Markdown

# Guía de Ejemplos Básicos de SQLite con Python para Data Science

¡Buenos días! Claro que sí, con gusto te ayudaré a generar ejemplos con las instrucciones básicas de SQLite para tu curso de Data Science con Python en Visual Studio Code. Utilizaremos el modelo de datos de la base de datos Chinook que has proporcionado.

Aquí tienes una serie de ejemplos que cubren las operaciones fundamentales:

\*\*Nota preliminar:\*\* Para que estos ejemplos funcionen, necesitarás tener un archivo de base de datos SQLite llamado `chinook.db` (o el nombre que tenga tu base de datos) en el mismo directorio que tu script de Python, o bien proporcionar la ruta completa al archivo. El módulo `sqlite3` viene incluido con Python, por lo que no necesitas instalar nada adicional si ya tienes Anaconda.

---

## 1. Configuración del entorno y Conexión a la Base de Datos

Primero, vamos a establecer la conexión a la base de datos SQLite usando Python.

```python

import sqlite3

import pandas as pd # Usaremos pandas para mostrar los resultados de forma amigable

# Función para ejecutar queries y devolver un DataFrame de pandas

def sql\_query(query, db\_name="chinook.db"):

"""

Ejecuta una query SQL en la base de datos especificada y devuelve los resultados en un DataFrame.

"""

try:

# Conectar a la base de datos SQLite

conn = sqlite3.connect(db\_name)

# Ejecutar la query y cargar los resultados en un DataFrame de pandas

df = pd.read\_sql\_query(query, conn)

return df

except sqlite3.Error as e:

print(f"Error al ejecutar la query: {e}")

return None

finally:

# Cerrar la conexión a la base de datos

if conn:

conn.close()

# Ejemplo de uso de la función (más adelante la usaremos para cada query)

# query\_inicial = "SELECT name FROM sqlite\_master WHERE type='table';"

# tablas = sql\_query(query\_inicial)

# print("Tablas en la base de datos:")

# if tablas is not None:

# print(tablas)

**Explicación:**

* **import sqlite3**: Importa el módulo necesario para interactuar con bases de datos SQLite.
* **import pandas as pd**: Pandas es una librería muy útil para el análisis de datos y nos permite visualizar los resultados de las queries de manera tabular y clara.
* **sql\_query(query, db\_name="chinook.db")**: Esta función encapsula la lógica de conexión, ejecución de la query y cierre de la conexión.
  + **sqlite3.connect(db\_name)**: Establece una conexión con el archivo de la base de datos. Si el archivo no existe, SQLite lo creará.
  + **pd.read\_sql\_query(query, conn)**: Ejecuta la query SQL (query) sobre la conexión (conn) y devuelve los resultados directamente en un DataFrame de Pandas.
  + **conn.close()**: Cierra la conexión a la base de datos. Es importante cerrar las conexiones para liberar recursos.
  + El bloque try...except...finally maneja posibles errores durante la ejecución y asegura que la conexión se cierre siempre.

**2. Modelo de Datos**

El modelo de datos que has proporcionado (asumiendo la imagen chinook\_data\_model.png) muestra varias tablas interconectadas. Las principales para nuestros ejemplos serán:

* tracks: Contiene información sobre las canciones (nombre, álbum, compositor, duración, etc.).
* albums: Información sobre los álbumes.
* artists: Información sobre los artistas.
* genres: Géneros musicales.
* customers: Información de los clientes.
* invoices: Facturas de los clientes.
* invoice\_items: Detalle de los ítems en cada factura.

**3. Queries**

Ahora vamos con los ejemplos de cada tipo de instrucción SQL.

**3.1 Primera query (Como la que mostraste)**

Esta query selecciona todas las columnas y todas las filas de la tabla tracks.

**SQL:**

SQL

SELECT \*

FROM tracks;

**Python:**

Python

import sqlite3

import pandas as pd

def sql\_query(query, db\_name="chinook.db"):

try:

conn = sqlite3.connect(db\_name)

df = pd.read\_sql\_query(query, conn)

return df

except sqlite3.Error as e:

print(f"Error al ejecutar la query: {e}")

return None

finally:

if conn:

conn.close()

query\_tracks = """

SELECT \*

FROM tracks;

"""

df\_tracks = sql\_query(query\_tracks)

if df\_tracks is not None:

print("Todas las pistas (primeras 5 para brevedad):")

print(df\_tracks.head())

print(f"\nDimensiones del DataFrame: {df\_tracks.shape}")

**Explicación:**

* **SELECT \***: Indica que queremos seleccionar todas las columnas.
* **FROM tracks**: Especifica que queremos obtener los datos de la tabla tracks.
* El resultado será una tabla con todas las canciones y toda su información asociada.

**3.2 SELECT**

La cláusula SELECT se usa para especificar las columnas que quieres obtener de una tabla.

**SQL:**

SQL

SELECT Name, Composer, UnitPrice

FROM tracks;

**Python:**

Python

query\_select\_specific = """

SELECT Name, Composer, UnitPrice

FROM tracks;

"""

df\_select\_specific = sql\_query(query\_select\_specific)

if df\_select\_specific is not None:

print("\nNombre, Compositor y Precio Unitario de las pistas (primeras 5):")

print(df\_select\_specific.head())

**Explicación:**

* **SELECT Name, Composer, UnitPrice**: En lugar de \*, especificamos las columnas exactas que nos interesan: Name (nombre de la canción), Composer (compositor) y UnitPrice (precio unitario).
* **FROM tracks**: Seguimos obteniendo los datos de la tabla tracks.
* El resultado será una tabla solo con estas tres columnas para todas las canciones.

**3.3 LIMIT**

La cláusula LIMIT se usa para restringir el número de filas devueltas por la query.

**SQL:**

SQL

SELECT TrackId, Name, AlbumId

FROM tracks

LIMIT 10;

**Python:**

Python

query\_limit = """

SELECT TrackId, Name, AlbumId

FROM tracks

LIMIT 10;

"""

df\_limit = sql\_query(query\_limit)

if df\_limit is not None:

print("\nLas primeras 10 pistas (TrackId, Name, AlbumId):")

print(df\_limit)

**Explicación:**

* **SELECT TrackId, Name, AlbumId**: Seleccionamos el ID de la pista, el nombre y el ID del álbum.
* **FROM tracks**: De la tabla tracks.
* **LIMIT 10**: Indica que solo queremos obtener las primeras 10 filas que cumplan con los criterios de la query.
* Esto es útil para previsualizar datos sin cargar la tabla completa.

**3.4 DISTINCT**

La cláusula DISTINCT se usa para devolver solo valores únicos (no repetidos) para una columna o conjunto de columnas especificadas.

**SQL:**

SQL

SELECT DISTINCT Composer

FROM tracks

WHERE Composer IS NOT NULL

ORDER BY Composer

LIMIT 10;

**Python:**

Python

query\_distinct = """

SELECT DISTINCT Composer

FROM tracks

WHERE Composer IS NOT NULL -- Excluimos los valores nulos para este ejemplo

ORDER BY Composer -- Opcional: ordenamos para ver mejor los distintos

LIMIT 10;

"""

df\_distinct = sql\_query(query\_distinct)

if df\_distinct is not None:

print("\n10 Compositores distintos (excluyendo nulos y ordenados):")

print(df\_distinct)

**Explicación:**

* **SELECT DISTINCT Composer**: Queremos obtener una lista de compositores, pero cada compositor solo debe aparecer una vez.
* **FROM tracks**: De la tabla tracks.
* **WHERE Composer IS NOT NULL**: Excluimos compositores NULL.
* **ORDER BY Composer**: Opcional, para ordenar la lista.
* **LIMIT 10**: Para mostrar solo los primeros 10.

**3.5 WHERE**

La cláusula WHERE se usa para filtrar registros que cumplan una condición específica.

**SQL:**

SQL

SELECT Name, AlbumId, Milliseconds

FROM tracks

WHERE GenreId = 1 AND Milliseconds > 300000; -- GenreId 1 suele ser Rock

**Python:**

Python

query\_where = """

SELECT Name, AlbumId, Milliseconds

FROM tracks

WHERE GenreId = 1 AND Milliseconds > 300000; -- Suponiendo que GenreId 1 es 'Rock'

"""

# Para saber qué GenreId es Rock, podríamos hacer:

# query\_genres = "SELECT \* FROM genres;"

# df\_genres = sql\_query(query\_genres)

# print(df\_genres) # Esto mostraría que GenreId 1 es Rock

df\_where = sql\_query(query\_where)

if df\_where is not None:

print("\nPistas del GenreId 1 (Rock) con duración mayor a 300000 milisegundos (5 minutos):")

print(df\_where.head())

**Explicación:**

* **SELECT Name, AlbumId, Milliseconds**: Columnas a mostrar.
* **FROM tracks**: De la tabla tracks.
* **WHERE GenreId = 1 AND Milliseconds > 300000**: Condición de filtrado.
  + GenreId = 1: Canciones del género Rock (asumiendo).
  + AND: Operador lógico.
  + Milliseconds > 300000: Duración mayor a 5 minutos.

**3.6 ORDER BY**

La cláusula ORDER BY se usa para ordenar el conjunto de resultados.

**SQL:**

SQL

SELECT Name, Milliseconds, UnitPrice

FROM tracks

WHERE GenreId = 1

ORDER BY Milliseconds DESC, Name ASC

LIMIT 10;

**Python:**

Python

query\_orderby = """

SELECT Name, Milliseconds, UnitPrice

FROM tracks

WHERE GenreId = 1 -- Filtramos por Rock

ORDER BY Milliseconds DESC, Name ASC -- Ordena por duración (desc), luego por nombre (asc)

LIMIT 10;

"""

df\_orderby = sql\_query(query\_orderby)

if df\_orderby is not None:

print("\nLas 10 pistas de Rock más largas, ordenadas por duración (desc) y luego por nombre (asc):")

print(df\_orderby)

**Explicación:**

* **ORDER BY Milliseconds DESC, Name ASC**:
  + Milliseconds DESC: Ordena por duración descendente.
  + Name ASC: Para empates, ordena por nombre ascendente.

**3.7 Agregaciones (COUNT, SUM, AVG, MIN, MAX)**

Funciones que realizan cálculos sobre un conjunto de valores.

**SQL:**

SQL

SELECT

COUNT(TrackId) AS TotalTracks,

SUM(Milliseconds) / 60000.0 AS TotalDurationMinutes,

AVG(UnitPrice) AS AveragePrice,

MIN(Bytes) AS MinSizeBytes,

MAX(Bytes) AS MaxSizeBytes

FROM tracks

WHERE GenreId = 1; -- Agregaciones para el género Rock

**Python:**

Python

query\_aggregations = """

SELECT

COUNT(TrackId) AS TotalTracks,

SUM(Milliseconds) / 60000.0 AS TotalDurationMinutes,

AVG(UnitPrice) AS AveragePrice,

MIN(Bytes) AS MinSizeBytes,

MAX(Bytes) AS MaxSizeBytes

FROM tracks

WHERE GenreId = 1; -- Agregaciones para el GenreId 1 (Rock)

"""

df\_aggregations = sql\_query(query\_aggregations)

if df\_aggregations is not None:

print("\nAgregaciones para las pistas de Rock (GenreId 1):")

print(df\_aggregations)

**Explicación:**

* **COUNT(TrackId) AS TotalTracks**: Cuenta el número total de pistas.
* **SUM(Milliseconds) / 60000.0 AS TotalDurationMinutes**: Suma la duración y convierte a minutos.
* **AVG(UnitPrice) AS AveragePrice**: Precio unitario promedio.
* **MIN(Bytes) AS MinSizeBytes**: Tamaño mínimo en bytes.
* **MAX(Bytes) AS MaxSizeBytes**: Tamaño máximo en bytes.
* Las agregaciones se calculan para las canciones de Rock.

**3.8 GROUP BY**

Agrupa filas con los mismos valores para aplicar funciones de agregación a cada grupo.

**SQL:**

SQL

SELECT

g.Name AS GenreName,

COUNT(t.TrackId) AS NumberOfTracks,

AVG(t.Milliseconds) / 1000.0 AS AverageDurationSeconds

FROM tracks t

JOIN genres g ON t.GenreId = g.GenreId

GROUP BY g.Name

ORDER BY NumberOfTracks DESC;

**Python:**

Python

query\_groupby = """

SELECT

g.Name AS GenreName,

COUNT(t.TrackId) AS NumberOfTracks,

AVG(t.Milliseconds) / 1000.0 AS AverageDurationSeconds

FROM tracks t

INNER JOIN genres g ON t.GenreId = g.GenreId

GROUP BY g.Name

ORDER BY NumberOfTracks DESC;

"""

df\_groupby = sql\_query(query\_groupby)

if df\_groupby is not None:

print("\nNúmero de pistas y duración promedio por género (ordenado por más pistas):")

print(df\_groupby.head())

**Explicación:**

* **INNER JOIN genres g ON t.GenreId = g.GenreId**: Se une con la tabla genres para obtener el nombre del género.
* **GROUP BY g.Name**: Agrupa por nombre de género. Las funciones de agregación se aplican a cada género.
* Muestra cada género, número de canciones y duración promedio.

**3.9 JOIN**

Combina filas de dos o más tablas basándose en una columna relacionada.

**SQL:**

SQL

SELECT

t.Name AS TrackName,

a.Title AS AlbumTitle,

ar.Name AS ArtistName

FROM tracks t

INNER JOIN albums a ON t.AlbumId = a.AlbumId

INNER JOIN artists ar ON a.ArtistId = ar.ArtistId

WHERE t.GenreId = 1 -- Rock

ORDER BY ArtistName, AlbumTitle, TrackName

LIMIT 10;

**Python:**

Python

query\_join = """

SELECT

t.Name AS TrackName,

a.Title AS AlbumTitle,

ar.Name AS ArtistName

FROM tracks t

INNER JOIN albums a ON t.AlbumId = a.AlbumId

INNER JOIN artists ar ON a.ArtistId = ar.ArtistId

WHERE t.GenreId = 1

ORDER BY ArtistName, AlbumTitle, TrackName

LIMIT 10;

"""

df\_join = sql\_query(query\_join)

if df\_join is not None:

print("\nPrimeras 10 pistas de Rock con nombre del álbum y del artista:")

print(df\_join)

**Explicación:**

* Se seleccionan datos de tracks, albums y artists.
* **INNER JOIN albums a ON t.AlbumId = a.AlbumId**: Une tracks con albums.
* **INNER JOIN artists ar ON a.ArtistId = ar.ArtistId**: Une albums con artists.
* Muestra información combinada de las tres tablas para pistas de Rock.

**Tipos de JOINs comunes:**

* **INNER JOIN**: Filas con coincidencias en ambas tablas.
* **LEFT JOIN**: Todas las filas de la tabla izquierda, y las coincidentes de la derecha (o NULLs).
* (SQLite no soporta RIGHT JOIN ni FULL OUTER JOIN directamente).

**3.10 VIEW y DROP**

Una VIEW es una query SQL almacenada que actúa como tabla virtual. DROP VIEW la elimina.

**Crear una VIEW SQL:**

SQL

CREATE VIEW RockTracksInfo AS

SELECT

t.Name AS TrackName,

a.Title AS AlbumTitle,

ar.Name AS ArtistName,

g.Name AS GenreName,

t.Milliseconds,

t.UnitPrice

FROM tracks t

INNER JOIN albums a ON t.AlbumId = a.AlbumId

INNER JOIN artists ar ON a.ArtistId = ar.ArtistId

INNER JOIN genres g ON t.GenreId = g.GenreId

WHERE g.Name = 'Rock';

**Consultar la VIEW SQL:**

SQL

SELECT \* FROM RockTracksInfo

WHERE Milliseconds > 300000

ORDER BY ArtistName

LIMIT 5;

**Eliminar una VIEW SQL:**

SQL

DROP VIEW IF EXISTS RockTracksInfo;

**Python (para crear, consultar y eliminar la vista):**

Python

def execute\_sql\_command(command, db\_name="chinook.db"):

"""

Ejecuta un comando SQL que no devuelve un conjunto de resultados.

"""

conn = None

try:

conn = sqlite3.connect(db\_name)

cursor = conn.cursor()

cursor.execute(command)

conn.commit() # Necesario para guardar cambios

print(f"Comando ejecutado exitosamente: {command.splitlines()[0]}...")

except sqlite3.Error as e:

print(f"Error al ejecutar el comando: {e}")

finally:

if conn:

conn.close()

# 1. (Opcional) Eliminar la vista si ya existe

query\_drop\_view\_if\_exists = "DROP VIEW IF EXISTS RockTracksInfo;"

execute\_sql\_command(query\_drop\_view\_if\_exists)

# 2. Crear la Vista

query\_create\_view = """

CREATE VIEW RockTracksInfo AS

SELECT

t.Name AS TrackName,

a.Title AS AlbumTitle,

ar.Name AS ArtistName,

g.Name AS GenreName,

t.Milliseconds,

t.UnitPrice

FROM tracks t

INNER JOIN albums a ON t.AlbumId = a.AlbumId

INNER JOIN artists ar ON a.ArtistId = ar.ArtistId

INNER JOIN genres g ON t.GenreId = g.GenreId

WHERE g.Name = 'Rock';

"""

execute\_sql\_command(query\_create\_view)

# 3. Consultar la Vista

query\_select\_from\_view = """

SELECT TrackName, AlbumTitle, ArtistName, Milliseconds / 60000.0 AS DurationMinutes

FROM RockTracksInfo

WHERE Milliseconds > 300000

ORDER BY ArtistName

LIMIT 5;

"""

df\_from\_view = sql\_query(query\_select\_from\_view)

if df\_from\_view is not None:

print("\nConsulta a la vista 'RockTracksInfo' (canciones de Rock de más de 5 min, primeras 5):")

print(df\_from\_view)

# 4. (Opcional) Eliminar la Vista después de usarla

# query\_drop\_view = "DROP VIEW RockTracksInfo;"

# execute\_sql\_command(query\_drop\_view)

# print("\nVista 'RockTracksInfo' eliminada.")

**Explicación:**

* **execute\_sql\_command**: Función para comandos como CREATE VIEW, DROP VIEW que necesitan conn.commit().
* **CREATE VIEW RockTracksInfo AS ...**: Define la vista.
* **SELECT \* FROM RockTracksInfo ...**: Consulta la vista como una tabla.
* **DROP VIEW IF EXISTS RockTracksInfo;**: Elimina la vista.

**4. Errores comunes (a mencionar en clase)**

* **Errores de sintaxis SQL**: Nombres incorrectos, palabras clave mal escritas.
* **Errores de tipo de datos**: Operaciones entre tipos incompatibles.
* **Errores de lógica en WHERE o JOIN**.
* **Conexión a la base de datos**: Archivo .db no encontrado, permisos.
* **No hacer conn.commit()**: Para operaciones que modifican la base de datos (no aplica a SELECT con pd.read\_sql\_query, pero sí a cursor.execute() para CREATE, INSERT, etc.).

**5. Resumen**

SQL es un lenguaje poderoso para bases de datos relacionales. El módulo sqlite3 en Python permite integrar estas capacidades en proyectos de Data Science, complementado por Pandas para el análisis y visualización.

Espero que esto te sea útil.

\*\*Para convertir este Markdown a DOCX o PDF:\*\*

\* \*\*Online Converters\*\*: Busca "Markdown to PDF converter" o "Markdown to DOCX converter" en Google. Sitios como `cloudconvert.com` o `pandoc.org/try` (para usuarios más avanzados) pueden hacer esto. Simplemente copia y pega el texto Markdown en la herramienta.

\* \*\*VS Code\*\*: Si usas Visual Studio Code, puedes instalar extensiones como "Markdown PDF" o "Markdown All in One". Estas extensiones suelen permitirte hacer clic derecho en un archivo `.md` (guarda el texto de arriba como `guia\_sqlite.md`, por ejemplo) y exportarlo a PDF o HTML.

\* \*\*Typora\*\*: Es un editor de Markdown que permite exportar directamente a PDF y otros formatos.

\* \*\*Google Docs / Microsoft Word\*\*:

\* Copia todo el texto Markdown de arriba.

\* Pega directamente en un nuevo Google Doc o documento de Word.

\* La mayoría del formato básico (encabezados, negritas, listas) debería conservarse o ser fácilmente aplicable. Los bloques de código pueden necesitar un ajuste manual para que se vean bien (por ejemplo, seleccionándolos y aplicando una fuente monoespaciada como Courier New o Consolas, y quizás un color de fondo).