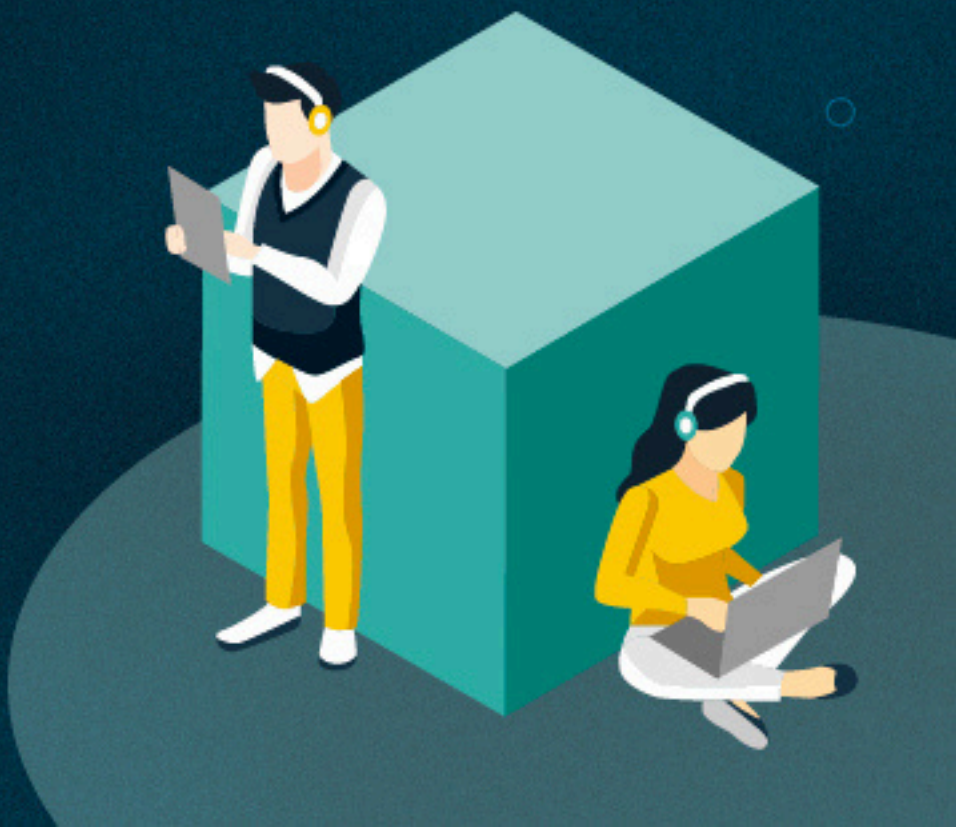
«Talento Tech»

Iniciación a la

Programación con Python

Clase 13



Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 13 | Módulo SQLite 3

Temario:

- Instalación y uso del módulo sqlite3.
- Conexión a una base de datos SQLite.
- Creación de tablas y definición de campos.
- Consultas SQL básicas desde Python: SELECT, INSERT, UPDATE, DELETE.

Objetivos de la Clase

En esta clase, aprenderás a trabajar con **bases de datos SQLite** directamente desde Python, un paso esencial para desarrollar aplicaciones que manejen grandes volúmenes de datos de manera organizada y eficiente. Descubrirás cómo instalar y utilizar el módulo sqlite3 para conectar tus programas con una base de datos SQLite, lo que permitirá almacenar y recuperar información de forma persistente.

Además, explorarás cómo crear bases de datos y definir tablas con campos específicos, entendiendo la importancia de estructurar los datos de manera lógica para facilitar su manipulación. También aprenderás a ejecutar consultas SQL básicas, como **SELECT**, **INSERT**, **UPDATE** y **DELETE**, que te permitirán gestionar y modificar la información almacenada según las necesidades de tu aplicación.

La integración de consultas SQL en tus programas será otro aspecto clave que abordarás, lo que te preparará para diseñar aplicaciones más avanzadas y robustas. Con estas herramientas, darás un gran paso hacia la creación de proyectos con una gestión profesional de la información.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Bienvenido a tu jornada en TalentoLab 🎉

Talento Lab

El día comienza con un correo de Mariana en tu bandeja de entrada. El asunto es claro y directo: “Nueva etapa del proyecto: Bases de datos SQL”. Al abrirlo, descubrís que el cliente ha planteado un nuevo desafío. Mariana te explica que, si bien el sistema actual cumple con su función al almacenar los datos en archivos de texto, necesita algo más robusto y profesional para manejar un volumen creciente de información.

"*Queremos dar el próximo paso. El cliente necesita que el sistema utilice una base de datos SQL para almacenar y gestionar la información. Esto no solo hará que el sistema sea más eficiente, sino que permitirá realizar consultas más rápidas y precisas sobre los datos. Además, podremos manejar grandes volúmenes de información sin problemas.”*

Más tarde, en una reunión con Diego, el desarrollador senior, te da más detalles. "*Vamos a trabajar con SQLite, que es ideal para este tipo de proyectos porque no requiere configuración compleja. Es una base de datos ligera, pero poderosa, y Python tiene todo lo necesario para conectarse y trabajar con ella. Hoy vas a aprender cómo instalarla, crear tablas y ejecutar consultas básicas desde tu programa. Te va a sorprender lo que podés lograr*."

Con este nuevo desafío en puerta, te sentís motivado para aprender una de las herramientas más importantes en el desarrollo de software profesional.

Estructura de la base de datos

En una base de datos, la información se organiza en tablas. Podemos imaginar una tabla como una hoja de cálculo donde cada fila representa un registro (como un producto del inventario) y cada columna representa un campo (por ejemplo, el nombre del producto, el precio o la cantidad disponible). Esta estructura es muy práctica porque facilita la búsqueda, el filtrado y la modificación de datos.

ID Producto	Nombre	Descripción	Cantidad	Precio	Categoría
1	Manzana	Fruta fresca	50	0.5	Frutas
2	Pan	Pan casero	20	1.0	Panadería
3	Leche	Leche descremada	100	0.75	Lácteos
4	Jugo	Jugo de naranja natural	30	1.5	Bebidas

Ejemplo simple de una tabla

Campos y registros en las tablas

En una base de datos, un **campo** es una columna dentro de una tabla que almacena un tipo específico de información sobre los **registros** que están en esa tabla. Por ejemplo, si pensamos en una tabla de productos, cada producto sería un registro y cada campo almacenaría una característica específica de ese producto, como *su nombre, cantidad o precio.*

Los tipos de datos que podemos almacenar en los campos de una tabla son importantes porque determinan qué clase de información puede guardarse en ese campo. Al definir una tabla, tenemos que indicar qué tipo de dato se acepta en cada campo para asegurarnos de que la información se almacene correctamente.

Estos son algunos de los tipos de datos más comunes y útiles para un proyecto de gestión de inventario:

- **Texto:** Este tipo de dato se utiliza para almacenar cadenas de texto. Es útil cuando queremos guardar nombres de productos, descripciones o categorías. Por ejemplo, podríamos tener un campo llamado "nombre" para guardar el nombre del producto ("Manzana", "Leche", etc.) y otro llamado "descripción" para una breve descripción del producto.

- **Números enteros:** Este tipo de dato se utiliza para almacenar números enteros. Es ideal para cantidades de productos o para campos que necesiten valores numéricos sin decimales, como el código de un producto (por ejemplo, un ID único que identifique cada producto). Un campo de tipo entero sería perfecto para registrar cuántas unidades de cada producto tenemos en el inventario.

- **Números de punto flotante:** Cuando necesitamos almacenar precios o valores que incluyan decimales, usamos un tipo de dato que permita números con fracciones. En un sistema de inventario, el precio de un producto es un ejemplo de un valor que suele requerir decimales.

- **Fechas y horas**: A veces es necesario registrar cuándo ocurrió algo, como la fecha en que un producto fue agregado al inventario o la última vez que se actualizó su cantidad. Para esto, usamos un tipo de dato que pueda manejar fechas y horas. Estos campos son útiles para llevar un control de las acciones que realizamos en la base de datos.

Al crear una tabla, decidimos qué tipo de datos vamos a almacenar en cada campo según el tipo de información que necesitamos guardar. De esta forma nos aseguramos de que la base de datos esté organizada y opere con normalidad.

Campos clave.

Ahora vamos a hablar sobre un concepto clave en las bases de datos: el **campo clave**, o también conocido como **clave primaria**. Este campo es uno de los elementos más importantes cuando diseñamos una tabla en nuestra base de datos.

Un campo clave es un campo especial dentro de una tabla que se utiliza para identificar de manera **única** cada registro. Imaginemos que tenemos una tabla donde guardamos todos los productos del inventario. Cada producto tiene su propio nombre, precio, cantidad, etc., pero necesitamos un dato que permita diferenciar cada producto de manera única. Para esto, usamos una clave primaria.

La clave primaria tiene dos características principales:

Unicidad: No puede haber dos registros en la tabla que tengan el mismo valor en el campo clave. Esto es lo que permite identificar de manera única cada producto en el inventario. Por ejemplo, en nuestra tabla de productos, podríamos usar un campo llamado **"ID Producto"**, que asigna un número único a cada producto. Así, aunque tengamos dos productos llamados "Pan", cada uno tendría un número de ID diferente, lo que nos permite diferenciarlos.

No puede ser nulo: El campo clave debe tener siempre un valor. No puede quedar vacío. Si no existiera una clave para un registro, no tendríamos manera de identificarlo claramente dentro de la tabla, lo que generaría problemas al momento de buscar o actualizar información.

Por ejemplo, en nuestra tabla de productos, podemos tener un campo llamado **"ID Producto"** que actúe como clave primaria. Supongamos que tenemos estos productos.

ID Producto	Nombre	Cantidad	Precio
1	Manzana	50	0.5
2	Pan	20	1.0
3	Jugo	30	1.5

En este ejemplo, el campo **”ID Producto”** es la clave primaria. Cada producto tiene un número único que lo diferencia de los demás, incluso si otros datos, como el nombre, llegaran a repetirse. El hecho de que sea único y obligatorio nos asegura que podemos identificar cualquier producto en la tabla sin confusión.

Una de las mayores ventajas de utilizar una clave primaria es que nos permite realizar búsquedas de manera rápida. Al tener un identificador único para cada registro, podemos encontrar lo que buscamos sin tener que comparar otros campos. Además, nos brinda seguridad al momento de actualizar datos. Si necesitamos cambiar la cantidad de un producto o modificar su precio, podemos hacerlo con la tranquilidad de que estamos editando el registro correcto, ya que está identificado de forma única. Por otro lado, la clave primaria también garantiza la **integridad de los datos.** Al no permitir valores repetidos o

vacíos, ayuda a mantener la tabla organizada y libre de errores. Esto hace que sea una herramienta casi indispensable para la correcta gestión de los datos, sobre todo cuando trabajamos con grandes volúmenes de información, como en el caso de un sistema de inventario.

Introducción al módulo SQLite3.

SQLite es un motor de base de datos ligero y autónomo que no requiere configuraciones complejas ni servidores externos para funcionar. Esto lo convierte en una excelente opción para proyectos pequeños y medianos, donde la simplicidad y la eficiencia son clave. En Python, el **módulo sqlite3** permite interactuar fácilmente con bases de datos SQLite, integrando consultas y gestión de datos directamente en nuestros programas.

Una de las grandes ventajas de SQLite es que los datos se almacenan en un archivo único que puede ser fácilmente transferido o respaldado. Además, al ser una **base de datos relacional,** organiza la información en tablas, tal como explicamos antes. Esto facilita la consulta y manipulación de datos.

Al trabajar con sqlite3, podés crear, leer, actualizar y eliminar datos en una base de datos utilizando comandos SQL directamente desde Python. Este enfoque no solo mejora la persistencia de los datos, sino que también permite gestionar grandes volúmenes de información de manera más eficiente que utilizando archivos de texto.

Conexión a una base de datos SQLite

Para trabajar con bases de datos SQLite en Python, es necesario establecer una **conexión**. Esta conexión actúa como un puente entre tu programa y la base de datos, permitiéndote enviar y recibir información. Si la base de datos no existe, SQLite la creará automáticamente en el lugar indicado, lo que simplifica enormemente el proceso de configuración.

La conexión se establece mediante la función **sqlite3.connect()**. Este método requiere que especifiques el nombre del archivo donde se almacenará la base de datos. Por ejemplo, si querés crear una base de datos llamada **productos.db**, simplemente indicás este nombre al método. Una vez creada la conexión, también es necesario cerrarla al final del programa para liberar recursos y asegurar que todos los cambios se guarden correctamente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Veamos un ejemplo básico:

```
import sqlite3

# Establecer la conexión a la base de datos
conexion = sqlite3.connect("productos.db")

print("Conexión establecida exitosamente.")

# Cerrar la conexión
conexion.close()
import sqlite3

# Establecer la conexión a la base de datos
conexion = sqlite3.connect("productos.db")

print("Conexión establecida exitosamente.")

# Cerrar la conexión
conexion.close()
```

En este ejemplo, se establece una conexión con la base de datos llamada productos.db. Si el archivo no existe, será creado en el mismo directorio donde se ejecuta el programa. El mensaje en pantalla confirma que la conexión se realizó correctamente, y al final, se cierra la conexión para evitar problemas futuros.

Cerrar la conexión no solo es una buena práctica, sino que también previene errores relacionados con bloqueos de archivo o recursos no liberados.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Creación de tablas y definición de campos

Una vez establecida la conexión con una base de datos SQLite, el siguiente paso es definir su estructura creando tablas. Una tabla es el componente principal de una base de datos, donde los datos se organizan en filas y columnas. Cada columna representa un campo con un tipo de dato específico, como texto, números o fechas, mientras que cada fila almacena un registro individual.

Para crear una tabla en SQLite desde Python, utilizamos sentencias SQL con el método execute() proporcionado por el objeto cursor. El cursor es una herramienta que nos permite interactuar con la base de datos, ejecutar consultas y recuperar resultados.

A continuación, un ejemplo donde se crea una tabla llamada productos con tres campos: id (clave primaria), nombre (nombre del producto) y precio (precio del producto):

```
import sqlite3

# Establecer la conexión a la base de datos
conexion = sqlite3.connect("productos.db")

# Crear un objeto cursor
cursor = conexion.cursor()

# Crear una tabla
cursor.execute('''
   CREATE TABLE IF NOT EXISTS productos (
       id INTEGER PRIMARY KEY AUTOINCREMENT,
       nombre TEXT NOT NULL,
       precio REAL NOT NULL
   )
''')

print("Tabla 'productos' creada exitosamente.")

# Confirmar los cambios y cerrar la conexión
conexion.commit()
conexion.close()
```

En el ejemplo, la conexión a la base de datos se realiza con **sqlite3.connect()**, creando o abriendo el archivo **productos.db**. Luego, se utiliza el método **cursor()** para obtener un objeto cursor, que es necesario para ejecutar comandos SQL.

La sentencia **CREATE TABLE IF NOT EXISTS** asegura que la tabla se cree solo si no existe ya en la base de datos, evitando errores por duplicados. Dentro de los paréntesis se define la estructura de la tabla:

- **id INTEGER PRIMARY KEY AUTOINCREMENT**: El campo id es un identificador único que se incrementa automáticamente con cada nuevo registro.
- **nombre TEXT NOT NULL**: El campo nombre almacena texto y no puede quedar vacío.
- **precio REAL NOT NULL**: El campo precio almacena números reales y tampoco puede quedar vacío.

Finalmente, se utiliza **commit()** para guardar los cambios realizados en la base de datos y **close()** para cerrar la conexión.

Inserción de datos en una tabla

Una vez creada la tabla, el siguiente paso es agregar datos a la misma. En SQLite, esto se realiza mediante la sentencia SQL INSERT INTO, que permite especificar los valores para cada columna. Este proceso se puede implementar desde Python usando el método **execute()** del cursor, proporcionando los valores de manera directa o a través de parámetros para mayor flexibilidad y seguridad.

El siguiente ejemplo muestra cómo insertar productos en la tabla productos creada anteriormente:

```
import sqlite3

# Conexión a la base de datos
conexion = sqlite3.connect("productos.db")
cursor = conexion.cursor()

# Insertar datos en la tabla
cursor.execute('''
```

```
    INSERT INTO productos (nombre, precio)
    VALUES (?, ?)
''', ("Lápiz", 25.50))

cursor.execute('''
    INSERT INTO productos (nombre, precio)
    VALUES (?, ?)
''', ("Cuaderno", 120.00))

cursor.execute('''
    INSERT INTO productos (nombre, precio)
    VALUES (?, ?)
''', ("Mochila", 890.99))

print("Productos agregados exitosamente.")

# Confirmar los cambios y cerrar la conexión
conexion.commit()
conexion.close()
```

En el ejemplo, se comienza estableciendo la conexión con la base de datos y creando un objeto cursor para interactuar con ella. La sentencia **SQL INSERT INTO** especifica la tabla (productos) y las columnas (nombre y precio) en las que se van a insertar valores.

En lugar de escribir los valores directamente en la consulta, se utiliza el símbolo ? como marcador de posición. Esto es una práctica recomendada porque evita vulnerabilidades como las inyecciones SQL. Los valores reales se pasan como una tupla en el segundo argumento del método execute().

- En el primer llamado a execute(), se inserta el producto "Lápiz" con un precio de 25.50.
- En el segundo y tercer llamados, se agregan otros dos productos: "Cuaderno" y "Mochila", con sus respectivos precios.

Finalmente, usamos el método commit() para guardar los cambios en la base de datos, asegurándose de que los datos se conserven. La conexión se cierra al final con close().

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Consulta de datos con SELECT

La sentencia **SELECT** en SQL se utiliza para recuperar información almacenada en una tabla. Es una de las operaciones más comunes en el manejo de bases de datos y permite obtener datos específicos o todos los registros según los criterios definidos.

En Python, se puede implementar SELECT utilizando el método execute() del cursor, al igual que otras sentencias SQL. Veamos cómo recuperar los datos insertados en la tabla productos.

Ejemplo: Recuperar todos los registros

```
import sqlite3

# Conexión a la base de datos
conexion = sqlite3.connect("productos.db")
cursor = conexion.cursor()

# Recuperar todos los registros de la tabla productos
cursor.execute('SELECT * FROM productos')
productos = cursor.fetchall()

print("\n=== Lista de Productos ===")
for producto in productos:
   print(f"ID: {producto[0]}, Nombre: {producto[1]}, Precio:
${producto[2]:.2f}")

# Cerrar la conexión
conexion.close()
```

El comando SELECT * FROM productos indica que queremos obtener todos los registros de la tabla productos. El asterisco (*) significa "todas las columnas". El método fetchall() recupera todos los resultados de la consulta en forma de lista de tuplas, donde cada tupla representa un registro de la tabla.

En el bucle for, se itera sobre los resultados para mostrar el ID, nombre y precio de cada producto. Esto facilita una visualización clara de los datos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Recuperar registros con filtros

Es posible añadir condiciones a la consulta para recuperar registros específicos usando **WHERE**. Por ejemplo, supongamos que queremos obtener los productos cuyo precio sea mayor a $100.

```
# Recuperar productos con precio mayor a $100
cursor.execute('SELECT * FROM productos WHERE precio > ?',
(100,))productos_filtrados = cursor.fetchall()

print("\n=== Productos con Precio Mayor a $100 ===")

for producto in productos_filtrados:
   print(f"ID: {producto[0]}, Nombre: {producto[1]}, Precio:
${producto[2]:.2f}")
```

La cláusula WHERE permite definir condiciones para filtrar los resultados. En este caso, precio > ? selecciona solo los productos cuyo precio sea mayor a 100. El marcador ? es reemplazado por el valor proporcionado en la tupla (100,), asegurando que la consulta sea segura.

Ordenar los resultados con ORDER BY

Se puede ordenar los resultados utilizando la cláusula **ORDER BY**. Por ejemplo, para mostrar los productos en orden ascendente por precio:

```
# Ordenar los productos por precio (ascendente)
cursor.execute('SELECT * FROM productos ORDER BY precio ASC')
productos_ordenados = cursor.fetchall()

print("\n=== Productos Ordenados por Precio (Ascendente) ===")
for producto in productos_ordenados:
   print(f"ID: {producto[0]}, Nombre: {producto[1]}, Precio:
${producto[2]:.2f}")
```

El uso de **ASC** indica que el orden es ascendente, mientras que **DESC** lo haría descendente.

Seleccionar columnas específicas

Si solo queremos recuperar ciertos campos, podemos especificar los nombres de las columnas en lugar de usar *. Por ejemplo, obtener solo los nombres de los productos:

```
# Obtener solo los nombres de los productos
cursor.execute('SELECT nombre FROM productos')
nombres = cursor.fetchall()

print("\n=== Nombres de Productos ===")
for nombre in nombres:
   print(f"Producto: {nombre[0]}")
```

La sentencia SELECT es casi indispensable para trabajar con bases de datos, y como hemos visto, su flexibilidad permite extraer la información de diferentes formas según las necesidades de cada aplicación.

Actualización de datos con UPDATE.

La sentencia **UPDATE** en SQL se utiliza para modificar registros existentes en una tabla. Es una herramienta esencial cuando necesitamos corregir o actualizar información sin eliminarla ni insertar nuevos datos.

En Python, se puede usar el método execute() para implementar UPDATE. Veamos como utilizarlo:

Ejemplo: Actualizar el precio de un producto

Supongamos que necesitamos modificar el precio de un producto específico en nuestra tabla productos. Por ejemplo, queremos actualizar el precio del producto cuyo ID es 1.

```
import sqlite3

# Conexión a la base de datos
conexion = sqlite3.connect("productos.db")
cursor = conexion.cursor()

# Actualizar el precio de un producto específico
nuevo_precio = 250.0
id_producto = 1
cursor.execute('UPDATE productos SET precio = ? WHERE id = ?',
(nuevo_precio, id_producto))

# Confirmar cambios
conexion.commit()

print(f"Producto con ID {id_producto} actualizado correctamente.")

# Verificar los cambios
cursor.execute('SELECT * FROM productos WHERE id = ?', (id_producto,))
producto_actualizado = cursor.fetchone()

print("\n=== Producto Actualizado ===")
print(f"ID: {producto_actualizado[0]}, Nombre:
{producto_actualizado[1]}, Precio: ${producto_actualizado[2]:.2f}")

# Cerrar la conexión
conexion.close()
```

La sentencia UPDATE se compone de tres partes clave:

- **UPDATE productos** especifica la tabla en la que queremos realizar la actualización.
- **SET precio = ?** define el campo que será modificado y el nuevo valor que tomará.
- **WHERE id = ?** agrega una condición para asegurar que solo se modifique el registro correspondiente. En este caso, se usa el ID del producto como identificador único.

El uso de los marcadores **?** permite realizar consultas parametrizadas, lo que evita vulnerabilidades como las inyecciones SQL.

Tras ejecutar la actualización, es fundamental llamar a conexion.commit() para guardar los cambios en la base de datos. Luego, se verifica el cambio con una consulta SELECT que muestra el registro actualizado.

Actualizar varios registros simultáneamente

También es posible modificar varios registros en una sola operación. Por ejemplo, supongamos que queremos aumentar un 10% al precio de todos los productos que cuestan menos de $100:

```
# Incrementar en un 10% los precios de los productos con precio menor a
$100
cursor.execute('UPDATE productos SET precio = precio * 1.10 WHERE
precio < ?', (100,))

# Confirmar cambios
conexion.commit()

print("Precios actualizados para productos con precio menor a $100.")
```

Este código utiliza la sentencia UPDATE para modificar los precios de los productos en una base de datos que cumplan con una condición específica. En este caso, se busca incrementar un 10% el precio de todos los productos cuyo valor sea menor a $100. La sentencia SQL define la tabla (productos), el campo a modificar (precio) y la operación que debe realizarse (precio * 1.10), además de incluir la condición **WHERE precio < ?**, donde el marcador ? se reemplaza por el valor 100.

Después de ejecutar la operación, el método **conexion.commit()** se utiliza para guardar los cambios realizados en la base de datos. Finalmente, se imprime un mensaje en la consola indicando que los precios de los productos han sido actualizados correctamente. Este enfoque asegura que solo los registros que cumplan con la condición definida sean afectados.

El uso de la cláusula **WHERE** es esencial para evitar modificaciones accidentales en todos los registros de la tabla. **Si WHERE se omite, se actualizarán todos los registros**, lo que puede no ser deseado.

Además, siempre es buena práctica verificar el impacto de la operación con una consulta previa o posterior que muestre los registros afectados, asegurando así la integridad de los datos.

Eliminar registros con DELETE.

La sentencia **DELETE** en SQL se utiliza para eliminar registros específicos de una tabla. Es una herramienta poderosa y debe usarse con cuidado, ya que una vez eliminados los registros, no se pueden recuperar a menos que se haya realizado un respaldo previo. Es fundamental siempre acompañar la instrucción con una condición clara usando **WHERE**, para evitar eliminar toda la información de la tabla.

La estructura básica de DELETE es la siguiente:

```
DELETE FROM nombre_de_tabla WHERE condicion;
```

Aquí, **nombre_de_tabla** es la tabla en la que se van a eliminar los registros, y **condicion** define qué registros serán eliminados.

Si no se especifica un WHERE, se eliminarán todos los registros de la tabla, dejando la estructura intacta pero vacía.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejemplo 1: Eliminando un producto específico

Supongamos que tenemos una tabla llamada productos y queremos eliminar un producto basándonos en su nombre.

```
# Conexión a la base de datos
import sqlite3
conexion = sqlite3.connect("productos.db")
cursor = conexion.cursor()

# Eliminar un producto específico
nombre_producto = "Lápiz"
cursor.execute("DELETE FROM productos WHERE nombre = ?",
(nombre_producto,))

# Confirmar cambios
conexion.commit()

print(f"Producto '{nombre_producto}' eliminado de la base de datos.")

# Cerrar la conexión
conexion.close()
```

En este ejemplo, comenzamos estableciendo la conexión a la base de datos y obteniendo un cursor para ejecutar las consultas. Luego, usamos la sentencia **DELETE** con un **WHERE** para especificar que queremos eliminar el registro donde el campo nombre coincide con el valor de la variable **nombre_producto**. Para prevenir inyecciones SQL, utilizamos un marcador **?** y pasamos el valor como una **tupla**. Finalmente, confirmamos los cambios con **commit** y cerramos la conexión.

Ejemplo 2 : Eliminando productos por precio

En algunos casos, podemos querer eliminar varios registros que cumplan con una condición, como todos los productos con un precio inferior a cierto valor.

```
# Eliminar productos con precio menor a $50
precio_limite = 50
cursor.execute("DELETE FROM productos WHERE precio < ?",
(precio_limite,))

# Confirmar cambios
conexion.commit()

print(f"Productos con precio menor a ${precio_limite} eliminados.")
```

La sentencia DELETE es extremadamente útil para mantener la base de datos limpia y actualizada. Sin embargo, siempre debe usarse con precaución, asegurándose de definir condiciones específicas, para evitar la eliminación accidental de datos valiosos.

Ejercicio Práctico

Al finalizar el día, Diego se acerca con una solicitud concreta:

"*Mariana ha hablado con el cliente, y tienen una tarea específica que quieren resolver ya mismo. Nos pidieron que hagamos una prueba con el sistema SQLite. Necesitamos algo básico pero funcional: registrar productos con su nombre y precio, y luego mostrarlos en pantalla para asegurarnos de que el registro funciona correctamente. ¿Te animás a encargarte de esta parte? Es clave para que avancemos hacia la solución más completa.*"

Necesitaras:

1. Crear una base de datos SQLite llamada productos.db.
2. Definir una tabla llamada productos con los campos id (clave primaria autoincremental), nombre (texto, no nulo) y precio (real, no nulo).
3. Diseñar un programa que permita:
    - Registrar un producto ingresando su nombre y precio.
    - Mostrar todos los productos almacenados en la base de datos.

El sistema debe incluir validaciones básicas:

- El nombre no puede estar vacío.
- El precio debe ser un número mayor que cero.

Materiales y Recursos Adicionales:

Artículos:
python.org: Interfaz para bases de datos SQLite
Datacamp: Cómo trabajar con SQLite en Python

Videos:
Auribox Training: SQLite con Python
Dimas: ¿Cómo usar SQLite3 en PYTHON?

Preguntas para Reflexionar:

1. Aprendiste a crear tablas y realizar consultas básicas. ¿Qué beneficios encontrás en estructurar los datos en tablas con campos definidos en lugar de usar listas o diccionarios?
2. Pensando en las operaciones SQL que vimos (SELECT, INSERT, UPDATE, DELETE), ¿cuál te pareció más intuitiva y por qué?
3. ¿Qué errores podrías prever al trabajar con bases de datos en programas reales y cómo creés que podrías evitarlos usando lo aprendido?
4. ¿Cómo podrías aplicar el uso de bases de datos SQLite en un proyecto más complejo, como el Trabajo Final Integrador?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos Pasos:

La clase siguiente te llevará a profundizar aún más en el manejo de bases de datos, ampliando tus habilidades para realizar operaciones **CRUD** (Crear, Leer, Actualizar y Eliminar) de manera segura y eficiente. Además, aprenderás cómo manejar parámetros en tus consultas para proteger tus programas contra posibles **inyecciones SQL**, una amenaza común en aplicaciones que interactúan con bases de datos.

Exploraremos también el manejo de transacciones, un recurso esencial para garantizar la integridad de los datos en operaciones más complejas. Estos conceptos te ayudarán a escribir código más robusto y profesional, sentando las bases para desarrollar aplicaciones que gestionen información de forma confiable y escalable.

Repasá los conceptos de esta clase y asegurate de comprender cómo estructurar y realizar consultas en SQLite, ya que serán fundamentales para continuar avanzando en el curso.

Buenos Aires Ciudad