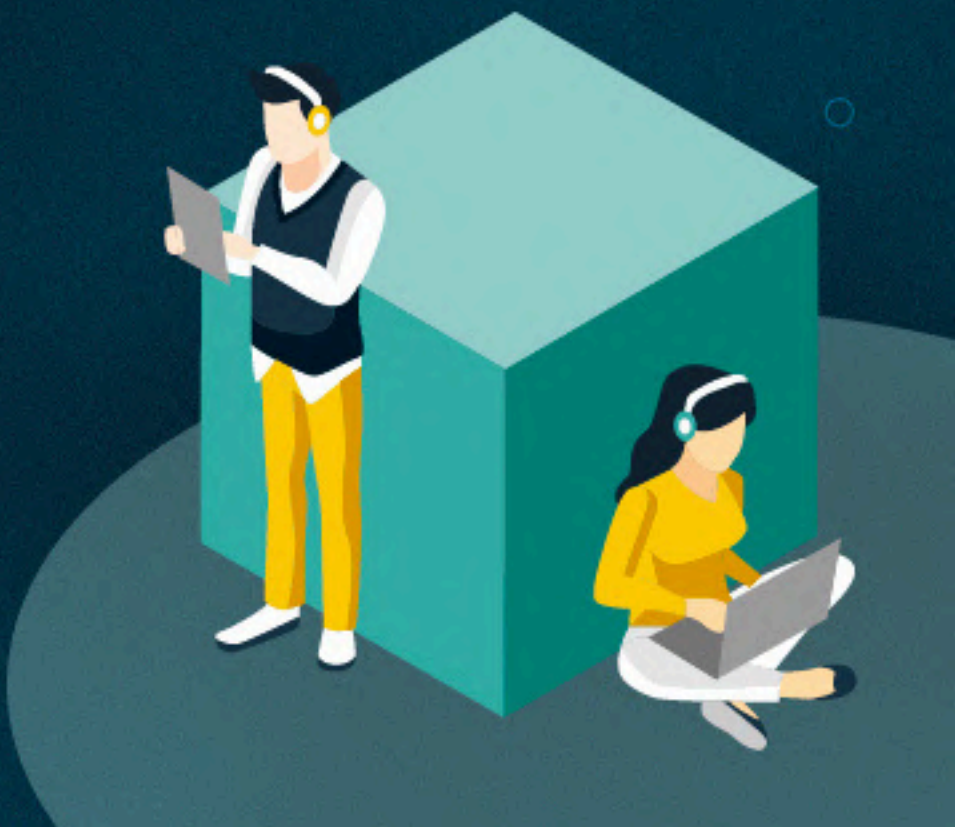
«Talento Tech»

Iniciación a la

Programación con Python

Clase 14



Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 14 | Operaciones CRUD con Bases de Datos

Temario:

- Profundización en operaciones CRUD.
- Uso de parámetros en consultas para prevenir inyecciones SQL.
- Manejo de transacciones y control de errores.

Objetivos de la Clase

En esta clase, te adentrarás en la implementación completa de las **operaciones CRUD** en bases de datos SQLite desde Python. Aprenderás a estructurar un sistema capaz de crear, leer, actualizar y eliminar registros de manera eficiente, lo que te permitirá gestionar información de forma organizada y persistente.

También conocerás las **inyecciones SQL**, un problema de seguridad común en aplicaciones que manejan bases de datos. Entenderás cómo se originan estos ataques y, lo más importante, cómo prevenirlos utilizando **consultas parametrizadas** en Python, garantizando así la integridad y seguridad de los datos.

Por último, abordarás el **manejo de transacciones y el control de errores** al trabajar con bases de datos. Descubrirás cómo asegurarte de que las operaciones realizadas en la base sean consistentes y cómo responder ante posibles fallos mediante el uso de **commit()**, **rollback()** y estructuras de manejo de excepciones. Todo esto sentará una base sólida para que puedas integrar bases de datos de manera segura y confiable en tu Trabajo Final Integrador.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Bienvenido a tu jornada en TalentoLab 🎉

Talento Lab

El día comienza con un nuevo desafío en **TalentoLab**. Revisás tu correo y encontrás un mensaje de Mariana con el asunto: "Fortaleciendo la gestión de datos".

"*Hola!*
El sistema ya almacena información en SQLite, pero necesitamos que los usuarios puedan modificar y eliminar registros de manera segura. Tu tarea es implementar correctamente el CRUD (Crear, Leer, Actualizar y Eliminar), asegurando que las operaciones sean confiables y protegiendo el sistema contra posibles errores o ataques de inyección SQL.

Diego está listo para explicarte cómo manejar transacciones y controlar errores en consultas SQL. Esto garantizará que los datos no se corrompan y que, en caso de fallos, podamos revertir cambios sin afectar la integridad del sistema.

¡Confío en que vas a hacer un gran trabajo!"

Te dirigís a la oficina de Diego, quien te recibe con su mate en la mano.

"*Bienvenido al siguiente nivel. Hoy vas a aprender cómo hacer que nuestro sistema sea más robusto, ejecutando operaciones CRUD de manera segura. También te voy a mostrar cómo las transacciones ayudan a mantener la integridad de los datos. Cuando terminemos, vas a estar listo para poder aplicar todo esto en tu Trabajo Final Integrador*."

Tomás asiento, listo para profundizar en la gestión avanzada de bases de datos en Python.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

CRUD y su importancia en bases de datos

En cualquier aplicación que maneje información estructurada, es fundamental poder crear, leer, actualizar y eliminar registros dentro de una base de datos. Estas cuatro operaciones conforman lo que se conoce como **CRUD** (por sus siglas en inglés: *Create*, *Read*, *Update*, *Delete*), y son la base de cualquier sistema de gestión de datos.

Cuando un usuario registra un producto o cliente en un sistema, en realidad está ejecutando una operación **Create** (Crear) en la base de datos. Cuando consulta información, el sistema está ejecutando una **Read** (Leer). Si necesita modificar algún dato, usa una operación **Update** (Actualizar), y cuando borra un registro, se aplica **Delete** (Eliminar).

Implementar correctamente CRUD no solo permite manipular datos, sino que también garantiza que la información esté organizada, accesible y actualizada. Sin un sistema CRUD correcto, los datos serían estáticos o estarían desorganizados, dificultando su administración.

SQLite facilita la implementación de estas operaciones mediante consultas SQL, permitiendo modificar los registros de manera estructurada. Sin embargo, al trabajar con datos en una base de datos, es fundamental proteger la integridad de la información y prevenir errores o vulnerabilidades.

Vamos a aplicar lo que ya sabemos sobre consultas SQL y la interacción con bases de datos, pero ahora estructurándolo en un sistema funcional que permita realizar las cuatro operaciones de CRUD de manera segura y eficiente.

Implementación de CRUD en SQLite con Python

Hasta ahora, hemos aprendido cómo realizar consultas SELECT, INSERT, UPDATE y DELETE en SQLite. Ahora vamos a integrar todas estas operaciones dentro de un único programa en Python, de manera que podamos gestionar una base de datos de productos, clientes o cualquier otra entidad de interés. La implementación de CRUD en Python sigue una estructura clara:

1. **Conexión a la base de datos:** Se establece una conexión con un archivo SQLite para almacenar los datos.
2. **Creación de una tabla si no existe:** Se define la estructura de los datos (nombre, precio, stock, etc.).

3. **Funciones para cada operación:** Se programan las funciones para **agregar, leer, actualizar y eliminar** registros.
4. **Menú interactivo:** Se permite al usuario seleccionar qué acción desea realizar.

Veamos un ejemplo concreto, escribiendo el código básico de un CRUD para manejar datos de alumnos.

```
import sqlite3

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

# Crear la tabla si no existe
cursor.execute('''
    CREATE TABLE IF NOT EXISTS alumnos (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT NOT NULL,
        edad INTEGER NOT NULL,
        curso TEXT NOT NULL
    )
''')
conexion.commit()

def agregar_alumno(nombre, edad, curso):
    """Inserta un nuevo alumno en la base de datos"""
    cursor.execute("INSERT INTO alumnos (nombre, edad, curso) VALUES (?,
?, ?)", (nombre, edad, curso))
    conexion.commit()
    print("Alumno agregado con éxito.")

def mostrar_alumnos():
    """Muestra todos los alumnos en la base de datos"""
    cursor.execute("SELECT * FROM alumnos")
    alumnos = cursor.fetchall()
    print("\nLista de alumnos:")
```

```
    for alumno in alumnos:
        print(f"ID: {alumno[0]}, Nombre: {alumno[1]}, Edad: {alumno[2]},
Curso: {alumno[3]}")

def actualizar_curso(id_alumno, nuevo_curso):
    """Modifica el curso de un alumno específico"""
    cursor.execute("UPDATE alumnos SET curso = ? WHERE id = ?",
(nuevo_curso, id_alumno))
    conexion.commit()
    print("Curso actualizado correctamente.")

def eliminar_alumno(id_alumno):
    """Elimina un alumno de la base de datos"""
    cursor.execute("DELETE FROM alumnos WHERE id = ?", (id_alumno,))
    conexion.commit()
    print("Alumno eliminado correctamente.")

# Menú interactivo
while True:
    print("\nGestión de Alumnos")
    print("1. Agregar alumno")
    print("2. Mostrar alumnos")
    print("3. Actualizar curso de un alumno")
    print("4. Eliminar alumno")
    print("5. Salir")

    opcion = input("Seleccioná una opción: ")

    if opcion == "1":
        nombre = input("Ingresá el nombre del alumno: ").strip()
        edad = input("Ingresá la edad del alumno: ").strip()
        curso = input("Ingresá el curso del alumno: ").strip()

        if edad.isdigit():  # Validar que la edad sea un número
            agregar_alumno(nombre, int(edad), curso)
        else:
```

```
            print("La edad debe ser un número válido.")

    elif opcion == "2":
        mostrar_alumnos()

    elif opcion == "3":
        id_alumno = input("Ingresá el ID del alumno a actualizar:
").strip()
        nuevo_curso = input("Ingresá el nuevo curso: ").strip()

        if id_alumno.isdigit():
            actualizar_curso(int(id_alumno), nuevo_curso)
        else:
            print("El ID debe ser un número válido.")

    elif opcion == "4":
        id_alumno = input("Ingresá el ID del alumno a eliminar:
").strip()

        if id_alumno.isdigit():
            eliminar_alumno(int(id_alumno))
        else:
            print("El ID debe ser un número válido.")

    elif opcion == "5":
        print("Saliendo del sistema...")
        break

    else:
        print("Opción inválida. Intentá nuevamente.")

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

El código propuesto implementa un **sistema CRUD** para la gestión de alumnos en una base de datos SQLite. Al comenzar, establece una conexión con una base de datos llamada **"instituto.db"** y crea una tabla llamada **"alumnos"** (en caso de que aún no exista). Esta tabla tiene cuatro columnas: **ID**, **nombre**, **edad** y **curso**. La columna ID es un número entero que se incrementa automáticamente para identificar cada alumno de forma única, mientras que las demás columnas almacenan los datos ingresados por el usuario.

Para realizar las operaciones de gestión, se han definido distintas funciones que encapsulan las acciones necesarias para interactuar con la base de datos. La función **agregar_alumno()** permite insertar un nuevo registro en la tabla, asegurándose de recibir el nombre, la edad y el curso del estudiante. La función **mostrar_alumnos()** recupera todos los registros de la base de datos y los muestra en pantalla. Cada fila obtenida a través de la consulta SQL se presenta con el ID, el nombre, la edad y el curso de una persona.

Cuando se necesita modificar el curso de un alumno, la función **actualizar_curso()** permite actualizar la información en la base de datos, solicitando el **ID del alumno** y el **nuevo curso** que se asignará. Del mismo modo, la función **eliminar_alumno()** permite borrar un registro de la base de datos de manera permanente, solicitando el **ID del alumno** que se desea eliminar. Estas operaciones garantizan que los datos se mantengan actualizados y organizados dentro del sistema.

El código principal ejecuta un **menú interactivo** que le permite al usuario seleccionar qué acción desea realizar. Cada opción está asociada a una función específica que ejecuta la operación correspondiente. Si el usuario elige agregar un nuevo alumno, se le solicitan los datos y se verifica que la edad ingresada sea un número válido antes de insertarlo en la base de datos. Si elige mostrar la lista de alumnos, se ejecuta la consulta para recuperar todos los registros existentes y mostrarlos en la pantalla.

Para modificar el curso de un alumno, el sistema solicita el **ID del alumno** y el **nuevo curso** a asignar. Antes de ejecutar la actualización, verifica que el ID ingresado sea numérico. Lo mismo ocurre en la opción de eliminación: el sistema solicita el **ID del alumno**, lo valida y, si es correcto, procede a eliminarlo de la base de datos.

Cuando el usuario decide salir del sistema, la conexión con la base de datos se cierra correctamente. Esto es fundamental para garantizar que los cambios realizados se guarden y evitar problemas de acceso a la base de datos en futuras ejecuciones del programa.

Este código permite gestionar alumnos en una base de datos de manera eficiente, aplicando las operaciones CRUD de una forma estructurada y organizada. Más adelante veremos cómo mejorar la seguridad de estas consultas para prevenir errores y vulnerabilidades, asegurando que la base de datos sea robusta y segura.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

¿Qué es una inyección SQL y por qué es un problema?

Cuando trabajamos con bases de datos en Python o en cualquier otro lenguaje, es fundamental asegurarnos de que nuestras consultas sean seguras. Una de las principales amenazas que enfrentamos al interactuar con bases de datos es la **inyección SQL**. Este tipo de ataque ocurre cuando un usuario malintencionado introduce código SQL en los campos de entrada de un formulario, manipulando la consulta para acceder, modificar o incluso eliminar datos de la base de datos.

Por ejemplo, si en un sistema sin protección, un usuario ingresa el siguiente texto en un campo de búsqueda:

```
' OR 1=1 --
```

Y el sistema ejecuta la consulta sin validar la entrada, se podría generar una instrucción como esta:

```
SELECT * FROM alumnos WHERE nombre = '' OR 1=1 --'
```

La condición **1=1** siempre es verdadera, por lo que la consulta devolverá todos los registros de la tabla. En algunos casos, una inyección SQL maliciosa podría incluso eliminar toda la base de datos o exponer información sensible.

Para evitar este problema, nunca debemos construir las consultas SQL concatenando variables directamente en la consulta. En su lugar, se deben utilizar **parámetros en consultas preparadas**, lo que evita que el contenido ingresado por el usuario se interprete como código SQL.

A continuación, presentamos una versión segura de una consulta de inserción de datos utilizando **parámetros con SQLite**:

```
def agregar_alumno_seguro(nombre, edad, curso):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()
```

```
    cursor.execute("INSERT INTO alumnos (nombre, edad, curso) VALUES (?, ?, ?)", (nombre, edad, curso))

    conexion.commit()
    conexion.close()
```

En este caso, los signos de interrogación (?) actúan como **marcadores de posición** en la consulta, y los valores que proporciona el usuario se pasan como una tupla aparte. De esta manera, **SQLite maneja los datos de manera segura y evita que sean interpretados como código SQL malicioso**.

Otro ejemplo, aplicando este principio en una consulta de selección:

```
def buscar_alumno(nombre):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()

    cursor.execute("SELECT * FROM alumnos WHERE nombre = ?", (nombre,))

    resultados = cursor.fetchall()
    conexion.close()

    return resultados
```

Aquí, en lugar de construir la consulta concatenando el valor de nombre, se utiliza un **parámetro de consulta**, asegurando que el dato ingresado por el usuario no sea tratado como código SQL.

Comparación entre consultas seguras y vulnerables

Para que quede claro cómo mejorar la seguridad de nuestras consultas, comparemos dos versiones de una consulta de eliminación de alumnos:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

❌ **Consulta insegura (vulnerable a inyección SQL):**

```
def eliminar_alumno_inseguro(nombre):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()

    # Construcción de la consulta con formato inseguro
    consulta = f"DELETE FROM alumnos WHERE nombre = '{nombre}'"
    cursor.execute(consulta)

    conexion.commit()
    conexion.close()
```

Este código es peligroso porque si alguien ingresa un nombre como: ' OR 1=1 -- se generaría una consulta que borraría todos los alumnos de la base de datos.

✅ **Consulta segura (protegida contra inyección SQL)**

```
def eliminar_alumno_seguro(nombre):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()

    # Uso de parámetros para prevenir inyección SQL
    cursor.execute("DELETE FROM alumnos WHERE nombre = ?", (nombre,))

    conexion.commit()
    conexion.close()
```

Este código evita el problema porque el valor ingresado se trata como un dato, no como parte de la consulta SQL. Este enfoque basado en **parámetros de consulta** no sólo protege nuestras bases de datos de ataques, sino que también mejora la eficiencia y seguridad del sistema. En la siguiente sección, veremos cómo el uso de transacciones y manejo de errores nos ayuda a garantizar la integridad de la información en la base de datos.

Uso de transacciones y manejo de errores en bases de datos

Hasta ahora, hemos aprendido a realizar operaciones sobre bases de datos utilizando consultas SQL seguras con SQLite. Sin embargo, para garantizar la **integridad de los datos**, es importante entender el concepto de **transacciones** y cómo manejar posibles errores al interactuar con la base de datos.

Una **transacción** es un conjunto de operaciones SQL que deben ejecutarse **de manera completa** o **no ejecutarse en absoluto**. En otras palabras, o se realizan todas las operaciones correctamente o ninguna de ellas se lleva a cabo.

Por ejemplo, imaginemos un sistema de inscripción de alumnos en un curso. Si agregamos un alumno en la base de datos, pero por un error no se registra correctamente su curso asignado, la base de datos quedaría en un estado inconsistente. Para evitar este tipo de problemas, las bases de datos utilizan transacciones que aseguran que todo el proceso se realice de manera correcta o que, en caso de error, se deshaga automáticamente.

Las transacciones en SQLite se controlan con las siguientes instrucciones:

- **BEGIN TRANSACTION**: Esta instrucción indica el inicio de una transacción. A partir de este punto, cualquier operación que se realice en la base de datos quedará en un estado pendiente y no será aplicada de inmediato. Es una forma de decirle a la base de datos que las siguientes operaciones forman parte de un mismo bloque y deben tratarse como una unidad.
- **COMMIT**: Si todas las operaciones dentro de la transacción se ejecutan correctamente, se usa esta instrucción para confirmar los cambios y hacerlos permanentes en la base de datos. Hasta que no se ejecute `COMMIT`, los cambios no son visibles para otros procesos ni quedan guardados de forma definitiva.
- **ROLLBACK**: Si en algún momento ocurre un error dentro de la transacción, esta instrucción permite revertir todos los cambios realizados desde `BEGIN TRANSACTION`, dejando la base de datos en el estado en que se encontraba antes de iniciar la transacción. Esto es especialmente útil para evitar inconsistencias en los datos y garantizar que no se almacene información incorrecta.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejemplo de transacción en SQLite

Supongamos que tenemos una base de datos con una tabla de **alumnos** y queremos registrar a un estudiante asegurándonos de que **se agregue correctamente** a la base de datos antes de confirmar los cambios.

```
import sqlite3

def registrar_alumno(nombre, edad, curso):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()

    try:
        # Iniciar la transacción
        conexion.execute("BEGIN TRANSACTION")

        # Insertar el nuevo alumno en la base de datos
        cursor.execute("INSERT INTO alumnos (nombre, edad, curso) VALUES
(?, ?, ?)", (nombre, edad, curso))

        # Confirmar los cambios
        conexion.commit()
        print("Alumno registrado con éxito.")

    except sqlite3.Error as e:
        # Si ocurre un error, revertimos los cambios
        conexion.rollback()
        print(f"Error al registrar al alumno: {e}")

    finally:
        conexion.close()
```

Primero, se establece la conexión con la base de datos y se obtiene un cursor para ejecutar consultas. Luego, se inicia una **transacción** con **`BEGIN TRANSACTION`**, lo que indica que cualquier operación SQL posterior será parte de la transacción.

Dentro del bloque **try**, se ejecuta una consulta **INSERT INTO** para agregar el alumno a la base de datos. Si la operación se realiza con éxito, el programa ejecuta **commit()**, lo que confirma los cambios. Sin embargo, si ocurre un error en cualquier punto del proceso (por ejemplo, si la tabla **alumnos** no existe o los datos ingresados son incorrectos), el programa captura la excepción en el bloque **except**. En este caso, se ejecuta **rollback()**, lo que anula cualquier modificación hecha durante la transacción y deja la base de datos en su estado original.

El bloque **finally** es una parte opcional de la estructura **try-except** que se ejecuta siempre, sin importar si ocurrió un error o no dentro del bloque **try**. Esto lo hace ideal para tareas que deben realizarse sin excepción, como cerrar una conexión con la base de datos.

Cuando trabajamos con bases de datos, es muy importante cerrar la conexión después de realizar una operación. Si no lo hacemos, podríamos dejar la base de datos en un estado inconsistente o consumir recursos innecesariamente. El bloque **finally** garantiza que la conexión se cierre incluso si ocurre un error en el proceso.

Por ejemplo, si intentamos insertar datos en la base y ocurre un problema con los valores ingresados, el bloque **except** manejará el error, pero el **finally** se ejecutará de todas formas para cerrar la conexión, asegurando que no queden conexiones abiertas.

Ejemplo de transacción con múltiples operaciones

Si queremos registrar a un alumno y asignarle un código de inscripción en una tabla aparte, podemos realizar ambas operaciones dentro de una misma transacción para asegurarnos de que se ejecuten **en conjunto**.

```
import sqlite3

def inscribir_alumno(nombre, edad, curso, codigo_inscripcion):
    conexion = sqlite3.connect("instituto.db")
    cursor = conexion.cursor()
```

```
    try:
        # Iniciar transacción
        conexion.execute("BEGIN TRANSACTION")

        # Insertar al alumno en la tabla principal
        cursor.execute("INSERT INTO alumnos (nombre, edad, curso) VALUES
(?, ?, ?)", (nombre, edad, curso))

        # Insertar el código de inscripción en otra tabla relacionada
        cursor.execute("INSERT INTO inscripciones (nombre, codigo)
VALUES (?, ?)", (nombre, codigo_inscripcion))

        # Confirmar los cambios
        conexion.commit()
        print("Alumno inscripto con éxito.")

    except sqlite3.Error as e:
        # Si hay un error, revertimos ambas operaciones
        conexion.rollback()
        print(f"Error al inscribir al alumno: {e}")

    finally:
        conexion.close()
```

Este ejemplo introduce un concepto muy importante en la manipulación de bases de datos: el uso de **transacciones en múltiples tablas**. A diferencia del código anterior, aquí no solo insertamos datos en una tabla, sino que realizamos operaciones en dos tablas relacionadas dentro de la misma transacción. Esto significa que ambas inserciones (en la tabla **alumnos** y en la tabla **inscripciones**) se consideran parte de una única operación.

El bloque **try** comienza con **BEGIN TRANSACTION**, lo que indica que cualquier cambio realizado en la base de datos después de esa línea quedará pendiente hasta que se confirme con **commit()**. Si ambas inserciones (**INSERT INTO alumnos** y **INSERT INTO inscripciones**) se ejecutan correctamente, la transacción se confirma y los datos quedan almacenados en ambas tablas.

Sin embargo, si ocurre un error en cualquiera de las dos operaciones, el bloque **except** entra en acción y ejecuta **rollback()**. Esto revierte todas las modificaciones realizadas desde el inicio de la transacción, asegurando que la base de datos no quede en un estado inconsistente.

El bloque **finally** se mantiene igual que en el ejemplo anterior, asegurando que la conexión con la base de datos se cierre siempre, sin importar si la operación fue exitosa o no.

Este enfoque es casi indispensable cuando trabajamos con datos que dependen unos de otros. Si solo se insertase al alumno pero fallara la inserción del código de inscripción, tendríamos información incompleta en la base de datos. Gracias al uso de transacciones, evitamos este problema garantizando que ambas inserciones ocurran juntas o ninguna de ellas se registre.

Ejercicio Práctico

La jornada en **TalentoLab** transcurre con el habitual ritmo de trabajo cuando Mariana se acerca a tu escritorio con su tablet en la mano.

"*¡Felicitaciones! El cliente ha quedado muy conforme con la primera versión del registro de productos. Pudieron cargar información y recuperarla sin problemas. Pero ahora nos piden algunas mejoras para garantizar que el sistema sea más seguro y fácil de administrar.*"

Te muestra en pantalla un par de puntos que debemos agregar:

1. **Validación de datos al registrar un producto:** Actualmente, el sistema permite agregar productos, pero no verifica si los datos ingresados son válidos. Es necesario asegurarse de que el nombre del producto no esté vacío y que el precio sea un número positivo antes de almacenarlo en la base de datos.
2. **Eliminar un producto con confirmación:** El cliente quiere evitar eliminaciones accidentales. Por lo tanto, al eliminar un producto, el sistema debe solicitar una confirmación antes de ejecutar la acción.

Desde su escritorio, Luis se suma a la conversación:

*Podrías aprovechar **Colorama** para mejorar la experiencia visual. Por ejemplo, podrías resaltar los mensajes de advertencia en **rojo**, las confirmaciones en **verde**, y la información general en **azul**. Haría que la interfaz sea más clara y atractiva para el usuario.*

Materiales y Recursos Adicionales:

Artículos:
Recursos Python: Inyección SQL en sqlite3
ITtrip: Uso y la importancia de las consultas parametrizadas en SQL

Videos:
DayiTecnologia: Creando un CRUD con SQLite3 y Python
LuisDev: Manejo de errores en Python

Preguntas para Reflexionar:

1. Cuando trabajamos con bases de datos en Python, ¿por qué es importante utilizar parámetros en las consultas en lugar de concatenar directamente los valores ingresados por el usuario?
2. En la implementación de transacciones con SQLite, ¿qué problema se podría generar si olvidamos hacer un COMMIT después de una serie de modificaciones en la base de datos?
3. El manejo de excepciones con try-except es clave para evitar errores inesperados. ¿Qué tipo de situaciones creés que podrían requerir un ROLLBACK dentro de un bloque except?
4. Si tuvieras que optimizar un sistema CRUD para mejorar su seguridad y rendimiento, ¿qué técnicas o herramientas considerarías implementar más allá de lo aprendido en esta clase?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos Pasos:

A medida que avanzamos en nuestro camino en **Python y bases de datos**, es momento de integrar todos los conocimientos adquiridos. En la próxima clase, vamos a llevar a cabo el **Proyecto Final**, un desafío en el que pondrás en práctica cada uno de los conceptos trabajados a lo largo del curso.

Será una oportunidad para **desarrollar un sistema completo**, aplicando desde estructuras básicas hasta bases de datos y manejo de errores. También veremos **buenas prácticas de programación** y cómo documentar tu código de manera profesional para que sea claro y comprensible para cualquier persona que lo lea.

Durante este proceso, no solo consolidarás lo aprendido, sino que también te prepararás para presentar tu trabajo de manera ordenada y efectiva, tal como lo harías en un entorno profesional. **¡Este es el momento de demostrar todo lo que has aprendido!**

Para aprovechar al máximo la próxima clase, repasá el código de tus proyectos anteriores, asegurate de comprender cómo funcionan las operaciones CRUD con SQLite y pensá en cómo podrías mejorar y optimizar tu código con lo que aprendiste hasta ahora.

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad