

```
0 response = requests.get(url)
1
2 # checking response.status_code (if you get 502, try rerunning the code)
3 if response.status_code != 200:
4     print(f"Status: {response.status_code} - Try rerunning the code!")
5 else:
6     print(f"Status: {response.status_code}\n")
7
8 # using BeautifulSoup to parse the response object
9 soup = BeautifulSoup(response.content, "html.parser")
10
11 # finding Post images in the soup
12 images = soup.find_all("img", attrs={"alt": "Post image"})
13
14 # downloading images
15 for i in range(len(images)):
16     # image url
17     url = images[i].get("src")
18     # image name
19     name = f"image_{i}.jpg"
20     # downloading image
21     response = requests.get(url)
22     with open(name, "wb") as f:
23         f.write(response.content)
```

# PROYECTO CRUD

## TPO FINAL

HTML - CSS - JAVASCRIPT - VUE  
BASES DE DATOS - PYTHON - FLASK

- Desarrollo de clases y objetos
- Creación de la base de datos
- Implementación de la API en Flask
- Despliegue en servidor PythonAnywhere
- Codificación del Front-End

Codo a Codo  
2023

# INDICE

PRESENTACIÓN DEL PROYECTO .....	3
ETAPA 1: DESARROLLO DE CLASES Y OBJETOS.....	3
Clase PRODUCTO .....	3
Clase INVENTARIO .....	4
Clase CARRITO .....	8
ETAPA 2: CREACIÓN DE LA BASE DE DATOS.....	11
Utilizar como almacenamiento una base de datos SQL.....	11
La base de datos.....	12
¿Qué es y cómo funciona SQLite? .....	12
Instalar el módulo sqlite3.....	13
Modificando la clase Producto.....	14
Modificando la clase Inventario .....	15
Método init.....	15
Método agregar_producto.....	15
Método consultar_producto .....	16
Método modificar_producto.....	16
Método listar_productos .....	18
Método eliminar_producto.....	17
Modificando la clase Carrito .....	18
Método init.....	18
Método agregar .....	19
Método quitar .....	20
Método mostrar.....	20
Ejemplo de uso de las clases y objetos definidos .....	21
ETAPA 3: IMPLEMENTACIÓN DE LA API EN FLASK .....	24
Instalar el módulo Flask .....	24
Código de la aplicación.....	25
Configuración y rutas de la API Flask .....	26
1) Importación de los módulos y creación de la aplicación Flask: .....	26
2) Decorador @app.route('/productos/<int:codigo>', methods=['GET']) y función obtener_producto(codigo):.....	26
3) Decorador @app.route('/') y función index(): .....	27
4) Decorador @app.route('/productos', methods=['GET']) y función obtener_productos():.....	28

5) Decorador @app.route('/productos', methods=['POST']) y función agregar_producto(): .....	28
6) Decorador @app.route('/productos/<int:codigo>', methods=['PUT']) y función modificar_producto(): .....	28
7) Decorador @app.route('/productos/<int:codigo>', methods=['DELETE']) y función eliminar_producto(): .....	29
8) Decorador @app.route('/carrito', methods=['POST']) y función agregar_carrito():...	29
9) Decorador @app.route('/carrito', methods=['DELETE']) y función quitar_carrito():...	30
10) Decorador @app.route('/carrito', methods=['GET']) y función obtener_carrito():	30
Clase Producto .....	31
Clase inventario .....	31
Método init.....	31
Método agregar_producto.....	32
Método consultar_producto .....	33
Método modificar_producto.....	34
Método listar_productos .....	35
Método eliminar_producto.....	35
Clase Carrito .....	36
Método init.....	36
Método agregar .....	36
Método quitar .....	37
Método mostrar.....	38
Ejecución del código.....	38
ETAPA 4: DESPLIEGUE EN SERVIDOR PYTHONANYWHERE .....	39
Registro y configuración.....	40
Prueba rápida de la API .....	43
CORS .....	45
ETAPA 5: CODIFICACIÓN DEL FRONT-END .....	45
Menú principal .....	46
Alta de productos.....	46
Hoja de estilos .....	50

**PROYECTO CRUD - TPO FINAL**  
*HTML - CSS - JAVASCRIPT - VUE*  
*BASES DE DATOS - PYTHON - FLASK*

## PRESENTACIÓN DEL PROYECTO

Este proyecto consiste en realizar un CRUD (Create, Read, Update, Delete) de un inventario de productos. Además incorpora un carrito de compras desde el cual también se pueden agregar o quitar productos.

Como ejemplo trabajaremos con una empresa de venta de **artículos** de computación: ofrece artículos a través de una Web que pueden agregarse a un **carrito de compras** y posee un **inventario**.

El proyecto implementa tres clases principales: Producto, Inventario y Carrito.

### Etapas del proyecto:

- 1) Desarrollo de clases y objetos
- 2) Creación de la base de datos
- 3) Implementación de la API en Flask
- 4) Despliegue en servidor PythonAnywhere
- 5) Codificación del Front-End

## ETAPA 1: DESARROLLO DE CLASES Y OBJETOS

Las clases nos permitirán crear objetos para nuestro proyecto.

---

### Clase PRODUCTO

---



La **Clase PRODUCTO** es una representación de un producto en el sistema y contendrá los datos y las operaciones relacionadas con un producto.

Esta clase contiene las propiedades y métodos necesarios para manipular y acceder a los datos de un producto específico.

Sus **propiedades** son:

- ✓ **Código:** Identificador
- ✓ **Descripción:** Nombre
- ✓ **Cantidad:** Disponibilidad en stock
- ✓ **Precio:** Valor de venta

Estas propiedades son **atributos de instancia**, lo que significa que son variables específicas de cada instancia de la clase. Cada vez que creas un objeto Producto, puedes acceder y modificar estas propiedades utilizando la notación de punto.

Sus **métodos** son:

- **\_\_init\_\_(self, codigo, descripcion, cantidad, precio):** Este es el constructor de la clase que se ejecuta al crear una instancia de la clase. Recibe los parámetros codigo, descripcion, cantidad y precio para inicializar las propiedades correspondientes del producto.
- **modificar(self, nueva\_descripcion, nueva\_cantidad, nuevo\_precio):** Este método permite modificar los datos de un producto. Recibe los nuevos valores para la descripción, cantidad y precio del producto y actualiza las propiedades correspondientes. No permite cambiar el código identificador.

Luego, utilizaremos esta clase en otras clases para implementar la funcionalidad de agregar, consultar, modificar y listar productos, así como el manejo del carrito de compras.

```
class Producto:
    # Definimos el constructor e inicializamos los atributos de instancia
    def __init__(self, codigo, descripcion, cantidad, precio):
        self.codigo = codigo          # Código
        self.descripcion = descripcion # Descripción
        self.cantidad = cantidad      # Cantidad disponible (stock)
        self.precio = precio          # Precio

    # Este método permite modificar un producto.
    def modificar(self, nueva_descripcion, nueva_cantidad, nuevo_precio):
        self.descripcion = nueva_descripcion # Modifica la descripción
        self.cantidad = nueva_cantidad      # Modifica la cantidad
        self.precio = nuevo_precio         # Modifica el precio
```

Así podemos probar el funcionamiento de la clase:

```
# Programa principal
producto = Producto(1, 'Teclado USB 101 teclas', 10, 4500)
# Accedemos a los atributos del objeto
print(f'{producto.codigo} | {producto.descripcion} | {producto.cantidad} | {producto.precio}')
# Modificar los datos del producto
producto.modificar('Teclado Mecánico USB', 20, 4800)
print(f'{producto.codigo} | {producto.descripcion} | {producto.cantidad} | {producto.precio}')
```

La salida por pantalla sería la siguiente:

```
1 | Teclado USB 101 teclas | 10 | 4500
1 | Teclado Mecánico USB | 20 | 4800
```

---

### Clase INVENTARIO

---



La **Clase INVENTARIO** es una representación de un inventario de productos en el sistema. Cuando creamos una instancia de esta clase, se inicializa una lista vacía que almacenará los productos.

Sus **propiedades** son:

- ✓ **Productos:** Lista de productos disponibles

Sus **métodos** son:

- **\_\_init\_\_(self):** El constructor de la clase Inventario se ejecuta al crear una instancia de la clase. No requiere parámetros y se encarga de inicializar la lista de productos vacía.
- **agregar\_producto(self, codigo, descripcion, cantidad, precio):** Permite agregar un nuevo producto al inventario. Recibe los parámetros codigo, descripcion, cantidad y precio que representan los datos del producto que queremos agregar. Dentro del método creamos un

objeto de la clase `Producto` con los valores proporcionados y se agrega a la lista de productos del inventario.

- **`consultar_producto(self, codigo)`**: Permite consultar datos de un producto que está en el inventario. Recibe el código del producto que queremos buscar, luego busca en la lista de productos del inventario y devuelve el objeto `Producto` correspondiente si se encuentra, o *False* si no se encuentra ningún producto con el código dado.
- **`modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio)`**: Permite modificar datos de productos que están en el inventario. Recibe el código del producto, la nueva descripción, nueva cantidad en stock y nuevo precio, luego realiza la búsqueda del producto a través del método `consultar_producto` de la clase `Inventario` y en caso de encontrarlo llama al método `modificar` de la clase `Producto`.
- **`eliminar_producto(self, codigo)`**: Recibe como parámetro el código del producto que se desea eliminar. Dentro del método, se realiza un bucle que recorre la lista de productos del inventario. Se compara el código del producto actual con el código proporcionado. Si se encuentra un producto con el código coincidente, se utiliza el método *remove* para eliminar ese producto de la lista. Luego se muestra un mensaje indicando que el producto ha sido eliminado. Si el bucle termina sin encontrar un producto con el código dado, se muestra un mensaje indicando que el producto no fue encontrado.
- **`listar_productos(self)`**: Imprime en la terminal una lista con los datos de los productos que existen en el inventario. Recorre la lista de productos y muestra las propiedades de cada producto, como el código, descripción, cantidad y precio, en un formato legible.

Estas funcionalidades nos permiten gestionar el inventario de productos de manera eficiente y realizar operaciones como agregar nuevos productos, consultar su información y obtener una lista actualizada de los productos disponibles.

```
class Inventario:
    # Definimos el constructor e inicializamos los atributos de instancia
    def __init__(self):
        self.productos = [] # Lista de productos en el inventario (variable
                             de clase)

    # Este método permite crear objetos de la clase "Producto" y agregarlos
    al inventario.
    def agregar_producto(self, codigo, descripcion, cantidad, precio):
        nuevo_producto = Producto(codigo, descripcion, cantidad, precio)
        self.productos.append(nuevo_producto) # Agrega un nuevo producto a
        la lista

    # Este método permite consultar datos de productos que están en el
    inventario
    # Devuelve el producto correspondiente al código proporcionado o False si
    no existe.
    def consultar_producto(self, codigo):
        for producto in self.productos:
            if producto.codigo == codigo:
                return producto # Retorna un objeto
        return False

    # Este método permite modificar datos de productos que están en el
    inventario
```

```
# Utiliza el método consultar_producto del inventario y modificar del
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)

# Este método elimina el producto indicado por codigo de la lista
mantenida en el inventario.
def eliminar_producto(self, codigo):
    eliminar = False
    for producto in self.productos:
        if producto.codigo == codigo:
            eliminar = True
            producto_eliminar = producto
    if eliminar == True:
        self.productos.remove(producto_eliminar)
        print(f'Producto {codigo} eliminado.')
    else:
        print(f'Producto {codigo} no encontrado.')

# Este método imprime en la terminal una lista con los datos de los
productos que figuran en el inventario.
def listar_productos(self):
    print("-"*50)
    print("Lista de productos en el inventario:")
    print("Código\tDescripción\t\t\tCant\tPrecio")
    for producto in self.productos:
        print(f'{producto.codigo}\t{producto.descripcion}\t{producto.cant
idad}\t{producto.precio}')
    print("-"*50)
```

Así podemos probar el funcionamiento de la clase:

```
# Programa principal
# Crear una instancia de la clase Inventario
mi_inventario = Inventario()

# Agregar productos
mi_inventario.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500)
mi_inventario.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500)
mi_inventario.agregar_producto(3, 'Monitor LCD 22 pulgadas', 15, 52500)
mi_inventario.agregar_producto(4, 'Monitor LCD 27 pulgadas', 25, 78500)
mi_inventario.agregar_producto(5, 'Mouse Pad color azul', 5, 500)

# Consultar un producto
producto = mi_inventario.consultar_producto(3)
if producto != False:
```

```
print(f'Producto encontrado:\nCódigo: {producto.codigo}\nDescripción: {producto.descripcion}\nCantidad: {producto.cantidad}\nPrecio: {producto.precio}')
else:
    print("Producto no encontrado.")

# Modificar un producto
mi_inventario.modificar_producto(3, 'Monitor LCD 24 pulgadas', 5, 62000)

# Listar todos los productos
mi_inventario.listar_productos()

# Eliminar un producto
mi_inventario.eliminar_producto(2)

# Confirmamos que haya sido eliminado
mi_inventario.listar_productos()
```

La salida por pantalla sería la siguiente:

```
Producto encontrado:
Código: 3
Descripción: Monitor LCD 22 pulgadas
Cantidad: 15
Precio: 52500
-----
Lista de productos en el inventario:
Código Descripción Cant Precio
1 Teclado USB 101 teclas 10 4500
2 Mouse USB 3 botones 5 2500
3 Monitor LCD 24 pulgadas 5 62000
4 Monitor LCD 27 pulgadas 25 78500
5 Mouse Pad color azul 5 500
-----
Producto 2 eliminado.
-----
Lista de productos en el inventario:
Código Descripción Cant Precio
1 Teclado USB 101 teclas 10 4500
3 Monitor LCD 24 pulgadas 5 62000
4 Monitor LCD 27 pulgadas 25 78500
5 Mouse Pad color azul 5 500
-----
```

En este ejemplo, creamos una instancia de la **clase Inventario** llamada `mi_inventario`. Luego, utilizamos el método **agregar\_producto** para agregar cinco productos al inventario, cada uno con un código, descripción, cantidad y precio específicos.

Después, utilizamos el método **consultar\_producto** para buscar un producto en el inventario utilizando su código. Si el producto se encuentra, se muestra su información. En caso contrario, se muestra un mensaje indicando que el producto no fue encontrado.



A continuación, utilizamos el método **modificar\_producto** para actualizar los datos de un producto en el inventario. En este caso, modificamos el producto con código 3 cambiando su descripción, cantidad y precio.

Utilizamos el método **listar\_productos** para mostrar en pantalla la lista completa de productos en el inventario, incluyendo los cambios realizados.

A continuación, el método **eliminar\_producto** recibe como parámetro el código del producto que se desea eliminar. Se realiza un bucle que recorre la lista de productos del inventario y se compara el código del producto actual con el código proporcionado. Si se encuentra un producto con el código coincidente, se elimina ese producto de la lista y se muestra un mensaje indicando que el producto ha sido eliminado. Si el bucle termina sin encontrar un producto con el código dado, se muestra un mensaje indicando que el producto no fue hallado.

**NOTA:** Debes asegurarte de que el código del producto sea único y no se repita en el inventario, ya que la eliminación se basa en el código del producto y al dar las altas no estamos validando esta situación. Más adelante el inventario se implementa con una tabla de la base de datos, y podemos hacer que el código o id sea único.

---

### Clase CARRITO

---



La **Clase CARRITO** representa un carrito de compras donde se pueden agregar y quitar productos. Permite realizar operaciones como agregar un producto al carrito, quitar un producto y mostrar el contenido del carrito.

Cuando creamos una instancia de esta clase, se inicializa una lista vacía que almacenará los productos que están en el carrito.

Sus **propiedades** son:

- ✓ **Items:** Lista de productos disponibles en el carrito

Sus **métodos** son:

- **\_\_init\_\_(self):** El constructor de la clase Carrito se ejecuta al crear una instancia de la clase. No requiere parámetros y se encarga de inicializar la lista de ítems vacía.
- **agregar(self, codigo, cantidad, inventario):** Permite agregar un producto al carrito de compras. Recibe como parámetros el código numérico del producto, la cantidad que se desea agregar y el inventario. El método realiza las siguientes verificaciones:
  1. Verifica si el producto existe en el inventario utilizando el método **consultar\_producto** de la clase Inventario. Si el producto no existe, muestra un mensaje de error.
  2. Verifica si la cantidad en stock del producto es suficiente para agregar al carrito. Si la cantidad es insuficiente, muestra un mensaje de error.Si el producto ya está en el carrito, actualiza la cantidad del producto en el carrito sumándole la cantidad deseada.  
Si el producto no está en el carrito, lo agrega como un nuevo ítem al carrito.
- **quitar(self, codigo, cantidad, inventario):** Permite quitar un producto del carrito de compras. Recibe como parámetros el código numérico del producto, la cantidad que se desea quitar y el inventario. El método realiza las siguientes verificaciones:
  1. Verifica si el producto está en el carrito. Si no se encuentra en el carrito, muestra un mensaje de error.
  2. Verifica si la cantidad a quitar es mayor a la cantidad del producto en el carrito. Si es mayor, muestra un mensaje de error. Si la cantidad a quitar es menor o igual a la cantidad del producto en el carrito, disminuye la cantidad del producto en el carrito en la cantidad deseada. Si la cantidad resultante es cero, elimina el producto del carrito.

- **mostrar(self):** Muestra en pantalla el contenido del carrito de compras. Recorre la lista de productos en el carrito y muestra la información de cada producto, como el código, la descripción, la cantidad y el precio.

La clase Carrito utiliza el método consultar\_producto de la clase Inventario para verificar la existencia de los productos en el inventario y obtener su información.

Para utilizar la clase Carrito, se debe crear una instancia de la misma y luego se pueden llamar a sus métodos para realizar las operaciones de agregar, quitar y mostrar el contenido del carrito.

```
class Carrito:
    # Definimos el constructor e inicializamos los atributos de instancia
    def __init__(self):
        self.items = [] # Lista de items en el carrito (variable de clase)

    # Este método permite agregar productos del inventario al carrito.
    def agregar(self, codigo, cantidad, inventario):
        # Nos aseguramos que el producto esté en el inventario
        producto = inventario.consultar_producto(codigo)
        if producto is False:
            print("El producto no existe.")
            return False

        # Verificamos que la cantidad en stock sea suficiente
        if producto.cantidad < cantidad:
            print("Cantidad en stock insuficiente.")
            return False

        # Si existe y hay stock, vemos si ya existe en el carrito.
        for item in self.items:
            if item.codigo == codigo:
                item.cantidad += cantidad
                # Actualizamos la cantidad en el inventario
                producto = inventario.consultar_producto(codigo)
                producto.modificar(producto.descripcion, producto.cantidad -
cantidad, producto.precio)
                return True

        # Si no existe en el carrito, lo agregamos como un nuevo item.
        nuevo_item = Producto(codigo, producto.descripcion, cantidad,
producto.precio)
        self.items.append(nuevo_item)
        # Actualizamos la cantidad en el inventario
        producto = inventario.consultar_producto(codigo)
        producto.modificar(producto.descripcion, producto.cantidad -
cantidad, producto.precio)
        return True

    # Este método quita unidades de un elemento del carrito, o lo elimina.
    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)
    # Actualizamos la cantidad en el inventario
    producto = inventario.consultar_producto(codigo)
    producto.modificar(producto.descripcion, producto.cantidad +
cantidad, producto.precio)
    return True

# Si el bucle finaliza sin novedad, es que ese producto NO ESTA en el
carrito.
print("El producto no se encuentra en el carrito.")
return False

def mostrar(self):
    print("-"*50)
    print("Lista de productos en el carrito:")
    print("Código\tDescripción\t\tCant\tPrecio")
    for item in self.items:
        print(f'{item.codigo}\t{item.descripcion}\t{item.cantidad}\t{item
.precio}')
    print("-"*50)
```

Así podemos probar el funcionamiento de la clase y todo lo implementado hasta aquí:

```
# Programa principal

# -----
# Ejemplo de uso de las clases y objetos definidos antes:
# -----

# Crear una instancia de la clase Inventario
mi_inventario = Inventario()

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

# Crear 3 productos y agregarlos al inventario
mi_inventario.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500)
mi_inventario.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500)
mi_inventario.agregar_producto(3, 'Monitor LCD 22 pulgadas', 15, 52500)

# Listar todos los productos del inventario
mi_inventario.listar_productos()

# Agregar 2 productos al carrito
mi_carrito.agregar(1, 2, mi_inventario) # Agregar 2 unidades del producto con
código 1 al carrito
```

```
mi_carrito.agregar(3, 4, mi_inventario) # Agregar 1 unidad del producto con
código 3 al carrito
mi_carrito.quitar (1, 1, mi_inventario) # Quitar 1 unidad del producto con
código 1 al carrito
# Listar todos los productos del carrito
mi_carrito.mostrar()
# Quitar 1 producto al carrito
mi_carrito.quitar (1, 1, mi_inventario) # Quitar 1 unidad del producto con
código 1 al carrito
# Listar todos los productos del carrito
mi_carrito.mostrar()
# Mostramos el inventario
mi_inventario.listar_productos()
```

-----  
Lista de productos en el inventario:

Código	Descripción	Cant	Precio
1	Teclado USB 101 teclas	10	4500
2	Mouse USB 3 botones	5	2500
3	Monitor LCD 22 pulgadas	15	52500

-----

-----  
Lista de productos en el carrito:

Código	Descripción	Cant	Precio
1	Teclado USB 101 teclas	1	4500
3	Monitor LCD 22 pulgadas	4	52500

-----

-----  
Lista de productos en el carrito:

Código	Descripción	Cant	Precio
3	Monitor LCD 22 pulgadas	4	52500

-----

-----  
Lista de productos en el inventario:

Código	Descripción	Cant	Precio
1	Teclado USB 101 teclas	10	4500
2	Mouse USB 3 botones	5	2500
3	Monitor LCD 22 pulgadas	11	52500

-----

## ETAPA 2: CREACIÓN DE LA BASE DE DATOS

### 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 cómo 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.*

Luego 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.

**Importante:** las siguientes líneas las escribimos en el mismo archivo **antes** de la definición de las clases.

```
import sqlite3

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

def get_db_connection():
    print("Obteniendo conexión...") # Para probar que se ejecuta la función
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla 'productos' si no existe
def create_table():
    print("Creando tabla productos...") # Para probar que se ejecuta la función
    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():
    print("Creando la BD...") # Para probar que se ejecuta la función
    conn = sqlite3.connect(DATABASE)
    conn.close()
    create_table()

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

En la primera línea, importamos la biblioteca `sqlite3` para trabajar con la base de datos. 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 con `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 y adaptar el código de cada clase.

---

### Modificando la clase Producto

---

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

```
class Producto:
    # Definimos el constructor e inicializamos los atributos de instancia
    def __init__(self, codigo, descripcion, cantidad, precio):
        self.codigo = codigo          # Código
        self.descripcion = descripcion # Descripción
        self.cantidad = cantidad      # Cantidad disponible (stock)
        self.precio = precio          # Precio

    # Este método permite modificar un producto.
    def modificar(self, nueva_descripcion, nueva_cantidad, nuevo_precio):
        self.descripcion = nueva_descripcion # Modifica la descripción
```

```
self.cantidad = nueva_cantidad      # Modifica la cantidad
self.precio = nuevo_precio          # Modifica el precio
```

---

### Modificando la 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.

Reemplazamos el método init que teníamos por lo siguiente:

```
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.

Reemplazamos el método `agregar_producto` que teníamos por lo siguiente:

```
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)
    sql = f'INSERT INTO productos VALUES ({codigo}, "{descripcion}",
{cantidad}, {precio});'
    self.cursor.execute(sql)
    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.

**sql = f'INSERT INTO productos VALUES ({codigo}, "{descripcion}", {cantidad}, {precio});'** Genera un string a través de f-string que será la sentencia de sql de inserción. Los valores de codigo, descripcion, cantidad y precio se pasan como una tupla.

**self.cursor.execute(sql):** Ejecuta una consulta SQL utilizando el cursor para insertar una nueva fila en la tabla "productos".

**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):
    sql = f'SELECT * FROM productos WHERE codigo = {codigo};'
    self.cursor.execute(sql)
    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:

**sql = f'SELECT \* FROM productos WHERE codigo = {codigo};'** Genera un string a través de f-string que será la sentencia de sql de selección. El valor de codigo se interpola dentro del string.

**self.cursor.execute(sql):** Ejecuta una consulta SQL utilizando el cursor para seleccionar todas las columnas de la tabla "productos" donde el código coincide con el valor proporcionado.

**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 luego se 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

Este método 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)
    sql = f'UPDATE productos SET descripcion = "{nueva_descripcion}",
cantidad = {nueva_cantidad}, precio = {nuevo_precio} WHERE codigo =
{codigo};'
    self.cursor.execute(sql)
    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.

**sql = f'UPDATE productos SET descripcion = "{nueva\_descripcion}", cantidad = {nueva\_cantidad}, precio = {nuevo\_precio} WHERE codigo = {codigo};':** Genera un string a través de f-string que será la sentencia de sql de actualización. Los valores de `nueva_descripcion`, `nueva_cantidad` y `nuevo_precio` se pasan como una tupla, mientras que `codigo` servirá como condición del `WHERE`.

**self.cursor.execute(sql):** Se ejecuta una consulta SQL utilizando el cursor para actualizar los datos del producto en la tabla "productos".

**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 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):
    sql = f'DELETE FROM productos WHERE codigo = {codigo};'
    self.cursor.execute(sql)
    if self.cursor.rowcount > 0:
        print(f'Producto {codigo} eliminado.')
        self.conexion.commit()
    else:
        print(f'Producto {codigo} no encontrado.')
```

Aquí está la descripción del método `eliminar_producto` paso a paso:

**sql = f'DELETE FROM productos WHERE codigo = {codigo};':** Genera un string a través de f-string que será la sentencia de sql de eliminación. `Codigo` servirá como condición del `WHERE`.

**self.cursor.execute(sql):** Ejecuta una consulta SQL utilizando el cursor para eliminar el producto de la tabla "productos" donde el código coincide con el valor proporcionado.

**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.

### *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("-"*50)
    print("Lista de productos en el inventario:")
    print("Código\tDescripción\tCant\tPrecio")
    self.cursor.execute("SELECT * FROM productos")
    rows = self.cursor.fetchall()
    for row in rows:
        codigo, descripcion, cantidad, precio = row
        print(f'{codigo}\t{descripcion}\t{cantidad}\t{precio}')
    print("-"*50)
```

**print("-" \* 50):** Imprime una línea horizontal de guiones para separar visualmente la lista de productos. Las siguientes dos líneas imprimen un título y los encabezados de las columnas donde se mostrará la información.

**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.

---

## Modificando la 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 e inicializa el atributo items como una lista vacía.

```
def __init__(self):
    self.conexion = sqlite3.connect('inventario.db') # Conexión a la BD
    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
            sql = f'UPDATE productos SET cantidad = cantidad - {cantidad} WHERE codigo = {codigo};'
            self.cursor.execute(sql)
            self.conexion.commit()
            return True

    nuevo_item = Producto(codigo, producto.descripcion, cantidad,
                           producto.precio)
    self.items.append(nuevo_item)
    sql = f'UPDATE productos SET cantidad = cantidad - {cantidad} WHERE codigo = {codigo};'
    self.cursor.execute(sql)
    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 a través de **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)
            sql = f'UPDATE productos SET cantidad = cantidad + {cantidad}
WHERE codigo = {codigo};'
            self.cursor.execute(sql)
            self.conexion.commit()
            return True
```

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 a través de **self.conexion.commit()** para guardar los cambios en la base de datos y se devuelve True.

### *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("-"*50)
    print("Lista de productos en el carrito:")
    print("Código\tDescripción\tCant\tPrecio")
    for item in self.items:
        print(f'{item.codigo}\t{item.descripcion}\t{item.cantidad}\t{item.precio}')
    print("-"*50)
```

Se imprime una línea de guiones ("-") repetida 50 veces para crear una separación visual en la salida. Las siguientes dos líneas imprimen un título y los encabezados de las columnas donde se mostrará la información.

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

- Para cada item, se imprime su código, descripción, cantidad y precio utilizando la función print.
- Después de imprimir los detalles de un item, se imprime otra línea de guiones ("-") repetida 50 veces para separar visualmente los detalles de los productos.

---

### Ejemplo de uso de las clases y objetos definidos

---

Los siguientes códigos muestran 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.

**Nota:** La base de datos se crea en la carpeta raíz del proyecto. Si deseamos probar nuevamente la creación de la BD para agregar productos debemos eliminar el archivo "inventario.db"

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

# Crear una instancia de la clase Inventario
mi_inventario = Inventario()

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

# Consultar algún producto del inventario
print(mi_inventario.consultar_producto(3)) #Existe, se muestra la dirección de memoria
print(mi_inventario.consultar_producto(4)) #No existe, se muestra False

# Listar los productos del inventario
mi_inventario.listar_productos()
```

Por el momento la salida es la siguiente:

```
Creando la BD...
Creando tabla productos...
Obteniendo conexión...
Obteniendo conexión...
<__main__.Producto object at 0x0000027F5A5B3AC0>
False
```

-----

Lista de productos en el inventario:

Código	Descripción	Cant	Precio
1	Producto 1	10	19.99
2	Producto 2	5	9.99
3	Producto 3	15	29.99

-----

Descripción paso a paso del código:

1. Se crea una instancia de la clase Inventario llamada mi\_inventario. La segunda línea de "Obteniendo conexión..." se imprime al instanciar la clase Inventario.
2. Se agregan productos al inventario utilizando el método agregar\_producto de la instancia de Inventario (mi\_inventario). Se agregan tres productos con diferentes códigos, descripciones, cantidades y precios.
3. Luego comprobamos la existencia del producto imprimiendo su dirección de memoria, en caso de que no exista el producto veremos que se imprime False.
4. Imprimimos la lista de productos en el inventario

Ahora probaremos las siguientes líneas:

```
# Modificar un producto del inventario
mi_inventario.modificar_producto(2, "Mouse Rojo", 10, 19.99)

# Listar nuevamente los productos del inventario para ver la modificación
mi_inventario.listar_productos()

# Eliminar un producto
mi_inventario.eliminar_producto(3)

# Listar nuevamente los productos del inventario para ver la eliminación
mi_inventario.listar_productos()
```

Descripción paso a paso del código:

1. Modificamos un producto del inventario.
2. Imprimimos la lista de productos para comprobar la modificación.
3. Eliminamos el producto del inventario.
4. Volvemos a imprimir la lista de productos para comprobar la eliminación

La salida es la siguiente:

-----

Lista de productos en el inventario:

Código	Descripción	Cant	Precio
1	Producto 1	10	19.99
2	Mouse Rojo	10	19.99
3	Producto 3	15	29.99

```
-----  
Producto 3 eliminado.  
-----  
Lista de productos en el inventario:  
Código Descripción Cant Precio  
1 Producto 1 10 19.99  
2 Mouse Rojo 10 19.99  
-----
```

El primer listado muestra el producto número 2 modificado (modificamos descripción, cantidad y precio).

La siguiente línea nos muestra la impresión del método `eliminar_producto(3)` y luego volvemos a imprimir los datos del inventario, confirmando la eliminación del producto 3.

Ahora comprobaremos el funcionamiento de los métodos de la clase `Carrito`:

```
# Crear una instancia de la clase Carrito  
mi_carrito = Carrito()  
# Agregar 2 unidades del producto con código 1 al carrito  
mi_carrito.agregar(1, 2, mi_inventario)  
# Agregar 1 unidad del producto con código 2 al carrito  
mi_carrito.agregar(2, 1, mi_inventario)  
  
# Mostrar el contenido del carrito y del inventario  
mi_carrito.mostrar()  
mi_inventario.listar_productos()
```

Descripción paso a paso del código:

1. Se crea una instancia de la clase `Carrito` llamada `mi_carrito`.
2. 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 2 al carrito.
3. Se muestra el contenido del carrito.
4. Se muestra el contenido del inventario.

Por el momento la salida es la siguiente:

```
-----  
Lista de productos en el carrito:  
Código Descripción Cant Precio  
1 Producto 1 2 19.99  
2 Mouse Rojo 1 19.99  
-----  
-----  
Lista de productos en el inventario:  
Código Descripción Cant Precio  
1 Producto 1 8 19.99  
2 Mouse Rojo 9 19.99  
-----
```

Aquí podemos notar que se agregaron los productos al carrito y se descontaron del inventario. Ahora haremos el proceso inverso: “devolveremos” los productos del carrito al inventario:



```
# Quitar 1 unidad del producto con código 1 al carrito y 1 unidad del
producto con código 2 al carrito
mi_carrito.quitar(1, 1, mi_inventario)
mi_carrito.quitar(2, 1, mi_inventario)

# Mostrar el contenido del carrito y del inventario
mi_carrito.mostrar()
mi_inventario.listar_productos()
```

Descripción paso a paso del código:

1. Se quita una unidad del producto con código 1 y una unidad del producto con código 2 del carrito utilizando el método `quitar` de la instancia de Carrito (`mi_carrito`).
2. Se muestra el contenido del carrito.
3. Se muestra el contenido del inventario.

La salida será la siguiente:

```
-----
Lista de productos en el carrito:
Código Descripción Cant Precio
1 Producto 1 1 19.99
-----
Lista de productos en el inventario:
Código Descripción Cant Precio
1 Producto 1 9 19.99
2 Mouse Rojo 10 19.99
-----
```

Al quitar 1 unidad del producto “Mouse rojo” esta vuelve al inventario y se elimina del carrito. Mientras que al quitar 1 unidad del producto “Producto 1” se descuenta del carrito y se reintegra al inventario.

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

### ETAPA 3: IMPLEMENTACIÓN DE LA API EN FLASK

Necesitamos realizar modificaciones en el código existente para que pueda ser publicado en el servidor de [PythonAnywhere](https://pythonanywhere.com/), y crear una API con Flask para interactuar desde el frontend. Para lograr esto necesitamos realizar varios cambios en el código que incluyen, entre otras, estas modificaciones:

- Se importa el módulo *Flask* y *jsonify* de la biblioteca Flask para facilitar la creación de la API.
- Se agregan decoradores de Flask a cada método para definir las rutas de la API.
- Se utilizan los métodos *jsonify* para devolver las respuestas en formato JSON.
- Se reciben los datos del frontend mediante el objeto *request.json*.
- Se crea una instancia de la clase Inventario y Carrito en cada ruta según sea necesario.

---

#### Instalar el módulo Flask

---

Para comenzar, necesitaremos instalar el módulo Flask en Python. Puedes seguir estos pasos:

1. Abre una terminal o línea de comandos en tu sistema operativo.
2. Asegúrate de tener Python instalado en tu computadora. Puedes verificarlo ejecutando el comando **python --version** en la terminal. Si está instalado verás la versión que tienes.
3. Ejecuta el siguiente comando en la terminal para instalar Flask utilizando **pip**, el administrador de paquetes de Python:

```
pip install flask
```

Si estás utilizando Python 3, es posible que necesites usar pip3 en lugar de pip:

```
pip3 install flask
```

4. Espera a que pip descargue e instale Flask y sus dependencias.
5. Una vez completada la instalación, puedes comenzar a utilizar Flask en tus aplicaciones Python importando el módulo flask y la clase Flask, como veremos más adelante.

*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.*

La clase Flask es la piedra angular de una aplicación Flask y se utiliza para crear una instancia de la aplicación.

---

### Código de la aplicación

---

**Importante:** las siguientes líneas las escribimos en el mismo archivo antes de la creación de la Base de datos.

Escribimos el código necesario para importar los módulos necesarios, crear la instancia de Flask y establecer la conexión con la base de datos. Como antes, se utiliza Flask y SQLite para crear una aplicación web con una base de datos SQLite.

En esta parte, el código es prácticamente el mismo que vimos en la etapa anterior. Solo hemos agregado una línea. Por comodidad agregaremos el resto del código:

```
import sqlite3
from flask import Flask, jsonify, request

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

def get_db_connection():
    print("Obteniendo conexión...") # Para probar que se ejecuta la función
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla 'productos' si no existe
def create_table():
    print("Creando tabla productos...") # Para probar que se ejecuta la
función
```

```
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():
    print("Creando la BD...") # Para probar que se ejecuta la función
    conn = sqlite3.connect(DATABASE)
    conn.close()
    create_table()

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

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

**from flask import Flask, jsonify, request:** importa clases y funciones del módulo Flask que necesitamos en nuestro proyecto.

---

### Configuración y rutas de la API Flask

---

La sección "Configuración y rutas de la API Flask" del código es completamente nueva. Este código se coloca después de las definiciones de las clases, justo antes de implementar los decoradores y funciones de nuestra API. Recordemos que este tutorial no analiza el código fuente de forma lineal, sino que lo hace explicando cada sección importante.

#### 1) *Importación de los módulos y creación de la aplicación Flask:*

Colocaremos este código al finalizar la definición de la clase Carrito:

```
app = Flask(__name__)

carrito = Carrito() # Instanciamos un carrito
inventario = Inventario() # Instanciamos un inventario
```

Importamos el módulo Flask y creamos una instancia de la clase Flask llamada app. Esta instancia representa nuestra aplicación Flask.

También instanciamos las clases Carrito e Inventario para crear los respectivos objetos.

#### 2) *Decorador @app.route('/productos/<int:codigo>', methods=['GET']) y función obtener\_producto(codigo):*

En la ruta llamada `/productos/<int:codigo>` podemos obtener los datos de un producto según su código.

```
# Ruta para obtener los datos de un producto según su código
@app.route('/productos/<int:codigo>', methods=['GET'])
def obtener_producto(codigo):
    producto = inventario.consultar_producto(codigo)
    if producto:
        return jsonify({
            'codigo': producto.codigo,
            'descripcion': producto.descripcion,
            'cantidad': producto.cantidad,
            'precio': producto.precio
        }), 200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

Veamos cómo funciona:

En la definición de la ruta, utilizamos `<int:codigo>` para indicar que esperamos un parámetro entero llamado `codigo`. Esto significa que en la URL de la solicitud, debes proporcionar el código del producto después de `/productos/`.

- Cuando se recibe una solicitud GET a esta ruta, se llama a la función `obtener_producto(codigo)`.
- Luego, llamamos al método `consultar_producto(codigo)` del objeto `inventario` para obtener los datos del producto correspondiente al código proporcionado.
- Si el producto existe en la base de datos, se crea un objeto `producto` con los detalles del producto, como el código, descripción, cantidad y precio.
- Utilizamos la función `jsonify()` de Flask para convertir el objeto `producto` en una respuesta JSON.
- Devolvemos la respuesta JSON con un código de estado 200, indicando que la solicitud se procesó correctamente y los datos del producto se encuentran en la respuesta.
- Si el producto no se encuentra en la base de datos, devolvemos un mensaje de error JSON con un código de estado 404, indicando que el producto no fue encontrado.

Con esta funcionalidad, los usuarios podrán obtener los datos de un producto específico utilizando el código del producto como referencia. Esto puede ser útil para mostrar los detalles de un producto en una página de detalles o cualquier otra funcionalidad relacionada con la obtención de datos específicos.

### 3) Decorador `@app.route('/')` y función `index()`:

```
# Ruta para obtener la lista de productos del inventario
@app.route('/')
def index():
    return 'API de Inventario'
```

El decorador `@app.route('/')` establece la ruta de la URL ("/") a la cual se asocia la función `index()`. En este caso, cuando accedemos a la página principal de la API, se ejecuta la función `index()` y devuelve la cadena de texto "API de Inventario". Aquí podríamos devolver el código HTML de una "página principal" que presente la API, o describa sus características. Queda como sugerencia para el alumno.

4) Decorador `@app.route('/productos', methods=['GET'])` y función `obtener_productos()`:

```
# Ruta para obtener la lista de productos del inventario
@app.route('/productos', methods=['GET'])
def obtener_productos():
    return inventario.listar_productos()
```

El decorador `@app.route('/productos', methods=['GET'])` establece la ruta de la URL ("`/productos`") y el método HTTP permitido (en este caso, solo permitimos el método GET) para la función `obtener_productos()`. Cuando se realiza una solicitud GET a la ruta "`/productos`", se ejecuta esta función.

Dentro de la función `obtener_productos()`, se llama al método `listar_productos()` del objeto de la clase `Inventario`, que devuelve una lista de productos. Luego, utilizamos la función `jsonify()` para convertir la lista de productos en un objeto JSON y lo devolvemos como respuesta.

5) Decorador `@app.route('/productos', methods=['POST'])` y función `agregar_producto()`:

```
# Ruta para agregar un producto al inventario
@app.route('/productos', methods=['POST'])
def agregar_producto():
    codigo = request.json.get('codigo')
    descripcion = request.json.get('descripcion')
    cantidad = request.json.get('cantidad')
    precio = request.json.get('precio')
    return inventario.agregar_producto(codigo, descripcion, cantidad,
    precio)
```

El decorador `@app.route('/productos', methods=['POST'])` establece la ruta de la URL ("`/productos`") y el método HTTP permitido (en este caso, solo permitimos el método POST) para la función `agregar_producto()`. Cuando se realiza una solicitud POST a la ruta "`/productos`", se ejecuta esta función.

Dentro de la función `agregar_producto()`, utilizamos `request.json.get()` para obtener los datos enviados en el cuerpo de la solicitud POST como un objeto JSON. Luego, extraemos los valores de los campos "`codigo`", "`descripcion`", "`cantidad`" y "`precio`" del objeto JSON. A continuación, llamamos al método `agregar_producto()` del objeto de la clase `Inventario` para agregar un nuevo producto. Dependiendo del resultado de la operación, devolvemos una respuesta JSON con un mensaje correspondiente y un código de estado HTTP 200 (éxito) o 400 (error).

6) Decorador `@app.route('/productos/<int:codigo>', methods=['PUT'])` y función `modificar_producto()`:

```
# Ruta para modificar un producto del inventario
@app.route('/productos/<int:codigo>', methods=['PUT'])
def modificar_producto(codigo):
    nueva_descripcion = request.json.get('descripcion')
    nueva_cantidad = request.json.get('cantidad')
    nuevo_precio = request.json.get('precio')
    return inventario.modificar_producto(codigo, nueva_descripcion,
    nueva_cantidad, nuevo_precio)
```

El decorador `@app.route('/productos/<int:codigo>', methods=['PUT'])` establece la ruta de la URL ("`/productos/<codigo>`") y el método HTTP permitido (en este caso, solo permitimos el método PUT) para la función `modificar_producto()`. El `<int:codigo>` indica que se espera un parámetro entero llamado "codigo" en la URL. Cuando se realiza una solicitud PUT a la ruta "`/productos/<codigo>`", se ejecuta esta función.

Dentro de la función `modificar_producto()`, utilizamos `request.json.get()` para obtener los datos enviados en el cuerpo de la solicitud PUT como un objeto JSON. Luego, extraemos los valores de los campos "descripcion", "cantidad" y "precio" del objeto JSON.

Llamamos al método `modificar_producto()` del objeto de la clase `Inventario` para modificar el producto correspondiente al código recibido. Devolvemos una respuesta JSON con un mensaje de éxito y un código de estado HTTP 200

### 7) Decorador `@app.route('/productos/<int:codigo>', methods=['DELETE'])` y función `eliminar_producto()`:

```
# Ruta para eliminar un producto del inventario
@app.route('/productos/<int:codigo>', methods=['DELETE'])
def eliminar_producto(codigo):
    return inventario.eliminar_producto(codigo)
```

El decorador `@app.route('/productos/<int:codigo>', methods=['DELETE'])` establece la ruta de la URL ("`/productos/<codigo>`") y el método HTTP permitido (en este caso, solo permitimos el método DELETE) para la función `eliminar_producto()`. El `<int:codigo>` indica que se espera un parámetro entero llamado "codigo" en la URL. Cuando se realiza una solicitud DELETE a la ruta "`/productos/<codigo>`", se ejecuta esta función.

Dentro de la función `eliminar_producto()`, llamamos al método `eliminar_producto()` del objeto de la clase `Inventario` para eliminar el producto correspondiente al código recibido. Devolvemos una respuesta JSON con un mensaje de éxito y un código de estado HTTP 200.

### 8) Decorador `@app.route('/carrito', methods=['POST'])` y función `agregar_carrito()`:

```
# Ruta para agregar un producto al carrito
@app.route('/carrito', methods=['POST'])
def agregar_carrito():
    codigo = request.json.get('codigo')
    cantidad = request.json.get('cantidad')
    inventario = Inventario()
    return carrito.agregar(codigo, cantidad, inventario)
```

Este código implementa una ruta en la API de Flask para agregar un producto al carrito. Define una ruta en la API de Flask que permite agregar un producto al carrito mediante una solicitud HTTP POST a la ruta `/carrito`. La función `agregar_carrito()` se encarga de procesar la solicitud, crear el carrito y agregar el producto utilizando la instancia de la clase `Carrito`.

- El decorador `@app.route('/carrito', methods=['POST'])` define la ruta `/carrito` y especifica que solo se permite el método POST para esta ruta. Esto significa que solo se puede realizar una solicitud HTTP POST a esta ruta en particular.
- La función `agregar_carrito()` se ejecutará cuando se realice una solicitud HTTP POST a la ruta `/carrito`. Esta función es responsable de agregar un producto al carrito.
- Se extraen los datos necesarios de la solicitud HTTP utilizando `request.json.get()`. En este caso, se obtiene el valor del parámetro `codigo` y `cantidad` del cuerpo de la solicitud JSON. Estos valores representan el código y la cantidad del producto que se va a agregar al carrito.

- Luego, se llama al método `agregar()` del objeto `carrito` para agregar el producto al carrito. Se le pasan como argumentos el código, la cantidad y el objeto `inventario`, previamente instanciado.
- Por último, el resultado de la llamada al método `agregar()` se devuelve como respuesta HTTP. Dependiendo de cómo esté implementado el método `agregar()` en la clase `Carrito`, podría devolver un mensaje de éxito si el producto se agrega correctamente, o un mensaje de error si hay algún problema.

### 9) Decorador `@app.route('/carrito', methods=['DELETE'])` y función `quitar_carrito()`:

```
# Ruta para quitar un producto del carrito
@app.route('/carrito', methods=['DELETE'])
def quitar_carrito():
    codigo = request.json.get('codigo')
    cantidad = request.json.get('cantidad')
    inventario = Inventario()
    return carrito.quitar(codigo, cantidad, inventario)
```

Este fragmento de código define una ruta en la API de Flask que permite quitar un producto del carrito mediante una solicitud HTTP DELETE a la ruta `/carrito`. La función `quitar_carrito()` se encarga de procesar la solicitud, crear el carrito y quitar el producto utilizando la instancia de la clase `Carrito`.

- El decorador `@app.route('/carrito', methods=['DELETE'])` define la ruta `/carrito` y especifica que solo se permite el método DELETE para esta ruta. Esto significa que solo se puede realizar una solicitud HTTP DELETE a esta ruta en particular.
- La función `quitar_carrito()` se ejecutará cuando se realice una solicitud HTTP DELETE a la ruta `/carrito`. Esta función es responsable de quitar un producto del carrito.
- Se extraen los datos necesarios de la solicitud HTTP utilizando `request.json.get()`. En este caso, se obtiene el valor del parámetro `codigo` y `cantidad` del cuerpo de la solicitud JSON. Estos valores representan el código y la cantidad del producto que se va a quitar del carrito.
- Luego, se llama al método `quitar()` del objeto `carrito` para quitar el producto del carrito. Se le pasan como argumentos el código, la cantidad y el objeto `inventario`, previamente instanciado.
- Por último, el resultado de la llamada al método `quitar()` se devuelve como respuesta HTTP. Dependiendo de cómo esté implementado el método `quitar()` en la clase `Carrito`, podría devolver un mensaje de éxito si el producto se quita correctamente, o un mensaje de error si hay algún problema.

### 10) Decorador `@app.route('/carrito', methods=['GET'])` y función `obtener_carrito()`:

```
# Ruta para obtener el contenido del carrito
@app.route('/carrito', methods=['GET'])
def obtener_carrito():
    return carrito.mostrar()
```

Esta sección del código es una ruta en la API Flask que responde a la solicitud GET en la URL `"/carrito"` para obtener y devolver el contenido del carrito.

- El decorador `@app.route('/carrito', methods=['GET'])` es un decorador de Flask que indica que esta función manejará solicitudes GET en la URL `"/carrito"`.
- La función `obtener_carrito()` se ejecutará cuando se realice una solicitud GET a la ruta `/carrito`.

- Finalmente, a través de **return carrito.mostrar()** se devuelve el resultado de llamar al método `mostrar()` en la instancia de la clase `Carrito`. Este método devuelve el contenido del carrito en formato JSON.

Estos son los decoradores Flask que definen las rutas y los métodos HTTP asociados a cada función. Las funciones manipulan los datos y devuelven respuestas JSON con mensajes y códigos de estado adecuados para la comunicación con el frontend. Ahora estamos listos para analizar los cambios que se han introducido en el código de cada clase.

**IMPORTANTE:** de aquí en adelante se mostrarán las modificaciones que es necesario llevar a cabo en los métodos de las clases previamente creadas.

---

### Clase Producto

---

La clase `Producto` sigue sin cambios, recordemos que sus instancias solo se utilizan como parte del inventario (citamos el código apra recordarlo):

```
# -----  
# Clase "Producto"  
# -----  
class Producto:  
    # Definimos el constructor e inicializamos los atributos de instancia  
    def __init__(self, codigo, descripcion, cantidad, precio):  
        self.codigo = codigo          # Código  
        self.descripcion = descripcion # Descripción  
        self.cantidad = cantidad      # Cantidad disponible (stock)  
        self.precio = precio          # Precio  
  
    # Este método permite modificar un producto.  
    def modificar(self, nueva_descripcion, nueva_cantidad, nuevo_precio):  
        self.descripcion = nueva_descripcion # Modifica la descripción  
        self.cantidad = nueva_cantidad      # Modifica la cantidad  
        self.precio = nuevo_precio          # Modifica el precio
```

---

### Clase inventario

---

#### Método `init`

Recordemos: 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()
```

Recordemos que 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.

**Nota:** este método no sufre cambios respecto a la etapa anterior.

### *Método agregar\_producto*

Este método permite agregar un nuevo producto al inventario. Recibe como parámetros el código, descripción, cantidad y precio del producto a agregar. En primer lugar, se realiza una verificación para determinar si ya existe un producto con el mismo código en la base de datos. Si se encuentra un producto con el mismo código, se imprime un mensaje de error indicando que ya existe un producto con ese código. En caso contrario, se crea una instancia de la clase Producto con los datos proporcionados y se inserta en la tabla productos de la base de datos utilizando una consulta SQL. Por último, se imprime un mensaje de éxito indicando que el producto ha sido agregado correctamente.

Reemplazamos el método agregar\_producto que teníamos por lo siguiente:

```
def agregar_producto(self, codigo, descripcion, cantidad, precio):
    producto_existente = self.consultar_producto(codigo)
    if producto_existente:
        return jsonify({'message': 'Ya existe un producto con ese
código.'}), 400
    nuevo_producto = Producto(codigo, descripcion, cantidad, precio)
    sql = f'INSERT INTO productos VALUES ({codigo}, "{descripcion}",
{cantidad}, {precio});'
    self.cursor.execute(sql)
    self.conexion.commit()
    return jsonify({'message': 'Producto agregado correctamente.'}), 200
```

Usaremos la función fetch desde el frontend para acceder a los métodos de la API que están alojados en, por ejemplo, **PythonAnywhere**.

Recordemos que la función fetch es una API de JavaScript que permite realizar solicitudes HTTP desde el navegador web. Puedes utilizarla para enviar solicitudes POST al método agregar\_producto de la API y enviar los datos del producto al servidor.

Veamos un ejemplo de cómo podríamos usar fetch para realizar la llamada al método **agregar\_producto** desde el frontend:

```
const url = 'https://tu-domino-en-pythonanywhere.com/productos'
const data = {
  codigo: 4,
  descripcion: 'Producto 4',
  cantidad: 20,
  precio: 49.99
}

fetch(url, {
  method: 'POST', headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(response => {
```

```
if (response.ok) {
    console.log('Producto agregado correctamente')
} else {
    console.log('Error al agregar el producto')
}
})
.catch(error => {
    console.error('Error de conexión:', error)
})
```

Recuerda que debes reemplazar **'https://tu-domino-en-pythonanywhere.com'** con la URL de tu aplicación alojada en PythonAnywhere. Asegúrate de que el dominio y la ruta sean correctos según tu configuración.

Al ejecutar este código en el navegador, se realizará una solicitud POST a la URL especificada, enviando los datos del producto en el cuerpo de la solicitud en formato JSON. Luego, la API procesará la solicitud y devolverá una respuesta. Dependiendo de si la operación es exitosa o no, el código en el bloque then manejará la respuesta correspondiente.

*Es importante tener en cuenta que, al realizar solicitudes desde el frontend a un servidor externo, debes asegurarte de que la API tenga habilitados los encabezados CORS (Cross-Origin Resource Sharing) para permitir solicitudes desde un dominio diferente al de la API. De lo contrario, es posible que encuentres restricciones de seguridad que bloqueen las solicitudes.*

**response.ok** es una propiedad booleana de la respuesta devuelta por la función fetch. Indica si la solicitud HTTP se realizó con éxito o no.

Cuando se realiza una solicitud HTTP, el servidor devuelve una respuesta con un código de estado HTTP. Un código de estado en el rango de 200 a 299 se considera exitoso, lo que significa que la solicitud se ha realizado correctamente. Por ejemplo, el código de estado 200 indica una respuesta exitosa.

Cuando **response.ok** es true, significa que la respuesta tiene un código de estado exitoso (en el rango de 200 a 299). En cambio, cuando **response.ok** es false, significa que la respuesta tiene un código de estado que indica un error, como un código de estado 400 (solicitud incorrecta) o un código de estado 500 (error interno del servidor).

En el ejemplo anterior, se utiliza **response.ok** para determinar si la solicitud de agregar un producto fue exitosa. Si **response.ok** es true, se muestra el mensaje "Producto agregado correctamente". De lo contrario, si **response.ok** es false, se muestra el mensaje "Error al agregar el producto".

Resumiendo, es una forma conveniente de verificar el estado de la respuesta HTTP sin tener que analizar el código de estado directamente.

### **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):
    sql = f'SELECT * FROM productos WHERE codigo = {codigo};'
    self.cursor.execute(sql)
    row = self.cursor.fetchone()
```

```
if row:
    codigo, descripcion, cantidad, precio = row
    return Producto(codigo, descripcion, cantidad, precio)
return None
```

Descripción del método consultar\_producto paso a paso:

- **sql = f'SELECT \* FROM productos WHERE codigo = {codigo};'** Genera un string a través de f-string que será la sentencia de sql de selección. El valor de codigo se interpola dentro del string.
- **self.cursor.execute(sql):** Ejecuta una consulta SQL utilizando el cursor para seleccionar todas las columnas de la tabla "productos" donde el código coincide con el valor proporcionado.
- **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 luego se 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 *None*.

### Método modificar\_producto

Este método 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)
        sql = f'UPDATE productos SET descripcion = "{nueva_descripcion}",
cantidad = {nueva_cantidad}, precio = {nuevo_precio} WHERE codigo =
{codigo};'
        self.cursor.execute(sql)
        self.conexion.commit()
        return jsonify({'message': 'Producto modificado
correctamente.'}), 200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

El método modificar\_producto() verifica la existencia del producto en el inventario, lo modifica tanto en memoria como en la base de datos y devuelve una respuesta JSON con el resultado de la operación.

### Método listar\_productos

El método listar\_productos recupera todos los productos de la base de datos.

```
def listar_productos(self):
    self.cursor.execute("SELECT * FROM productos")
    rows = self.cursor.fetchall()
    productos = []
    for row in rows:
        codigo, descripcion, cantidad, precio = row
        producto = {'codigo': codigo, 'descripcion': descripcion,
'cantidad': cantidad, 'precio': precio}
        productos.append(producto)
    return jsonify(productos), 200
```

- No recibe ningún parámetro, ya que simplemente obtiene los productos existentes en el inventario.
- Ejecuta la consulta SQL **"SELECT \* FROM productos"** para obtener todos los registros de la tabla productos en la base de datos.
- Utiliza el método **fetchall()** para obtener todas las filas resultantes de la consulta. Itera sobre cada fila y extrae los valores de codigo, descripcion, cantidad y precio.
- Crea un diccionario producto con los valores extraídos y lo agrega a la lista productos.
- Finalmente, devuelve una respuesta JSON utilizando **jsonify(productos)**, donde la lista de productos se convierte en un objeto JSON. El código de estado HTTP 200 se utiliza para indicar una respuesta exitosa.

En resumen, el método listar\_productos() obtiene todos los productos almacenados en el inventario, los transforma en una lista de diccionarios y devuelve esa lista como una respuesta JSON junto con un código de estado HTTP 200.

### Método eliminar\_producto

El método eliminar\_producto() se utiliza para eliminar un producto del inventario. Recibe el código de un producto, lo elimina de la base de datos si existe y devuelve una respuesta JSON con un mensaje indicando el resultado de la operación y el código de estado HTTP correspondiente.

```
def eliminar_producto(self, codigo):
    sql = f'DELETE FROM productos WHERE codigo = {codigo};'
    self.cursor.execute(sql)
    if self.cursor.rowcount > 0:
        self.conexion.commit()
        return jsonify({'message': 'Producto eliminado correctamente.'}),
200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

- Recibe un parámetro codigo que representa el código del producto que se desea eliminar.
- Ejecuta la consulta SQL para eliminar el registro correspondiente al código proporcionado.
- Verifica si se ha eliminado algún registro utilizando la propiedad rowcount del cursor. Si se ha eliminado al menos un registro se realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit().
- En cualquiera de los dos casos (producto encontrado o no) se devuelve una respuesta JSON utilizando jsonify() con un diccionario que contiene el mensaje correspondiente ("Producto

eliminado" o "Producto no encontrado") y el código de estado HTTP apropiado: 200 si el producto fue eliminado exitosamente o 404 si el producto no se encontró.

---

### 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, crea un cursor e inicializa el atributo items como una lista vacía.

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

#### Método agregar

El método agregar(self, codigo, cantidad, inventario) de la clase Carrito se utiliza para agregar un producto al carrito de compras. Recibe el código y la cantidad de un producto, verifica su disponibilidad en el inventario, actualiza la cantidad en el carrito y en el inventario, y devuelve un valor booleano para indicar si el producto se ha agregado correctamente al carrito o no.

```
def agregar(self, codigo, cantidad, inventario):
    producto = inventario.consultar_producto(codigo)
    if producto is None:
        return jsonify({'message': 'El producto no existe.'}), 404
    if producto.cantidad < cantidad:
        return jsonify({'message': 'Cantidad en stock insuficiente.'}),
400

    for item in self.items:
        if item.codigo == codigo:
            item.cantidad += cantidad
            sql = f'UPDATE productos SET cantidad = cantidad -
{cantidad} WHERE codigo = {codigo};'
            self.cursor.execute(sql)
            self.conexion.commit()
            return jsonify({'message': 'Producto agregado al carrito
correctamente.'}), 200

    nuevo_item = Producto(codigo, producto.descripcion, cantidad,
producto.precio)
    self.items.append(nuevo_item)
    sql = f'UPDATE productos SET cantidad = cantidad - {cantidad} WHERE
codigo = {codigo};'
    self.cursor.execute(sql)
    self.conexion.commit()
    return jsonify({'message': 'Producto agregado al carrito
correctamente.'}), 200
```

- Recibe tres parámetros: `codigo`, que representa el código del producto a agregar, `cantidad`, que indica la cantidad de unidades del producto a agregar, e `inventario`, que es una instancia de la clase `Inventario` utilizada para consultar los productos.
- Llama al método `consultar_producto(codigo)` del `inventario` para verificar si el producto existe en el inventario.
- Si el producto no existe, muestra el mensaje "El producto no existe" y devuelve el código de estado 404 para indicar que no se pudo agregar el producto al carrito.
- Si el producto existe, verifica si la cantidad solicitada está disponible en el inventario. Si la cantidad en el inventario es menor a la cantidad solicitada, devuelve el código de estado 400 para indicar que no se pudo agregar el producto al carrito.
- Recorre los elementos del carrito de compras (`self.items`) para verificar si el producto ya está en el carrito. Dado que el producto ya está en el carrito, actualiza la cantidad del producto sumando la cantidad solicitada a la cantidad existente.
- Realiza una consulta SQL para actualizar la cantidad en el inventario, restando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante `self.conexion.commit()`. Devuelve el código de estado 200 para indicar que el producto se ha agregado exitosamente al carrito.
- Si el producto no está en el carrito, crea una nueva instancia de la clase `Producto` con los datos del producto obtenidos del inventario.
- Añade el nuevo producto al carrito (`self.items`).
- Realiza una consulta SQL para actualizar la cantidad en el inventario, restando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante `self.conexion.commit()`. Devuelve el código de estado 200 para indicar que el producto se ha agregado exitosamente al carrito.

### *Método quitar*

El método `quitar(self, codigo, cantidad, inventario)` de la clase `Carrito` se utiliza para quitar un producto del carrito de compras. Recibe el código y la cantidad de un producto, busca el producto en el carrito, actualiza la cantidad en el carrito y en el inventario, y devuelve un valor booleano para indicar si el producto se ha quitado correctamente del carrito o no. Además, maneja la verificación de la cantidad solicitada para quitar, evitando que se quiten más unidades de las disponibles en el carrito.

```
def quitar(self, codigo, cantidad):
    for item in self.items:
        if item.codigo == codigo:
            if cantidad > item.cantidad:
                return jsonify({'message': 'Cantidad a quitar mayor a la cantidad en el carrito.'}), 400
            item.cantidad -= cantidad
            if item.cantidad == 0:
                self.items.remove(item)
            sql = f'UPDATE productos SET cantidad = cantidad + {cantidad} WHERE codigo = {codigo};'
            self.cursor.execute(sql)
            self.conexion.commit()
            return jsonify({'message': 'Producto quitado del carrito correctamente.'}), 200
```

```
return jsonify({'message': 'El producto no se encuentra en el  
carrito.'}), 404
```

- Recibe tres parámetros: código, que representa el código del producto a quitar, cantidad, que indica la cantidad de unidades del producto a quitar, e inventario, que es una instancia de la clase Inventario utilizada para consultar los productos.
- Recorre los elementos del carrito de compras (self.items) para buscar el producto que se desea quitar.
- Si encuentra el producto en el carrito, verifica si la cantidad solicitada para quitar es mayor a la cantidad actual en el carrito. Si es así devuelve el código de estado 400 para indicar que la cantidad a quitar es mayor a la cantidad en el carrito.
- Si la cantidad a quitar es la adecuada actualiza la cantidad del producto en el carrito restando la cantidad solicitada.
- Realiza una consulta SQL para actualizar la cantidad en el inventario, sumando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit(). Devuelve el código de estado 200 para indicar que el producto se ha quitado exitosamente del carrito.
- Si no encuentra el producto en el carrito devuelve el código de estado 404 para indicar que el producto no se encuentra en el carrito.

### *Método mostrar*

El método mostrar(self) de la clase Carrito se utiliza para mostrar el contenido del carrito de compras. Transforma los productos en el carrito en un formato JSON estructurado y los devuelve como respuesta HTTP junto con el código 200, lo que permite visualizar el contenido del carrito en el frontend de manera legible y fácil de procesar.

```
def mostrar(self):  
    productos_carrito = []  
    for item in self.items:  
        producto = {'codigo': item.codigo, 'descripcion':  
item.descripcion, 'cantidad': item.cantidad, 'precio': item.precio}  
        productos_carrito.append(producto)  
    return jsonify(productos_carrito), 200
```

- No recibe ningún parámetro.
- Crea una lista vacía llamada productos\_carrito para almacenar los productos del carrito en formato JSON. Recorre cada elemento del carrito de compras (self.items).
- Para cada producto en el carrito, crea un diccionario con las claves "codigo", "descripcion", "cantidad" y "precio" que representan los atributos del producto.
- Agrega el diccionario del producto a la lista productos\_carrito.
- Retorna una respuesta JSON utilizando la función jsonify de Flask, que convierte la lista de productos (productos\_carrito) en formato JSON.
- Además, se devuelve el código de estado HTTP 200, indicando que la solicitud se ha procesado correctamente.

---

### **Ejecución del código**

---

El código completo que hemos elaborado está estructurado de la siguiente manera:

- Importamos las bibliotecas necesarias, incluyendo Flask y otros módulos relacionados.

- Creamos una instancia de la clase Flask y la asignamos a la variable `app`. Esta instancia representa nuestra aplicación Flask.
- Definimos las rutas y los métodos asociados a cada ruta utilizando los decoradores `@app.route()`. Estos decoradores especifican la URL y los métodos HTTP permitidos para cada ruta.
- Implementamos las funciones correspondientes a cada ruta. Estas funciones se ejecutan cuando se accede a las rutas específicas de la API.
- Dentro de estas funciones, realizamos las operaciones necesarias, como agregar, modificar o eliminar productos del inventario o del carrito. Utilizamos las clases Carrito e Inventario que hemos definido para realizar estas operaciones.
- Utilizamos los métodos `jsonify()` de Flask para convertir los datos en formato JSON y devolverlos como respuesta a las solicitudes.

Finalmente, fuera de las funciones de ruta, agregamos las siguientes líneas de código:

```
# Finalmente, si estamos ejecutando este archivo, lanzamos app.
if __name__ == '__main__':
    app.run()
```

Estas líneas se encargan de iniciar el servidor de Flask. El condicional `if name == 'main':` asegura que el servidor solo se ejecute si el archivo se ejecuta directamente (no cuando se importa como módulo).

Cuando ejecutamos el archivo, Flask se pone en marcha y escucha las solicitudes entrantes en el servidor local. El servidor se inicia en la dirección **`http://localhost:5000/`**, donde `localhost` es la dirección IP del servidor local y `5000` es el número de puerto por defecto utilizado por Flask. Puedes acceder a la API a través de esta URL en tu navegador o desde el frontend para realizar las solicitudes HTTP y obtener las respuestas correspondientes.

En la terminal veremos algo similar a esto:

```
* Serving Flask app 'etapa3 flask'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Con esto, finalizamos la implementación de la API.

Resta subir la aplicación a un servidor como PythonAnywhere, y luego implementar un frontend que utilice los servicios de esta API.

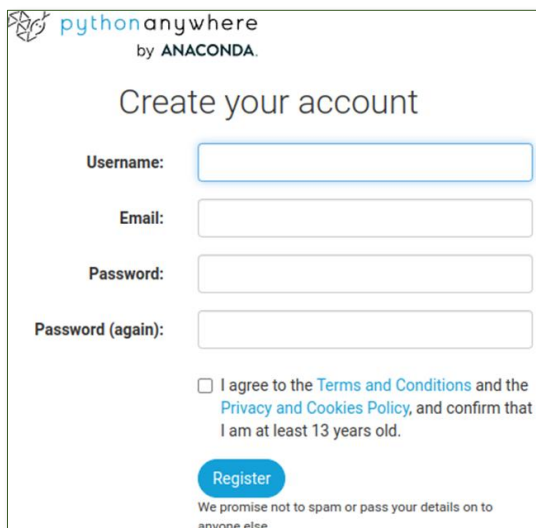
### ETAPA 4: DESPLIEGUE EN SERVIDOR PYTHONANYWHERE

PythonAnywhere es una empresa de hosting para aplicaciones web escritas en Python. Al crear un usuario, conseguimos de forma gratuita una especie de máquina virtual Linux con varios intérpretes de Python instalados, múltiples módulos y paquetes de terceros y la capacidad de instalar nuevos vía pip, una base de datos MySQL lista para usar, un servidor web plenamente configurado, un sistema de archivos con 512 MB de capacidad y muchas otras cosas interesantes. No solamente se trata de una solución ideal para llevar por primera vez una aplicación web a producción, sino también para proyectos profesionales. PythonAnywhere permite seleccionar únicamente los recursos que queremos usar y pagar en consecuencia; y, en los planes pagos, configurar nuestras aplicaciones con un dominio propio.



### Registro y configuración

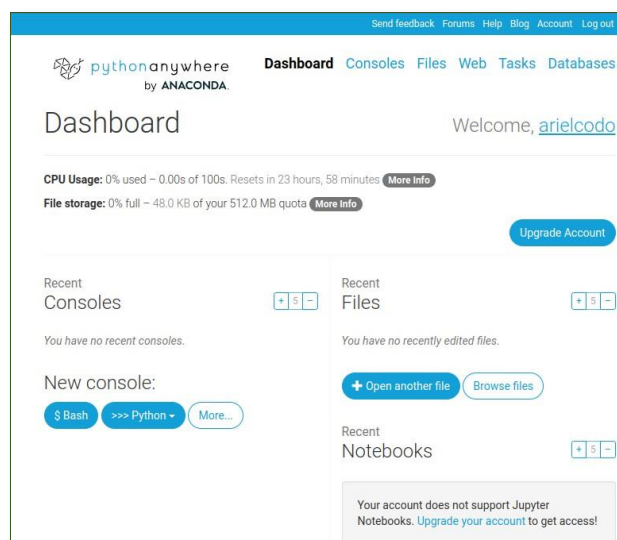
Para comenzar a utilizar PythonAnywhere, nos dirigimos a <https://www.pythonanywhere.com/pricing/> y presionamos el botón **Create a Beginner account**:



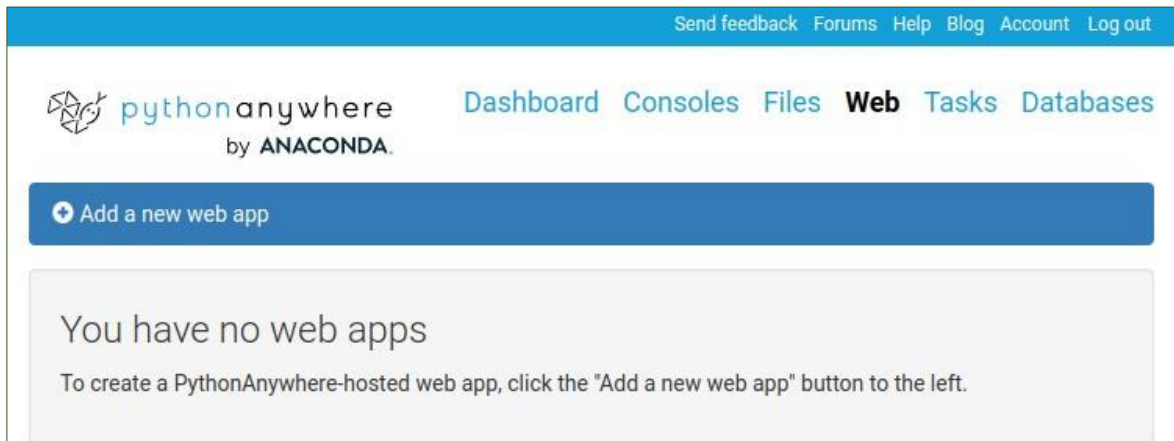
Luego nos registramos completando el formulario con un nombre de usuario, una dirección de correo electrónico y una contraseña. Esto enviará un correo electrónico a la dirección indicada para confirmar el registro.

Como siempre, nos aseguramos de tener los datos (username, pass, etc) a mano porque los vamos a necesitar luego. Esto enviará un correo electrónico a la dirección indicada para confirmar el registro.

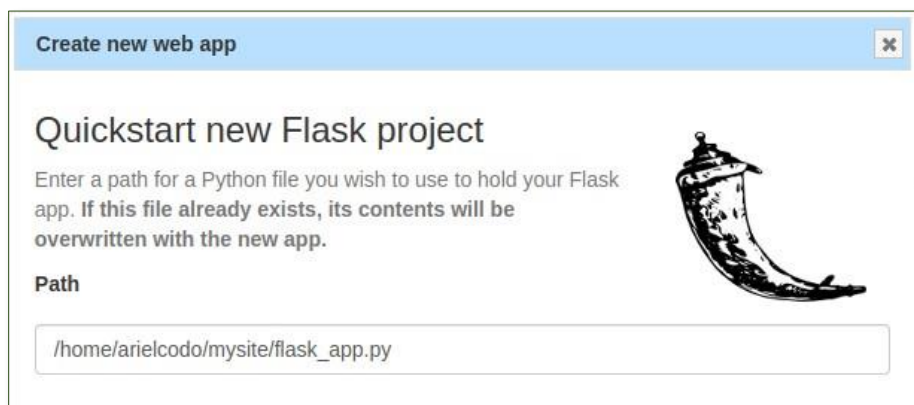
Ahora, al ingresar a <https://www.pythonanywhere.com/> deberíamos ver algo así:



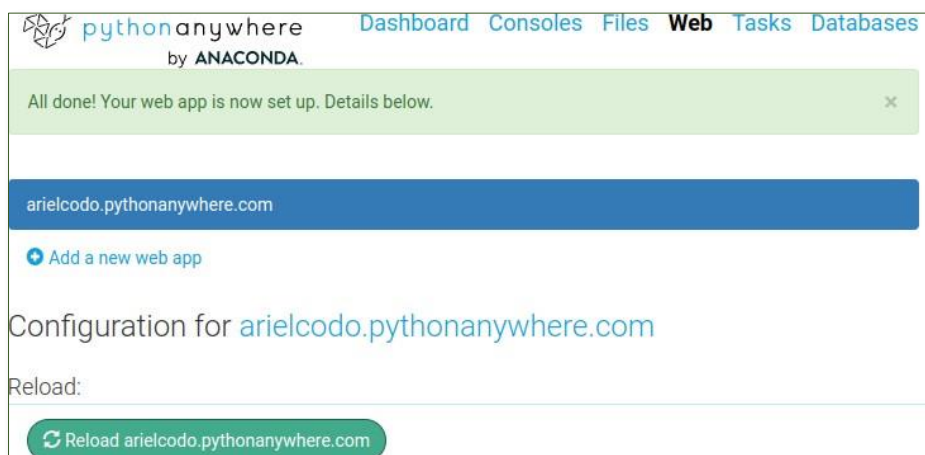
Tenemos que indicarle a PythonAnywhere que queremos subir una aplicación de Flask. En el menú superior derecho, vamos a dirigirnos a la opción Web. Una vez allí, a la izquierda, presionamos el botón "+ Add a new web app":



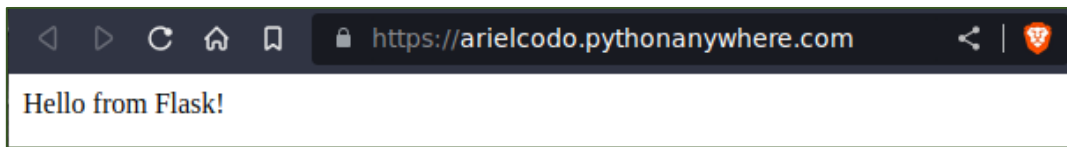
Se iniciará un asistente que nos consultará el framework que queremos usar, la versión de Python y el nombre del proyecto. Completaremos esos datos con las opciones «Flask», «Python 3.10» (o la versión que estés utilizando) y «mysite» (o el nombre de la carpeta que contenga tu archivo `app_flask.py`), como se ilustra a continuación:



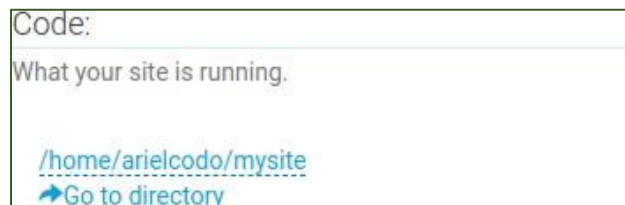
En este punto nos encontramos con la configuración básica realizada:



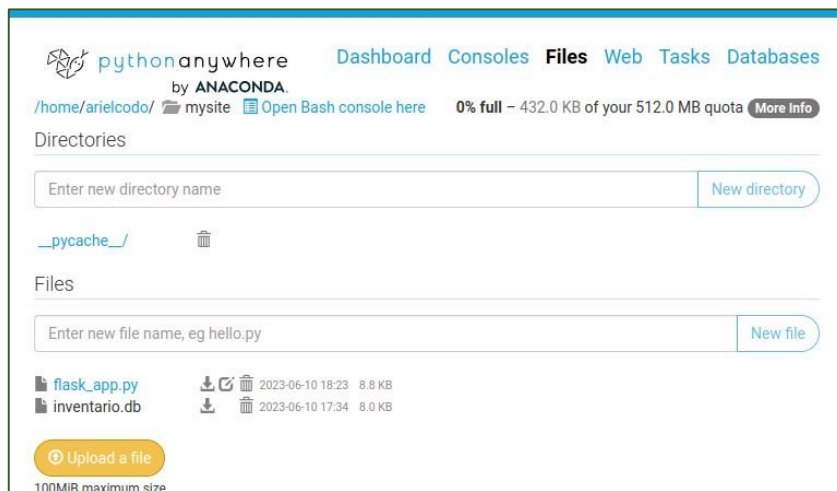
Y vemos cual es la URL de nuestro sitio. Si dirigimos nuestro navegador a esa dirección, vemos:



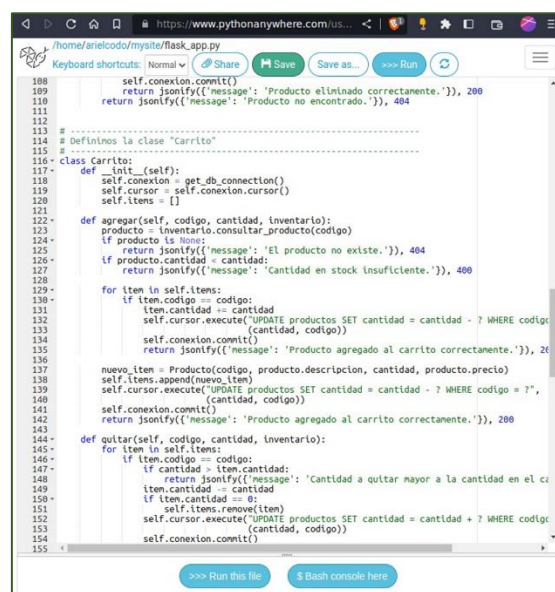
Ahora necesitamos cambiar la flask\_app.py que el sitio ha puesto por defecto por nuestro código. Hay muchas maneras de hacer eso, una sencilla es simplemente editar flask\_app.py y copiar encima nuestro código. Para ello, bajamos y hacemos click en el siguiente enlace:



En la pantalla siguiente vemos los archivos que se encuentran en nuestro directorio, en "Files":



Hacemos click en **flask\_app.py** para editarlo, y copiamos nuestro código:



Antes de salir, guardamos (Save) y recargamos el sitio con el último botón de la barra:



Con esto tendríamos lista nuestra API, corriendo en la dirección web (URL) que nos ha proporcionado Pythonanywhere. En este texto, es <https://arielcodo.pythonanywhere.com>. Podemos probar si funciona escribiendo en el navegador <https://arielcodo.pythonanywhere.com/productos> y deberíamos recibir como respuesta un arreglo vacío ( [] ), ya que no hemos cargado aún productos.

---

### Prueba rápida de la API

---

Podemos hacer una prueba rápida usando este archivo HTML. Lo ejecutamos en nuestra computadora con liveServer, completamos el formulario y enviamos su contenido:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Agregar producto</title>
</head>

<body>
  <h1>Agregar Producto al Inventario</h1>
  <form id="formulario">
    <label for="codigo">Código:</label>
    <input type="text" id="codigo" name="codigo" required><br>

    <label for="descripcion">Descripción:</label>
    <input type="text" id="descripcion" name="descripcion" required><br>

    <label for="cantidad">Cantidad:</label>
    <input type="number" id="cantidad" name="cantidad" required><br>

    <label for="precio">Precio:</label>
    <input type="number" step="0.01" id="precio" name="precio"
required><br>

    <button type="submit">Agregar Producto</button>
  </form>

  <script>
    // Capturamos el evento de envío del formulario
    document.getElementById('formulario').addEventListener('submit',
function (event) {
      event.preventDefault() // Evitamos que se recargue la página

      // Obtenemos los valores del formulario
```

```
var codigo = document.getElementById('codigo').value
var descripcion = document.getElementById('descripcion').value
var cantidad = document.getElementById('cantidad').value
var precio = document.getElementById('precio').value

// Creamos un objeto con los datos del producto
var producto = {
  codigo: codigo,
  descripcion: descripcion,
  cantidad: cantidad,
  precio: precio
}
console.log(producto)
// Realizamos la solicitud POST al servidor
url = 'https://juanpablocodo.pythonanywhere.com/productos'
fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(producto)
})
.then(function (response) {
  if (response.ok) {
    return response.json() // Parseamos la respuesta JSON
  } else {
    throw new Error('Error al agregar el producto.')
  }
})
.then(function (data) {
  alert('Producto agregado correctamente.')
})
.catch(function (error) {
  console.log('Error:', error)
  alert('Error al agregar el producto.')
})
})
</script>
</body>
</html>
```

**Importante:** Recuerda cambiar la URL <https://xxxxxx.pythonanywhere.com/productos> por la de TU implementación.

Si todo ha funcionado, al apuntar el navegador a

<https://tuusuario.pythonanywhere.com/productos> nuevamente, en lugar del arreglo vacío deberías ver algo como esto:

```
[{"cantidad":12,"codigo":1,"descripcion":"Mouse","precio":3222.0},
{"cantidad":7,"codigo":2,"descripcion":"Teclado","precio":4500.0}]
```

Es decir, un arreglo con los datos de los productos de la base de datos. Pero puede que no haya funcionado. Veamos por qué.

---

### CORS

---

Es probable que al intentar ingresar un producto en el inventario se obtenga un error similar a este:



```
✖ Access to fetch at 'https://www.pythonanywhere.com/p_index.html:1
roductos' from origin 'http://127.0.0.1:5500' has been blocked by
CORS policy: Response to preflight request doesn't pass access
control check: No 'Access-Control-Allow-Origin' header is present
on the requested resource. If an opaque response serves your
needs, set the request's mode to 'no-cors' to fetch the resource
with CORS disabled.
```

El error se debe a una política de seguridad denominada **Same-Origin Policy** (Política del mismo origen) implementada por los navegadores web. Esta política restringe las solicitudes HTTP realizadas desde un origen (dominio, protocolo y puerto) a otro origen diferente.

Ocurre, por ejemplo, se estás realizando una solicitud desde el origen <http://127.0.0.1:5500> a <https://www.pythonanywhere.com>, lo cual no cumple con la política de mismo origen y, por lo tanto, se bloquea.

Para solucionar este problema, necesitas habilitar el **intercambio de recursos de origen cruzado** (*Cross-Origin Resource Sharing, CORS*) en el servidor de PythonAnywhere para permitir que tu sitio web acceda a la API.

En el código de Flask, debes agregar los encabezados de respuesta adecuados que permitan las solicitudes desde el origen deseado. Puedes hacerlo utilizando la extensión Flask-CORS, que debemos instalar así:

- 1) Instalar Flask-CORS en el entorno de PythonAnywhere. Para ello, abrimos una terminal bash desde el dashboard de PythonAnywhere (Dashborad - \$ bash) y en ella ejecutamos:

```
pip install flask-cors
```

- 2) Importar y configurar Flask-CORS en la aplicación Flask, modificando nuestro código Python para que se agregue al principio la línea **from flask\_cors import CORS**:

```
from flask import Flask, jsonify, request
from flask_cors import CORS
```

Y cerca del final, luego de crear la app Flask agregamos CORS(app) :

```
app = Flask(__name__)
CORS(app)
```

Con estos cambios, el error desaparece. Ahora solo queda implementar el frontend.

## ETAPA 5: CODIFICACIÓN DEL FRONT-END

En esta etapa veremos una serie de códigos mínimos que nos ayuden a desarrollar un frontend que utilice las funciones de la API desarrollada.

Haremos hincapié en los temas relacionados con la API en sí, y no tanto en el apartado CSS o HTML.

Usaremos Vue3 para simplificar el código.

---

### Menú principal

---

El archivo de entrada al frontend es simplemente un menú con enlaces que llevan a cada una de las secciones implementadas:

**index.html:**

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo de uso de la API</title>
  <link rel="stylesheet" href="css/estilos.css">
</head>

<body>
  <div>
    <h1>Ejemplos de uso de la API</h1>
    <h3>Codo a Codo 2023</h3>
    <table>
      <tr><td class="contenedor-centrado"><a href="altas.html">Alta de
productos</a></td></tr>
      <tr><td class="contenedor-centrado "><a
href="listado.html">Listado de productos</a></td></tr>
      <tr><td class="contenedor-centrado "><a
href="modificaciones.html">Modificar datos de productos</a></td></tr>
      <tr><td class="contenedor-centrado "><a
href="listadoEliminar.html">Eliminar productos</a></td></tr>
      <tr><td class="contenedor-centrado "><a
href="carrito1.html">Carrito de compras</a></td></tr>
    </table>
  </div>
</body>

</html>
```

---

### Alta de productos

---

**altas.html:**

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Agregar producto</title>
```



```
<link rel="stylesheet" href="css/estilos.css">
</head>

<body>
  <h1>Agregar Productos al Inventario</h1>
  <h3>Codo a Codo 2023</h3>
  <form id="formulario">
    <label for="codigo">Código:</label>
    <input type="text" id="codigo" name="codigo" required><br>

    <label for="descripcion">Descripción:</label>
    <input type="text" id="descripcion" name="descripcion" required><br>

    <label for="cantidad">Cantidad:</label>
    <input type="number" id="cantidad" name="cantidad" required><br>

    <label for="precio">Precio:</label>
    <input type="number" step="0.01" id="precio" name="precio"
required><br>

    <button type="submit">Agregar Producto</button>
    <a href="index.html">Menu principal</a>
  </form>
  <script>
    const URL = "https://tuusuario.pythonanywhere.com/"
    // Capturamos el evento de envío del formulario
    document.getElementById('formulario').addEventListener('submit',
function (event) {
      event.preventDefault() // Evitamos que se recargue la página

      // Obtenemos los valores del formulario
      var codigo = document.getElementById('codigo').value
      var descripcion = document.getElementById('descripcion').value
      var cantidad = document.getElementById('cantidad').value
      var precio = document.getElementById('precio').value

      // Creamos un objeto con los datos del producto
      var producto = {
        codigo: codigo,
        descripcion: descripcion,
        cantidad: cantidad,
        precio: precio
      }
      console.log(producto)
      // Realizamos la solicitud POST al servidor
      fetch(URL + 'productos', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(producto)
      })
    })
  </script>
</body>
```



```
        .then(function (response) {
            if (response.ok) {
                return response.json() // Parseamos la respuesta JSON
            } else {
                throw new Error('Error al agregar el producto.')
            }
        })
        .then(function (data) {
            alert('Producto agregado correctamente.')
            document.getElementById('codigo').value = ""
            document.getElementById('descripcion').value = ""
            document.getElementById('cantidad').value = ""
            document.getElementById('precio').value = ""
        })
        .catch(function (error) {
            console.log('Error:', error)
            alert('Error al agregar el producto.')
        })
    })
</script>
</body>
</html>
```

Descripción del script utilizado:

Se comienza declarando una constante:

```
const URL = "https://tuusuario.pythonanywhere.com/"
```

**Importante:** Recuerda reemplazar esta línea con tu usuario de Pythonanywhere.

Esta constante almacena la URL del servidor al que se enviarán las solicitudes POST. Puede ser una dirección local (como "http://127.0.0.1:5000/") o una dirección remota (como en este caso). Se puede cambiar la URL dependiendo de las necesidades del proyecto.

A continuación, se utiliza el método **addEventListener** para agregar un evento de escucha al formulario con el identificador "formulario" cuando se envíe:

```
document.getElementById('formulario').addEventListener('submit', function
(event) {
    // Código de evento...
})
```

Esto permite capturar el evento de envío del formulario y ejecutar el código correspondiente cuando se envía el formulario.

**event.preventDefault()** se utiliza para evitar que la página se recargue cuando se envía el formulario. Esto se logra llamando al método preventDefault() en el objeto event pasado como parámetro en la función del evento.

A continuación, se obtienen los valores de los campos de entrada de texto y se almacenan en variables:

```
// Obtenemos los valores del formulario
var codigo = document.getElementById('codigo').value
```

```
var descripcion = document.getElementById('descripcion').value
var cantidad = document.getElementById('cantidad').value
var precio = document.getElementById('precio').value
```

Esto se logra utilizando el método `getElementById()` para obtener los elementos del DOM correspondientes a los campos de entrada y accediendo a su propiedad `value` para obtener el valor ingresado por el usuario.

Luego, se crea un objeto producto con los datos obtenidos del formulario:

```
// Creamos un objeto con los datos del producto
var producto = {
  codigo: codigo,
  descripcion: descripcion,
  cantidad: cantidad,
  precio: precio
}
```

Este objeto producto contiene las propiedades `codigo`, `descripcion`, `cantidad` y `precio`, con sus respectivos valores obtenidos del formulario.

Se realiza una solicitud POST al servidor utilizando el método `fetch()`:

```
fetch(URL + 'productos', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(producto)
})
.then(function (response) {
  if (response.ok) {
    return response.json() // Parseamos la respuesta JSON
  } else {
    throw new Error('Error al agregar el producto.')
  }
})
.then(function (data) {
  alert('Producto agregado correctamente.')
  document.getElementById('codigo').value = ""
  document.getElementById('descripcion').value = ""
  document.getElementById('cantidad').value = ""
  document.getElementById('precio').value = ""
})
.catch(function (error) {
  console.log('Error:', error)
  alert('Error al agregar el producto.')
})
```

El URL utilizado en la solicitud POST se construye concatenando la constante URL con la ruta específica, en este caso, 'productos'. Se especifica que el método de la solicitud es POST, se establece el encabezado 'Content-Type': 'application/json' para indicar que se está enviando datos en formato JSON, y se convierte el objeto producto a JSON utilizando `JSON.stringify()` antes de enviarlo como cuerpo de la solicitud.

Después de realizar la solicitud POST, se utiliza el método `then()` para manejar la respuesta del servidor. En este caso, se verifica si la respuesta del servidor es exitosa (`response.ok`). Si es así, se llama al método `json()` en la respuesta para obtener los datos de la respuesta en formato JSON. Si la respuesta no es exitosa, se lanza un error.

Finalmente, se utiliza otro `then()` para manejar los datos de la respuesta.

En este caso, se muestra una alerta indicando que el producto se agregó correctamente y se restablecen los valores de los campos del formulario a través de la manipulación del DOM.

En caso de producirse algún error durante el proceso, se captura en el bloque `catch()` y se muestra una alerta de error.

Este script maneja el evento de envío del formulario, obtiene los valores ingresados por el usuario, realiza una solicitud POST al servidor con los datos del producto y maneja la respuesta del servidor.

---

### Hoja de estilos

---

Todos los archivos HTML mostrados utilizan el mismo código CSS (`estilos.css`):

```
/* Estilos para todo el proyecto */
body {
  font-family: Arial, sans-serif;
  background-color: #f9f9f9;
}

.contenedor-centrado {
  display: flex;
  width: 100%;
  justify-content: center;
}

p {
  font-family: 'Times New Roman', Times, serif;
  background-color: #f9f9f9;
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
}

h1, h2, h3, h4, h5, h6 {
  text-align: center;
  color: #007bff;
}

form {
  max-width: 400px;
  margin: 0 auto;
  padding: 20px;
  background-color: white;
  border: 1px solid lightslategray;
  border-radius: 5px;
}
```

```
table {
  max-width: 90%;
  margin: 0 auto;
  padding: 20px;
  background-color: white;
  border: 1px solid lightslategray;
  border-radius: 5px;
}

label {
  display: block;
  margin-bottom: 5px;
}

input[type="text"],
input[type="number"],
textarea {
  width: 90%;
  padding: 10px;
  margin-bottom: 10px;
  border: 1px solid #cccccc;
  border-radius: 5px;
}

input[type="submit"] {
  padding: 10px;
  background-color: #007bff;
  color: #ffffff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

input[type="submit"]:hover {
  background-color: #0056b3;
}

button {
  padding: 8px;
  margin: 4px;
  background-color: #007bff;
  color: #ffffff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

button:hover {
  background-color: #0056b3;
}

a {
```

```
padding: 8px;
margin: 4px;
background-color: #fdab13;
color: #ffffff;
border: none;
border-radius: 5px;
cursor: pointer;
text-decoration: none;
font-size: 85%;
}

a:hover {
    background-color: #cd7b00;
}
```

Breve descripción de cada estilo implementado:

- **body:** Aplica estilos al elemento <body> de la página. Establece la fuente del texto como Arial o una fuente genérica sin serifa (sans-serif) y establece el color de fondo como #f9f9f9.
- **.contenedor-centrado:** Aplica estilos a un contenedor específico utilizando la clase .contenedor-centrado. Hace uso de flexbox para centrar horizontalmente su contenido y establece su ancho como el 100% del contenedor padre.
- **p:** Aplica estilos a los elementos <p> (párrafos). Establece la fuente del texto como 'Times New Roman', Times, serif, establece el color de fondo como #f9f9f9, limita el ancho máximo a 400px y centra el elemento horizontalmente utilizando los márgenes automáticos. Además, se añade un relleno interno de 20px.
- **h1, h2, h3, h4, h5, h6:** Aplica estilos a los elementos de encabezado del nivel 1 al 6 (<h1> a <h6>). Los alinea al centro y establece el color del texto como #007bff (un tono de azul).
- **form:** Aplica estilos a los elementos <form> (formularios). Establece el ancho máximo a 400px, centra el formulario horizontalmente utilizando los márgenes automáticos y añade un relleno interno de 20px. Además, establece el color de fondo como blanco, agrega un borde de 1px sólido en lightslategray y un borde redondeado de 5px.
- **table:** Aplica estilos a los elementos <table> (tablas). Establece el ancho máximo al 90%, centra la tabla horizontalmente utilizando los márgenes automáticos y añade un relleno interno de 20px. Además, establece el color de fondo como blanco, agrega un borde de 1px sólido en lightslategray y un borde redondeado de 5px.
- **label:** Aplica estilos a los elementos <label> (etiquetas). Los muestra como bloques y agrega un margen inferior de 5px.
- **input[type="text"], input[type="number"], textarea:** Aplica estilos a los elementos de entrada de tipo texto, número y área de texto (<input type="text">, <input type="number"> y <textarea>). Establece el ancho al 90%, agrega un relleno de 10px, un margen inferior de 10px, un borde de 1px sólido en #cccccc y un borde redondeado de 5px.
- **input[type="submit"]:** Aplica estilos al botón de envío (<input type="submit">). Agrega un relleno de 10px, establece el color de fondo como #007bff, el color del texto como blanco, elimina el borde y añade un borde redondeado de 5px. Además, cambia el cursor del mouse a un puntero cuando se pasa sobre el botón.
- **input[type="submit"]:hover:** Aplica estilos al botón de envío cuando se pasa el cursor sobre él. Cambia el color de fondo a #0056b3.
- **button:** Aplica estilos a los elementos <button> (botones). Agrega un relleno de 8px, un margen de 4px, establece el color de fondo como #007bff, el color del texto como blanco,

elimina el borde y añade un borde redondeado de 5px. Además, cambia el cursor del mouse a un puntero cuando se pasa sobre el botón.

- **button:hover:** Aplica estilos a los botones cuando se pasa el cursor sobre ellos. Cambia el color de fondo a #0056b3.
- **a:** Aplica estilos a los elementos <a> (enlaces). Agrega un relleno de 8px, un margen de 4px, establece el color de fondo como #fdab13, el color del texto como blanco, elimina el borde y añade un borde redondeado de 5px. Además, cambia el cursor del mouse a un puntero cuando se pasa sobre el enlace y establece el tamaño de fuente al 85%.
- **a:hover:** Aplica estilos a los enlaces cuando se pasa el cursor sobre ellos. Cambia el color de fondo a #cd7b00.