER', 0, None, 1),
(1, 'name', 'TEXT', 1, None, 0),
(2, 'gender', 'TEXT', 0, None, 0)
(3, 'age', 'INTEGER', 0, None, 0)
```

You can actually use `df.to_sql()` to insert data into a table. Parameters we need to consider:

- `if_exists`: If the table already exists, we can choose to `replace` the table, or `append` the data to the existing table.
- `index`: whether to include the index of the dataframe as a column in the table.

```
df = pd.DataFrame(
    {
        "name": ["Camille", "Mike", "Jason", "Maria"],
        "gender": ["female", "male", "male", "female"],
        "age": [40, 25, 35, 20],
    }
)
df.to_sql("users", conn, if_exists="append", index=False)  # if_exists="replace"
```

```
4
```

```
print_table(cur, "users")
```

```
[(1, 'Mary', 'F', 25),
 (2, 'John', None, None),
 (3, 'Camille', 'female', 40),
 (4, 'Mike', 'male', 25),
 (5, 'Jason', 'male', 35),
 (6, 'Maria', 'female', 20)]
```

The gender column has inconsistent values

Column Constraints

Add constraints to columns

You might notice that the gender values were not in a consistent format, which should either be [`M`, `F`] or [`Male`, `Female`]. We can use `CHECK` to add constraints to the columns.

Before re-creating the table, we need to drop the table first.

```
cur.execute("DROP TABLE users")
```

```
<sqlite3.Cursor at 0x28fdb7b20>
```

Then create a new table with the constraints:

- id - INTEGER (PRIMARY KEY)
- name - TEXT - NOT NULL
- gender - TEXT - can only be either 'male' or 'female'
- age - INTEGER - must be in the range of 0 to 150
- weight - REAL - must be in the range of 0 to 300

```
cur.execute(
    """
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        gender TEXT CHECK(gender IN ("male", "female")),
        age INTEGER CHECK(age >= 0 AND age <= 150),
        weight REAL CHECK(weight >= 0 AND weight <= 300)
    )
    """
)
```

Column Constraints

Table definition

```
cur.execute(
    """
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        gender TEXT CHECK(gender IN ("male", "female")),
        age INTEGER CHECK(age >= 0 AND age <= 150),
        weight REAL CHECK(weight >= 0 AND weight <= 300)
    )
    """
)
```

Valid insertion

```
cur.execute("INSERT INTO users VALUES (1, 'Mary', 'female', 25, 150)")
```

```
<sqlite3.Cursor at 0x28fdb7b20>
```

```
print_table(cur, "users")
```

```
[(1, 'Mary', 'female', 25, 150.0)]
```

Error insertion

```
cur.execute("INSERT INTO users VALUES (2, 'Mary', 'female', -3, 200)")
```

```
---------------------------------------------------------------------------

IntegrityError                            Traceback (most recent call last)

Cell In[114], line 1
----> 1 cur.execute("INSERT INTO users VALUES (2, 'Mary', 'female', -3, 200)")


IntegrityError: CHECK constraint failed: age >= 0 AND age <= 150
```

Error insertion

```
cur.execute("INSERT INTO users VALUES (2, 'Mary', 'f', 25, 150)")
```

```
---------------------------------------------------------------------------

IntegrityError                            Traceback (most recent call last)

Cell In[112], line 1
----> 1 cur.execute("INSERT INTO users VALUES (2, 'Mary', 'f', 25, 150)")


IntegrityError: CHECK constraint failed: gender IN ("male", "female")
```

Complete Code

Complete code of constructing a table and data insertion

Creating the table

Data insertion

```
with sqlite3.connect("demo.db") as conn:
    cur = conn.cursor()
    cur.execute(
    """
    CREATE TABLE users (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        gender TEXT CHECK(gender IN ("male", "female")),
        age INTEGER CHECK(age >= 0 AND age <= 150),
        weight REAL CHECK(weight >= 0 AND weight <= 300)
    )
    """)

    df = pd.DataFrame(
        {
            "name": ["Camille", "Mike", "Jason", "Maria"],
            "gender": ["female", "male", "male", "female"],
            "age": [40, 25, 35, 20],
            "weight": [150, 200, 180, 120],
        }
    )
    df.to_sql("users", conn, if_exists="append", index=False)
    print_table(cur, "users")
```

Reference Integrity

Create another table “walks”

We already have a table `users` to define users' information. Now, let's create another table `walks` to record the walking activity.

```
conn = sqlite3.connect("demo.db")
cur = conn.cursor()
cur.execute(
    """
    CREATE TABLE walks (
        id INTEGER PRIMARY KEY,
        user_id INTEGER NOT NULL,
        date TEXT NOT NULL,
        distance FLOAT NOT NULL,
        duration INTEGER NOT NULL,
        FOREIGN KEY (user_id) REFERENCES users (id))
    """
)
```

→ Define the integrity

Reference Integrity

Data insertion

```
data = pd.DataFrame(
    data={
        "user_id": [1, 1, 1, 2, 2, 3, 3, 4, 4, 4],
        "date": [
            "02-26-2023",
            "02-27-2023",
            "02-28-2023",
            "02-26-2023",
            "02-27-2023",
            "02-26-2023",
            "02-27-2023",
            "02-26-2023",
            "02-27-2023",
            "02-28-2023",
        ],
        "distance": [1.2, 1.5, 1.7, 2.2, 2.5, 3.2, 3.5, 4.2, 4.5, 4.7],
        "duration": [30, 40, 50, 60, 70, 80, 90, 100, 110, 120],
    }
)
data.to_sql("walks", conn, if_exists="append", index=False)
```

Foreign key

Must match `id` in the table `users`

Print the table content

```
print_table(cur, "walks")
```

```
[(1, '02-26-2023', 1.2, 30),
 (1, '02-27-2023', 1.5, 40),
 (1, '02-28-2023', 1.7, 50),
 (2, '02-26-2023', 2.2, 60),
 (2, '02-27-2023', 2.5, 70),
 (3, '02-26-2023', 3.2, 80),
 (3, '02-27-2023', 3.5, 90),
 (4, '02-26-2023', 4.2, 100),
 (4, '02-27-2023', 4.5, 110),
 (4, '02-28-2023', 4.7, 120)]
```

Reference Integrity - JOIN

Merge two tables, “users” and “walks”, together

SQLite3

```
cur.execute(
    """
    SELECT * FROM walks JOIN users
    ON walks.user_id = users.id
    """
)
cur.fetchall()
```

Table (left) Table (right)

```
[(1, '02-26-2023', 1.2, 30, 1, 'Camille', 'female', 40, 150.0),
 (1, '02-27-2023', 1.5, 40, 1, 'Camille', 'female', 40, 150.0),
 (1, '02-28-2023', 1.7, 50, 1, 'Camille', 'female', 40, 150.0),
 (2, '02-26-2023', 2.2, 60, 2, 'Mike', 'male', 25, 200.0),
 (2, '02-27-2023', 2.5, 70, 2, 'Mike', 'male', 25, 200.0),
 (3, '02-26-2023', 3.2, 80, 3, 'Jason', 'male', 35, 180.0),
 (3, '02-27-2023', 3.5, 90, 3, 'Jason', 'male', 35, 180.0),
 (4, '02-26-2023', 4.2, 100, 4, 'Maria', 'female', 20, 120.0),
 (4, '02-27-2023', 4.5, 110, 4, 'Maria', 'female', 20, 120.0),
 (4, '02-28-2023', 4.7, 120, 4, 'Maria', 'female', 20, 120.0)]
```

Pandas

```
df_users = pd.read_sql("SELECT * FROM users", conn)
df_walks = pd.read_sql("SELECT * FROM walks", conn)
pd.merge(df_walks, df_users, left_on="user_id", right_on="id")
```

	user_id	date	distance	duration	id	name	gender	age	weight
0	1	02-26-2023	1.2	30	1	Camille	female	40	150.0
1	1	02-27-2023	1.5	40	1	Camille	female	40	150.0
2	1	02-28-2023	1.7	50	1	Camille	female	40	150.0
3	2	02-26-2023	2.2	60	2	Mike	male	25	200.0
4	2	02-27-2023	2.5	70	2	Mike	male	25	200.0
5	3	02-26-2023	3.2	80	3	Jason	male	35	180.0
6	3	02-27-2023	3.5	90	3	Jason	male	35	180.0
7	4	02-26-2023	4.2	100	4	Maria	female	20	120.0
8	4	02-27-2023	4.5	110	4	Maria	female	20	120.0
9	4	02-28-2023	4.7	120	4	Maria	female	20	120.0

SQLite3 VS. Pandas

Sorting

SQLite3

column

```
cur.execute("SELECT * FROM users ORDER BY age DESC")  # or ASC
cur.fetchall()
```

```
[(1, 'Camille', 'female', 40, 150.0),
 (3, 'Jason', 'male', 35, 180.0),
 (2, 'Mike', 'male', 25, 200.0),
 (4, 'Maria', 'female', 20, 120.0)]
```

Pandas

```
df_users.sort_values(by="age", ascending=False)
```

	id	name	gender	age	weight
0	1	Camille	female	40	150.0
2	3	Jason	male	35	180.0
1	2	Mike	male	25	200.0
3	4	Maria	female	20	120.0

Grouping

SQLite3

func. column

```
cur.execute(
    """
    SELECT user_id, avg(distance), sum(duration), count(distance)
    FROM walks GROUP BY user_id
    """
)
cur.fetchall()
```

```
[(1, 1.4666666666666668, 120, 3),
 (2, 2.35, 130, 2),
 (3, 3.35, 170, 2),
 (4, 4.466666666666666, 330, 3)]
```

Pandas

```
df_walks.groupby("user_id").aggregate(
    distance_mean=("distance", "mean"),
    duration_sum=("duration", "sum"),
    distance_count=("distance", "count"),
)
```

	distance_mean	duration_sum	distance_count
user_id			
1	1.466667	120	3
2	2.350000	130	2
3	3.350000	170	2
4	4.466667	330	3

SQLite3 VS. Pandas

Filtering

SQLite3

Subset columns

Filtering criteria

```
cur.execute("SELECT age, gender FROM users WHERE age > 30")
cur.fetchall()
```

```
[(40, 'female'), (35, 'male')]
```

Pandas

```
df_users.loc[:, ["age", "gender"]].query("age > 30")
```

	age	gender
0	40	female
2	35	male

SQLite3

```
cur.execute(
    """
    SELECT name, weight FROM users
    WHERE name LIKE '%m%'
    OR name LIKE '%n%'
    """
)
cur.fetchall()
```

```
[('Camille', 150.0), ('Mike', 200.0), ('Jason', 180.0), ('Maria', 120.0)]
```

Pandas

```
df_users.loc[:, ["name", "weight"]].query(
    """
    name.str.upper().str.contains('M') |\
    name.str.upper().str.contains('N')
    """
)
```

	name	weight
0	Camille	150.0
1	Mike	200.0
2	Jason	180.0
3	Maria	120.0

Master Table

sqlite_master

`sqliet_master` is a system table that contains the information of all tables in the database. We can use `SELECT * FROM sqlite_master` to get the information of all tables. The output will look like this:

SELECT * FROM sqlite_master

- `type` : the type of the object. In this case, it is `table` .
- `name` : the name of the table.
- `tbl_name` : the name of the table.
- `rootpage` : the page number of the root b-tree page for the table.
- `sql` : the SQL statement used to create the table.

```
cur.execute("SELECT * FROM sqlite_master")
cur.fetchall()
```

```
[('table',
  'users',
  'users',
  2,
  'CREATE TABLE users (\n        id INTEGER PRIMARY KEY,\n        name TEXT NOT NULL,\n        gende
 ('table',
  'walks',
  'walks',
  3,
  'CREATE TABLE "walks" (\n"user_id" INTEGER,\n  "date" TEXT,\n  "distance" REAL,\n  "duration" INTE
```

Or simply list all the tables in the database.

```
cur.execute("SELECT name FROM sqlite_master WHERE type='table'")
cur.fetchall()
```

```
[('users',), ('walks',)]
```

Collection of Functions


```
def list_cols(cur, table):
    cur.execute("PRAGMA table_info(%s)" % table)
    cols = [x[1:3] for x in cur.fetchall()]
    return display(cols)

def print_table(cur, table):
    cur.execute("SELECT * FROM %s" % table)
    output = cur.fetchall()
    display(output)

def clean_table(cur, table):
    cur.execute("DELETE FROM %s" % table)

def drop_table(cur, table):
    cur.execute("DROP TABLE %s" % table)
```

Exercise

1. Define two SQL tables that have the same data structure of the billboard data (song and rank)

2. Define your own table filtering function