Etapa 3

Rumbo al Proyecto final Full Stack Python con objetos y API REST

Tercera etapa:

Utilizar como almacenamiento una base de datos SQL.

Vamos a modificar las clases y objetos desarrollados antes para almacenar los datos en una base de datos SQL. Para realizar esta tarea, necesitaremos realizar los siguientes pasos:

- Crear una base de datos SQL y las tablas correspondientes para almacenar los productos y el contenido del carrito.
- Importar el módulo **sqlite3** en el código.
- Crear una conexión a la base de datos.
- Crear tablas en la base de datos para almacenar los productos y el contenido del carrito.
- Modificar los métodos relevantes en las clases existentes para interactuar con la base de datos en lugar de las listas actuales.
- Actualizar los métodos de agregar y quitar productos en la clase Carrito para reflejar los cambios en la cantidad de productos en el inventario.
- Modificar los métodos de las clases Inventario y Carrito para que interactúen con la base de datos en lugar de utilizar listas en memoria.
- Actualizar los métodos relacionados con la modificación y eliminación de productos en el carrito para reflejar los cambios en la base de datos.

La base de datos

El código que veremos utiliza una base de datos SQLite llamada '**inventario.db**' y crea una tabla llamada '**productos**' en esa base de datos. La estructura de la tabla 'productos' se define con las siguientes columnas:

codigo: Es una columna de tipo INTEGER y se define como la clave primaria (PRIMARY KEY) de la tabla. Esta columna almacena el código único para identificar cada producto.

descripcion: Es una columna de tipo TEXT y almacena la descripción del producto.

cantidad: Es una columna de tipo INTEGER y almacena la cantidad disponible del producto.

precio: Es una columna de tipo FLOAT y almacena el precio del producto.

En resumen, la tabla 'productos' tiene cuatro columnas que representan el código, la descripción, la cantidad y el precio de cada producto.

La base de datos SQLite, en este caso representada por el archivo 'inventario.db', almacenará los datos de los productos utilizados en el proyecto carrito de compras. Cada vez que se agrega, modifica o elimina un producto, se realizarán las operaciones correspondientes en la tabla 'productos' de la base de datos.

¿Qué es y como funciona SQLite?

El módulo sqlite de Python es una biblioteca integrada que proporciona una interfaz para trabajar con bases de datos SQLite. SQLite es un sistema de gestión de bases de datos relacional ligero y autónomo que se implementa como una biblioteca en C. Las siguientes son algunas características y conceptos clave relacionados con el uso de SQLite y el módulo sqlite de Python:

Base de datos SQLite: SQLite es una base de datos relacional que no requiere un servidor de base de datos separado. Se almacena en un archivo local en el sistema de archivos en lugar de ejecutarse en un proceso de servidor. Esto facilita su integración y distribución, ya que toda la base de datos está contenida en un solo archivo.

Características de SQLite:

Ligero: SQLite está diseñado para ser liviano y tiene una huella de memoria mínima.
Sin servidor: No se requiere un proceso de servidor dedicado para utilizar SQLite.
Transaccional: SQLite es compatible con transacciones ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad).
Soporte de SQL: SQLite admite una amplia variedad de comandos SQL estándar, como SELECT, INSERT, UPDATE, DELETE, etc.
Sin configuración: No se requiere una configuración complicada para comenzar a usar SQLite.
Portátil: Las bases de datos SQLite se almacenan en un solo archivo, lo que facilita su portabilidad entre sistemas y plataformas.

Almacenamiento de bases de datos SQLite: Las bases de datos SQLite se almacenan en un archivo local en el sistema de archivos del sistema operativo. Puedes especificar la ubicación y el nombre del archivo al conectarte a la base de datos.

Módulo sqlite de Python: El módulo sqlite de Python proporciona una interfaz para interactuar con bases de datos SQLite desde programas escritos en Python. Permite ejecutar comandos SQL, realizar consultas, insertar, modificar y eliminar datos, y administrar transacciones. El módulo sqlite proporciona una serie de funciones y clases que facilitan la interacción con las bases de datos SQLite.

Al utilizar el módulo sqlite de Python, puedes aprovechar todas las características de SQLite y administrar fácilmente bases de datos locales en tus aplicaciones.

Instalar el módulo sqlite3

Para comenzar, necesitaremos instalar el módulo sqlite3 en Python. Puedes instalarlo ejecutando el siguiente comando en tu entorno virtual o terminal:

```
pip install pysqlite3
```

Recuerda que es recomendable utilizar un entorno virtual para tus proyectos de Python. Puedes crear un entorno virtual utilizando herramientas como **venv** o **virtualenv**, y luego instalar Flask dentro del entorno virtual. Esto ayuda a mantener las dependencias de cada proyecto separadas y evita conflictos entre versiones.

Y escribimos el código necesario para importar los módulos necesarios y establecer la conexión con la base de datos. Se utiliza SQLite para crear una aplicación con una base de datos SQLite.

```
import sqlite3

# Configurar la conexión a la base de datos SQLite
DATABASE = 'inventario.db'

def get_db_connection():
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla 'productos' si no existe
def create_table():
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS productos (
            codigo INTEGER PRIMARY KEY,
            descripcion TEXT NOT NULL,
            cantidad INTEGER NOT NULL,
            precio REAL NOT NULL
        )
    ''')
    conn.commit()
    cursor.close()
    conn.close()
```

```
# Verificar si la base de datos existe, si no, crearla y crear la tabla
def create_database():
    conn = sqlite3.connect(DATABASE)
    conn.close()
    create_table()

# Crear la base de datos y la tabla si no existen
create_database()


# ----------------------------------------------------------------
# Definimos la clase "Inventario"
# ----------------------------------------------------------------
class Inventario:
    def __init__(self):
        self.conexion = get_db_connection()
        self.cursor = self.conexion.cursor()
```

En la primera línea, importamos la clase Flask del módulo flask para crear una instancia de la aplicación Flask.

Luego, se establece el nombre de la base de datos SQLite en la variable DATABASE. En este caso, se utiliza el archivo '**inventario.db**' como base de datos.

La función **get_db_connection()** se define para establecer la conexión con la base de datos SQLite utilizando la biblioteca sqlite3. Esta función devuelve una conexión a la base de datos que se puede utilizar para realizar consultas y transacciones.

La función **create_table()** se define para crear la tabla 'productos' en la base de datos SQLite si no existe. Utiliza la conexión obtenida mediante **get_db_connection()** y el método **execute()** para ejecutar una sentencia SQL que crea la tabla con las columnas 'codigo', 'descripcion', 'cantidad' y 'precio'. Luego, se realiza una confirmación (**commit()**) para guardar los cambios y se cierra el cursor y la conexión.

Finalmente, se llama a **create_table()** para asegurarse de que la tabla 'productos' exista en la base de datos antes de continuar con el resto del código.

En resumen, se establece la conexión a la base de datos SQLite y crea la tabla 'productos' si no existe. Esto asegura que la base de datos esté lista para almacenar los datos de los productos antes de que la aplicación Flask comience a interactuar con ella.

Recuerda reemplazar los nombres del servidor, usuario, contraseña y base de datos con los valores correspondientes para tu configuración de MySQL.

Ahora estamos listos para analizar el código de cada clase.

Clase Producto

La clase Producto no sufre cambios, ya que sus instancias solo se utilizan como parte del inventario:

```
# ----------------------------------------------------------------
# Definimos la clase "Producto"
# ----------------------------------------------------------------
class Producto:
    def __init__(self, codigo, descripcion, cantidad, precio):
        self.codigo = codigo
        self.descripcion = descripcion
        self.cantidad = cantidad
        self.precio = precio

    def modificar(self, nueva_descripcion, nueva_cantidad, nuevo_precio):
        self.descripcion = nueva_descripcion
        self.cantidad = nueva_cantidad
        self.precio = nuevo_precio
```

Clase Inventario

Método init:

La clase Inventario representa un inventario de productos y tiene un constructor init que se llama cuando se crea una instancia de la clase. Al crear una instancia de la clase Inventario, se establece una conexión a la base de datos y se crea un cursor que se puede utilizar para interactuar con la base de datos. Esto permite que la clase Inventario realice operaciones de base de datos, como agregar, modificar, consultar y eliminar productos.

```
def __init__(self):
    self.conexion = get_db_connection()
    self.cursor = self.conexion.cursor()
```

En el constructor init, se realiza lo siguiente:

self.conexion = get_db_connection(): Se establece la conexión a la base de datos utilizando la función **get_db_connection()**. Esta función devuelve una conexión a la base de datos SQLite.

self.cursor = self.conexion.cursor(): Se crea un objeto cursor a partir de la conexión a la base de datos. Un cursor es utilizado para ejecutar consultas y obtener resultados de la base de datos.

Método agregar_producto:

El método agregar_producto crea una instancia de la clase Producto con los datos proporcionados y luego inserta esos datos en la base de datos utilizando una consulta SQL. Después de la inserción, se realiza una confirmación para guardar los cambios en la base de datos.

```
def agregar_producto(self, codigo, descripcion, cantidad, precio):
    producto_existente = self.consultar_producto(codigo)
    if producto_existente:
        print("Ya existe un producto con ese código.")
        return False

    nuevo_producto = Producto(codigo, descripcion, cantidad, precio)
    self.cursor.execute("INSERT INTO productos VALUES (?, ?, ?, ?)", (codigo, descripcion, cantidad, precio))
    self.conexion.commit()
    return True
```

El método agregar_producto es parte de la clase Inventario. Su función es agregar un nuevo producto a la base de datos y también crear una instancia de la clase Producto con los datos proporcionados. Aquí está la descripción del método agregar_producto paso a paso:

Para evitar agregar un producto con un código que ya existe en la base de datos, se realiza una verificación antes de realizar la inserción. Si se encuentra un producto existente con el mismo código, se imprime un mensaje de error y se devuelve False. En caso contrario, se procede a agregar el nuevo producto a la base de datos y se devuelve True para indicar que la operación se realizó correctamente. *Esto se puede evitar utilizando en la tabla de la base de datos un campo ID que se genere de forma automática.*

nuevo_producto = Producto(codigo, descripcion, cantidad, precio): Crea una nueva instancia de la clase Producto con los parámetros de entrada proporcionados (codigo, descripcion, cantidad, precio). Esta línea crea un objeto nuevo_producto que representa el producto que se va a agregar.

self.cursor.execute("INSERT INTO productos VALUES (?, ?, ?, ?)", (codigo, descripcion, cantidad, precio)): Ejecuta una consulta SQL utilizando el cursor para insertar una nueva fila en la tabla "productos". La consulta utiliza parámetros de marcador de posición (?) para evitar posibles problemas de seguridad al construir la consulta SQL. Los valores de codigo, descripcion, cantidad y precio se pasan como una tupla en el segundo argumento de la función execute.

self.conexion.commit(): Realiza una confirmación explícita de la transacción en la base de datos. Esto asegura que los cambios realizados en la tabla (en este caso, la inserción del nuevo producto) se guarden de forma permanente en la base de datos.

Método consultar_producto:

Su objetivo es buscar un producto en la base de datos según el código proporcionado y devolver una instancia de la clase Producto si se encuentra, o False si no se encuentra.
Ejecuta una consulta SQL para buscar un producto en la base de datos según el código proporcionado. Si se encuentra, se crea una instancia de la clase Producto con los datos recuperados y se devuelve. De lo contrario, se devuelve False.

```
def consultar_producto(self, codigo):
    self.cursor.execute("SELECT * FROM productos WHERE codigo = ?", (codigo,))
    row = self.cursor.fetchone()
    if row:
        codigo, descripcion, cantidad, precio = row
        return Producto(codigo, descripcion, cantidad, precio)
    return False
```

Descripción del método consultar_producto paso a paso:

self.cursor.execute("SELECT * FROM productos WHERE codigo = ?", (codigo,)): Ejecuta una consulta SQL utilizando el cursor para seleccionar todas las columnas de la tabla "productos" donde el código coincide con el valor proporcionado. La consulta utiliza un parámetro de marcador de posición (?) para evitar problemas de seguridad y el valor del código se pasa como una tupla en el segundo argumento de la función execute.

row = self.cursor.fetchone(): Recupera la primera fila de resultados de la consulta realizada. El método **fetchone()** devuelve una tupla que contiene los valores de las columnas seleccionadas.

Si row es verdadero (es decir, se encontró una fila que coincide con el código), se extraen los valores de codigo, descripcion, cantidad y precio de la tupla row, y luegose crea una instancia de la clase Producto utilizando los valores extraídos y se devuelve como resultado de la función .

Si no se encontró ninguna fila que coincida con el código, se devuelve False.

Método modificar_producto:

El método modificar_producto también forma parte de la clase Inventario. Su propósito es modificar los datos de un producto existente en la base de datos, así como en la instancia correspondiente de la clase Producto

```
def modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio):
    producto = self.consultar_producto(codigo)
    if producto:
        producto.modificar(nueva_descripcion, nueva_cantidad, nuevo_precio)
        self.cursor.execute("UPDATE productos SET descripcion = ?, cantidad = ?, precio = ? WHERE codigo = ?",
                            (nueva_descripcion, nueva_cantidad, nuevo_precio, codigo))
        self.conexion.commit()
```

Veamos la explicación paso a paso del método modificar_producto:

producto = self.consultar_producto(codigo): Se llama al método **consultar_producto** para obtener una instancia existente de la clase Producto según el código proporcionado. Si se encuentra el producto, se asigna a la variable producto; de lo contrario, producto será None.

Se verifica si producto es verdadero, lo que significa que se encontró un producto con el código proporcionado en la base de datos. Si es así, se procede a realizar la modificación.

producto.modificar(nueva_descripcion, nueva_cantidad, nuevo_precio): Se llama al método modificar del objeto producto para actualizar sus atributos descripcion, cantidad y precio con los nuevos valores proporcionados.

self.cursor.execute("UPDATE productos SET descripcion = ?, cantidad = ?, precio = ? WHERE codigo = ?", (nueva_descripcion, nueva_cantidad, nuevo_precio, codigo)): Se ejecuta una consulta SQL utilizando el cursor para actualizar los datos del producto en la tabla "productos". La consulta utiliza parámetros de marcador de posición (?) para evitar problemas de seguridad y los nuevos valores se pasan como una tupla en el segundo argumento de la función execute.

self.conexion.commit(): Se realiza una confirmación explícita de la transacción en la base de datos para guardar los cambios de la actualización.

En resumen, el método modificar_producto busca un producto en la base de datos mediante el código proporcionado. Si se encuentra, se actualiza la instancia del objeto Producto y se ejecuta una consulta SQL para actualizar los datos correspondientes en la base de datos.

Método listar_productos:

El método listar_productos recupera todos los productos de la base de datos y muestra su información en la consola en un formato legible.

```
def listar_productos(self):
        print("-" * 30)
        self.cursor.execute("SELECT * FROM productos")
        rows = self.cursor.fetchall()
        for row in rows:
            codigo, descripcion, cantidad, precio = row
            print(f"Código: {codigo}")
            print(f"Descripción: {descripcion}")
            print(f"Cantidad: {cantidad}")
            print(f"Precio: {precio}")
            print("-" * 30)
```

print("-" * 30): Imprime una línea horizontal de guiones para separar visualmente la lista de productos.

self.cursor.execute("SELECT * FROM productos"): Ejecuta una consulta SQL utilizando el cursor para seleccionar todos los productos de la tabla "productos".

rows = self.cursor.fetchall(): Recupera todas las filas de resultados de la consulta en una lista llamada rows. Cada fila es una tupla que contiene los valores de las columnas seleccionadas.

Luego se itera sobre cada fila en rows utilizando un bucle **for row in rows**. Dentro del bucle, se desempaqueta cada fila en las variables codigo, descripcion, cantidad y precio.

Se imprime en la consola la información del producto utilizando el formato especificado, incluyendo el código, la descripción, la cantidad y el precio.

Para finalizar, se imprime otra línea horizontal de guiones después de imprimir la información de cada producto para separar visualmente los productos en la lista.

Método eliminar_producto:

Ejecuta una consulta SQL para eliminar un producto de la base de datos según el código proporcionado. Si se encuentra y se elimina al menos una fila, se muestra un mensaje de confirmación y se realiza la confirmación en la base de datos. De lo contrario, se muestra un mensaje indicando que el producto no fue encontrado.

```
def eliminar_producto(self, codigo):
    self.cursor.execute("DELETE FROM productos WHERE codigo = ?", (codigo,))
    if self.cursor.rowcount > 0:
        print("Producto eliminado.")
        self.conexion.commit()
    else:
        print("Producto no encontrado.")
```

Aquí está la descripción del método eliminar_producto paso a paso:

self.cursor.execute("DELETE FROM productos WHERE codigo = ?", (codigo,)): Ejecuta una consulta SQL utilizando el cursor para eliminar el producto de la tabla "productos" donde el código coincide con el valor proporcionado. La consulta utiliza un parámetro de marcador de posición (?) para evitar problemas de seguridad y el valor del código se pasa como una tupla en el segundo argumento de la función execute.

if self.cursor.rowcount > 0:: Se verifica si el número de filas afectadas por la operación de eliminación, obtenido a través del atributo rowcount del cursor, es mayor que cero. Esto indica que se eliminó al menos una fila de la base de datos.

Si se eliminó al menos una fila, se imprime "Producto eliminado" en la consola y se realiza una confirmación explícita de la transacción en la base de datos mediante **self.conexion.commit()** para guardar los cambios de la eliminación. Si no se encontró ninguna fila para eliminar, se imprime "Producto no encontrado" en la consola.

Clase Carrito

Método init:

El método init es el constructor de la clase Carrito. Se llama automáticamente cuando se crea una nueva instancia de la clase Carrito. Su objetivo es inicializar los atributos de la clase.
El método init establece la conexión a la base de datos SQLite, crea un cursor y inicializa el atributo items como una lista vacía.

```
def __init__(self):
    self.conexion = sqlite3.connect('inventario.db')  # Conexión a la base de datos
    self.cursor = self.conexion.cursor()
    self.items = []
```

self.conexion = sqlite3.connect('inventario.db'): Crea una conexión a la base de datos SQLite llamada "inventario.db". Se utiliza el módulo sqlite3 de Python para establecer la conexión.

self.cursor = self.conexion.cursor(): Crea un objeto de cursor a través de la conexión. El cursor se utiliza para ejecutar consultas y realizar operaciones en la base de datos.

self.items = []: Inicializa el atributo items como una lista vacía. Este atributo se utiliza para almacenar los productos en el carrito de compras.

Método agregar:

Verifica la existencia y disponibilidad del producto en el inventario. Si el producto existe y hay suficiente cantidad disponible, se agrega al carrito y se actualizan tanto self.items como la cantidad en la base de datos. Si el producto no existe o no hay suficiente cantidad disponible, se imprime un mensaje y se devuelve False.

```
def agregar(self, codigo, cantidad, inventario):
        producto = inventario.consultar_producto(codigo)
        if producto is False:
            print("El producto no existe.")
            return False
        if producto.cantidad < cantidad:
            print("Cantidad en stock insuficiente.")
            return False

        for item in self.items:
            if item.codigo == codigo:
                item.cantidad += cantidad
                self.cursor.execute("UPDATE productos SET cantidad = cantidad - ? WHERE codigo = ?",
                                    (cantidad, codigo))
                self.conexion.commit()
                return True

        nuevo_item = Producto(codigo, producto.descripcion, cantidad, producto.precio)
        self.items.append(nuevo_item)
        self.cursor.execute("UPDATE productos SET cantidad = cantidad - ? WHERE codigo = ?",
                            (cantidad, codigo))
        self.conexion.commit()
        return True
```

producto = inventario.consultar_producto(codigo): Utiliza el objeto inventario para llamar al método **consultar_producto(codigo)** y obtener el objeto Producto correspondiente al código proporcionado. Si el producto no existe, se imprime "El producto no existe" y se devuelve False.

if producto.cantidad < cantidad: Comprueba si la cantidad solicitada del producto es mayor que la cantidad disponible en el inventario. Si es así, se imprime "Cantidad en stock insuficiente" y se devuelve False.

Se itera sobre los elementos (item) en self.items, que representa los productos en el carrito actual. Si se encuentra un item con el mismo código que el producto a agregar, se incrementa la cantidad del item por la cantidad especificada. Luego, se ejecuta una consulta SQL utilizando **self.cursor.execute** para actualizar la cantidad del producto en la base de datos. La cantidad se reduce en la cantidad especificada. Finalmente, se realiza la confirmación (**self.conexion.commit()**) para guardar los cambios en la base de datos y se devuelve True.

Si no se encontró un item con el mismo código, se crea un nuevo objeto Producto llamado nuevo_item con los detalles del producto a agregar. Luego, se agrega nuevo_item a self.items, que representa los productos en el carrito actual. Después, se ejecuta una consulta SQL para actualizar la cantidad del producto en la base de datos, reduciendo la cantidad en la cantidad especificada. Finalmente, se realiza la confirmación en la base de datos y se devuelve True.

Método quitar:

Se utiliza para quitar una cantidad específica de un producto del carrito de compras. Busca el producto en el carrito y, si se encuentra, verifica si la cantidad a quitar es válida. Si es válida, se actualiza la cantidad del producto en el carrito y en la base de datos. Si el producto no se encuentra en el carrito, se imprime un mensaje y se devuelve False.

```
def quitar(self, codigo, cantidad, inventario):
    for item in self.items:
        if item.codigo == codigo:
            if cantidad > item.cantidad:
                print("Cantidad a quitar mayor a la cantidad en el carrito.")
                return False
            item.cantidad -= cantidad
            if item.cantidad == 0:
                self.items.remove(item)
            self.cursor.execute("UPDATE productos SET cantidad = cantidad + ? WHERE codigo = ?",
                                (cantidad, codigo))
            self.conexion.commit()
            return True

    print("El producto no se encuentra en el carrito.")
    return False
```

Veamos el paso a paso de este método:

Se itera sobre los elementos (item) en self.items, que representa los productos en el carrito actual.

a. Si se encuentra un item con el mismo código que el producto a quitar, se realizan las siguientes comprobaciones:

i. Se verifica si la cantidad especificada es mayor que la cantidad actual del item. Si es así, se imprime "Cantidad a quitar mayor a la cantidad en el carrito" y se devuelve False.

ii. Se reduce la cantidad del item por la cantidad especificada.
iii. Si la cantidad del item llega a cero, se elimina el item de self.items.
iv. Se ejecuta una consulta SQL utilizando **self.cursor.execute** para actualizar la cantidad del producto en la base de datos. La cantidad se incrementa en la cantidad especificada.

v. Se realiza la confirmación (**self.conexion.commit()**) para guardar los cambios en la base de datos y se devuelve True.

Si no se encuentra un item con el mismo código, se imprime "El producto no se encuentra en el carrito" y se devuelve False.

Método mostrar:

El método mostrar muestra los detalles de cada producto en el carrito, incluyendo su código, descripción, cantidad y precio. Cada producto se imprime en una sección separada visualmente mediante líneas de guiones.

```
def mostrar(self):
        print("-" * 30)
        for item in self.items:
            print(f"Código: {item.codigo}")
            print(f"Descripción: {item.descripcion}")
            print(f"Cantidad: {item.cantidad}")
            print(f"Precio: {item.precio}")
            print("-" * 30)
```

Se imprime una línea de guiones ("-") repetida 30 veces para crear una separación visual en la salida.

Se itera sobre los elementos (item) en self.items, que representa los productos en el carrito actual.

a. Para cada item, se imprime su código, descripción, cantidad y precio utilizando la función print.
b. Después de imprimir los detalles de un item, se imprime otra línea de guiones ("-") repetida 30 veces para separar visualmente los detalles de los productos.

Ejemplo de uso de las clases y objetos definidos

Este código muestra un ejemplo de uso de las clases y objetos definidos anteriormente. Ilustra cómo utilizar las clases y objetos definidos para agregar, quitar y mostrar productos en un inventario y un carrito de compras:

```
# ------------------------------------------------------------------
# Ejemplo de uso de las clases y objetos definidos antes:
# ------------------------------------------------------------------
# Crear una instancia de la clase Inventario
x = Inventario()

# Crear una instancia de la clase Carrito
mi_carrito = Carrito()

# Agregar productos al inventario
x.agregar_producto(1, "Producto 1", 10, 19.99)
x.agregar_producto(2, "Producto 2", 5, 9.99)
x.agregar_producto(3, "Producto 3", 15, 29.99)

# Agregar productos al carrito
mi_carrito.agregar(1, 2, x)  # Agregar 2 unidades del producto con código 1 al carrito
mi_carrito.agregar(3, 1, x)  # Agregar 1 unidad del producto con código 3 al carrito
mi_carrito.quitar(1, 1, x)   # Quitar 1 unidad del producto con código 1 al carrito
mi_carrito.agregar(2, 1, x)  # Agregar 1 unidad del producto con código 2 al carrito

mi_carrito.mostrar()
```

Descripción paso a paso del código:

- Se crea una instancia de la clase Inventario llamada x.
- Se crea una instancia de la clase Carrito llamada mi_carrito
- Se agregan productos al inventario utilizando el método agregar_producto de la instancia de Inventario (x). Se agregan tres productos con diferentes códigos, descripciones, cantidades y precios.
- Se agregan productos al carrito utilizando el método agregar de la instancia de Carrito (mi_carrito). Se agregan dos unidades del producto con código 1 y una unidad del producto con código 3 al carrito.
- Se quita una unidad del producto con código 1 del carrito utilizando el método quitar de la instancia de Carrito (mi_carrito).
- Se agrega una unidad del producto con código 2 al carrito utilizando el método agregar de la instancia de Carrito (mi_carrito).
- Se muestra el contenido del carrito utilizando el método mostrar de la instancia de Carrito (mi_carrito).

La salida espera es la siguiente:

```
------------------------------
Código: 1
Descripción: Producto 1
Cantidad: 1
Precio: 19.99
------------------------------
Código: 3
Descripción: Producto 3
Cantidad: 1
Precio: 29.99
------------------------------
Código: 2
Descripción: Producto 2
Cantidad: 1
Precio: 9.99
------------------------------
```

El paso siguiente consiste en agregar a nuestro proyecto el código necesario para implementar una API.