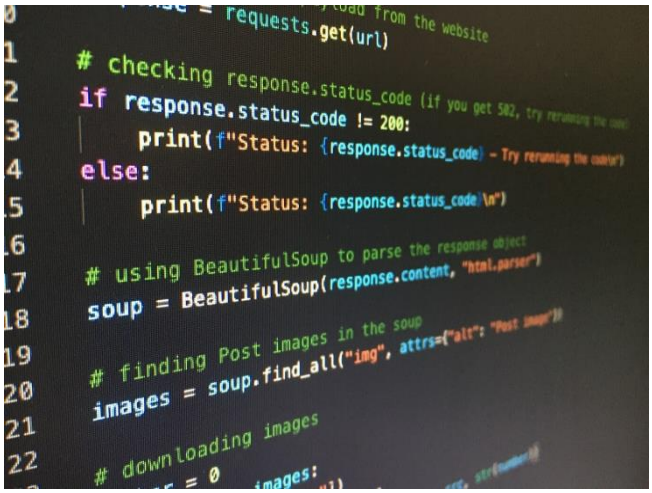

PROYECTO CRUD

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

Agencia de Aprendizaje a lo largo de la vida

INDICE

PROYECTO CRUD - TPO FINAL

HTML - CSS - JAVASCRIPT - VUE

BASES DE DATOS - PYTHON - FLASK

DESCRIPCIÓN GENERAL DEL PROYECTO

El proyecto se trata de un sistema que permite administrar una base de datos de productos, implementado una API en Python utilizando el framework Flask. Las operaciones que se realizarán en este proyecto es dar de alta, modificar, eliminar y listar los productos, operaciones que podrán hacer los usuarios a través de una página Web.
Como ejemplo trabajaremos con una empresa de venta de **artículos de computación** que ofrece sus productos a través de una Web.

Aquí hay un resumen de las principales características y funcionalidades del proyecto:

Gestión de productos:

- Agregar un nuevo producto al catálogo.
- Modificar la información de un producto existente en el catálogo.
- Eliminar un producto del catálogo.
- Consultar información de un producto por su código.

Persistencia de datos:

- Los datos de los productos se almacenan en una base de datos SQL.
- Se utiliza una conexión a la base de datos para realizar operaciones CRUD en los productos.
- El código proporcionado incluye las clases Catálogo, que representa la estructura y funcionalidad relacionada con el catálogo de productos. Además, se define una serie de rutas en Flask para manejar las solicitudes HTTP relacionadas con la gestión de productos.

Se implementan desde cero el backend y el frontend. En el caso del backend, el proyecto va "evolucionando", comenzando en el desarrollo de las funciones que se necesitan para manipular los productos utilizando arreglos en memoria, luego se modifica para utilizar objetos, más tarde se gestiona la persistencia de los datos utilizando una base de datos SQL, después se implementa la API en Flask y se aloja el script Python en un servidor, y por último se crea un frontend básico para interactuar con los datos desde el navegador, a través dela API creada.

El proyecto se divide en seis etapas:

1) **Desarrollo de arreglos y funciones:** Implementar un CRUD de productos utilizando arreglos y funciones.
2) **Conversión a clases y objetos:** Convertir las funciones vistas en objetos y clases.
3) **Creación de la base de datos SQL:** Utilizar como almacenamiento una base de datos SQL.
4) **Implementación de la API en Flask:** A través de este framework implementar una API que permita ser consumida desde el front.
5) **Codificación del Front-End:** Vistas que permitan realizar las operaciones del CRUD.
6) **Despliegue en servidor PythonAnywhere:** Hosting para aplicaciones web escritas en Python.

ETAPA 1: DESARROLLO DE ARREGLOS Y FUNCIONES

Objetivo: Implementar un CRUD de productos utilizando arreglos y funciones.

Funciones para gestionar un arreglo con datos de productos

Escribimos una serie de funciones para crear una pequeña app que maneje un arreglo que contenga diccionarios con los datos de los productos.
Los diccionarios tienen las siguientes claves:

- **codigo:** un número entero que sirve como identificación única para cada producto.
- **descripcion:** una cadena de texto que describe el producto, como su nombre o modelo.
- **cantidad:** otro número entero que indica cuántas unidades de este producto están disponibles en el inventario.
- **precio:** un número decimal que representa el precio de venta del producto.
- **imagen:** un texto que contiene el nombre de la imagen relacionada con el producto (esto podría ser útil en una aplicación de comercio electrónico).
- **proveedor:** un número entero que identifica al proveedor del producto.

Nuestras funciones harán lo siguiente:
- ✓ Agregar un producto al arreglo
- ✓ Consultar un producto a partir de su código
- ✓ Modificar los datos de un producto a partir de su código
- ✓ Obtener un listado de los productos que existen en el arreglo.
- ✓ Eliminar un producto del arreglo.

Antes de definir las funciones haremos lo siguiente:

```
# Definimos una lista de diccionarios para almacenar los productos.
productos = []
```

A continuación, explicaremos en detalle cada una de las funciones:

agregar_producto(codigo, descripcion, cantidad, precio, imagen, proveedor)

Esta función tiene la tarea de agregar un nuevo producto a una lista llamada **productos**, siempre que no exista un producto con el mismo código previamente. Esto asegura la unicidad de productos en función de su código numérico. Veamos cómo funciona:

Parámetros:
- **codigo:** int, código numérico del producto.
- **descripcion:** str, descripción alfabética del producto.
- **cantidad:** int, cantidad en stock del producto.
- **precio:** float, precio de venta del producto.
- **imagen:** str, nombre de la imagen del producto.
- **proveedor:** int, número de proveedor del producto.

Retorna:
- **Valor booleano: True** si el producto se agregó exitosamente al arreglo y **False** si ya existe un producto con el mismo código y no se agrega el nuevo producto.

```python
def agregar_producto(codigo, descripcion, cantidad, precio, imagen, proveedor):

    if consultar_producto(codigo):
        return False

    nuevo_producto = {
        'codigo': codigo,
        'descripcion': descripcion,
        'cantidad': cantidad,
        'precio': precio,
        'imagen': imagen,
        'proveedor': proveedor
    }
    productos.append(nuevo_producto)
    return True
```

Explicación del código:

1. **Verificamos su existencia:** Verificamos si ya existe un producto con el mismo código, utilizando la función **consultar_producto**. Si un producto con el mismo código se encuentra en el arreglo, se evita agregar el nuevo producto y se retorna ***False***. Si no se encuentra un producto con el mismo código, se procede a crear un diccionario que contiene los datos del nuevo producto.
2. **Creamos un Producto:** En caso de no existir el nuevo producto la función toma estos parámetros y crea un nuevo producto. Esta representación del producto es un "diccionario" de datos[1]. Las claves de este diccionario son '**codigo**', '**descripcion**', '**cantidad**', '**precio**', '**imagen**' y '**proveedor**'. Cada clave se asocia a su valor correspondiente, que se toma de los parámetros de la función.
3. **Agregamos el Producto:** Una vez que tenemos el diccionario del producto con todos los detalles, la función agrega este producto a la lista llamada **productos**, que almacena todos los productos disponibles.
4. **Valor de Retorno:** La función devuelve ***True*** como resultado. Esto es una señal para indicar que el proceso de agregar el producto se completó con éxito.

En resumen: La función verifica la existencia del producto y, en caso de no existir en la lista, toma la información pasada por parámetro, la organiza en un formato específico (el diccionario) y agrega ese producto a la lista.

consultar_producto(codigo)

Esta función se encarga de **buscar y recuperar** la información de un producto de acuerdo a su código. Esto es especialmente útil en una aplicación de gestión de inventario para encontrar detalles específicos de un producto, además de que le permite a la función **agregar_producto** verificar si un producto no fue agregado previamente a la lista, evitando así la duplicación de registros. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto. Se utiliza para identificar un producto en particular. La función buscará un producto que coincida con ese código.

[1] Un diccionario es una estructura de datos que permite almacenar varios valores asociados a claves.

Retorna:

- Si el producto fue encontrado retorna un **diccionario con los datos del producto**. Si no se encontró retorna el valor booleano ***False***.

```
def consultar_producto(codigo):
    for producto in productos:
        if producto['codigo'] == codigo:
            return producto
    return False
```

Explicación del código:

Proceso de Búsqueda con bucle: La función comienza recorriendo **productos,** la lista que almacena todos los productos disponibles en la aplicación. El bucle ***for*** examina cada producto, verificando si el valor de la clave '**codigo**' en el diccionario del producto coincide con el valor proporcionado en el parámetro codigo. Esto es fundamental para identificar el producto correcto. Este proceso de búsqueda condiuce a dos resultados:

a. **Producto Encontrado:** Si se encuentra un producto con el código coincidente, la función regresa el diccionario que representa ese producto. Esto significa que la función proporciona todos los detalles de ese producto en particular, como su descripción, cantidad en stock, precio, imagen, y proveedor. Este diccionario se puede utilizar posteriormente en el programa para mostrar o manipular la información del producto.
b. **Producto no Encontrado:** Si el bucle finaliza sin encontrar el producto deseado, la función regresa ***False***. Esto sirve como indicador de que no se ha encontrado un producto con el código proporcionado.

En resumen: Esta función es un mecanismo esencial para recuperar información detallada de un producto utilizando su código único como referencia. A través de un bucle y una comparación de códigos, la función encuentra el producto correcto y devuelve todos sus detalles en forma de un diccionario. De esta manera, los usuarios pueden acceder a la información de un producto en base a su código, lo que facilita la gestión y visualización de datos en la aplicación.

modificar_producto(codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor)

Esta función se encarga de actualizar los datos de un producto existente en función de su código. Esto es fundamental en una aplicación de gestión de inventario, ya que permite realizar cambios en la descripción, cantidad en stock, precio, imagen y proveedor de un producto. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto. Se utiliza para identificar el producto que se desea modificar.
- **nueva_descripcion:** str, contiene la nueva descripción que se asignará al producto.
- **nueva_cantidad:** int, indica la nueva cantidad en stock del producto.
- **nuevo_precio:** float, representa el nuevo precio de venta del producto.
- **nueva_imagen:** str, contiene el nombre de la nueva imagen del producto.
- **nuevo_proveedor:** int, identifica al nuevo proveedor del producto.

Retorna:

- **Valor booleano: True** si el producto fue modificado exitosamente y **False** si no pudo realizarse la modificación.

```python
def modificar_producto(codigo, nueva_descripcion, nueva_cantidad,
nuevo_precio, nueva_imagen, nuevo_proveedor):
    for producto in productos:
        if producto['codigo'] == codigo:
            producto['descripcion'] = nueva_descripcion
            producto['cantidad'] = nueva_cantidad
            producto['precio'] = nuevo_precio
            producto['imagen'] = nueva_imagen
            producto['proveedor'] = nuevo_proveedor
            return True
    return False
```

Explicación del código:
Proceso de Modificación con bucle: La función comienza recorriendo **productos,** la lista que almacena todos los productos disponibles en la aplicación. El bucle ***for*** examina cada producto, verificando si el valor de la clave '**codigo**' en el diccionario del producto coincide con el valor proporcionado en el parámetro codigo. Esto es fundamental para identificar el producto correcto que se va a modificar. Este proceso de búsqueda condiuce a dos resultados:

a. **Producto Encontrado y salida exitosa:** Si se encuentra un producto con el código coincidente, la función procede a actualizar los valores de sus claves en el diccionario. Los valores proporcionados en los parámetros nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen y nuevo_proveedor se asignan a las claves correspondientes en el diccionario del producto, esto actualiza efectivamente los datos del producto. Después de realizar la modificación, la función devuelve ***True*** como indicación de que la operación ha tenido éxito.
b. **Producto no Encontrado:** Si el bucle finaliza sin encontrar el producto deseado, la función regresa ***False***. Esto significa que el producto no existe en la lista y, por lo tanto, no se puede modificar.

En resumen: Esta función es una parte esencial de la aplicación de gestión de inventario, ya que permite a los usuarios realizar cambios en los datos de los productos existentes. Al proporcionar un código de producto, junto con los nuevos valores para descripción, cantidad, precio, imagen y proveedor, los usuarios pueden mantener actualizado su inventario de productos de manera efectiva y eficiente.

listar_productos()

Esta función tiene como objetivo mostrar en pantalla un listado de los productos almacenados en la aplicación de gestión de inventario. Resulta de utilidad para que los usuarios puedan ver una vista general de todos los productos disponibles y sus respectivos detalles. Veamos cómo funciona:

Parámetros: No requiere.
Retorna: No retorna valores.

```python
def listar_productos():
    print("-" * 50)
    for producto in productos:
        print(f"Código.....: {producto['codigo']}")
        print(f"Descripción: {producto['descripcion']}")
        print(f"Cantidad...: {producto['cantidad']}")
```

```
        print(f"Precio.....: {producto['precio']}")
        print(f"Imagen.....: {producto['imagen']}")
        print(f"Proveedor..: {producto['proveedor']}")
        print("-" * 50)
```

Explicación del código:

1. **Separador visual de inicio:** La función comienza imprimiendo una línea de guiones (-) repetida 50 veces, lo que crea una especie de separación visual en la pantalla para distinguir entre los productos.
2. **Bucle de Productos y visualización de datos:** Luego, utiliza un bucle ***for*** para recorrer la lista **productos**, que contiene todos los productos de la aplicación. Por cada iteración del bucle, se procesa un producto y se imprimen sus detalles en la pantalla:
   - **Código**: código numérico único de identificación del producto.
   - **Descripción**: descripción alfabética que contiene una breve información sobre el producto.
   - **Cantidad**: cantidad en stock del producto, ejemplares disponibles.
   - **Precio**: precio de venta del producto en formato decimal.
   - **Imagen**: nombre de la imagen asociada al producto, si está disponible.
   - **Proveedor**: número identificador del proveedor del producto.
3. **Separador visual de cierre:** Después de mostrar los datos de cada producto, se imprime nuevamente una línea de guiones (-). Esto crea una división visual entre los productos y facilita la lectura y comprensión de la lista.

En resumen: Esta función es esencial para la aplicación de gestión de inventario, ya que permite a los usuarios obtener una vista completa de todos los productos almacenados. Al proporcionar detalles clave, como código, descripción, cantidad y precio, los usuarios pueden tomar decisiones informadas sobre la gestión y compra de productos. La función es especialmente útil cuando se necesita una visión general rápida de todo el inventario disponible.

eliminar_producto(codigo)

Esta función tiene la responsabilidad de eliminar un producto específico de la lista de productos en la aplicación de gestión de inventario. Esto puede ser necesario cuando un producto ya no está disponible o cuando se comete un error al ingresar información incorrecta. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto.

Retorna:

- **Valor booleano: True** si el producto se eliminó exitosamente del arreglo y **False** si no fue posible eliminar el producto.

```
def eliminar_producto(codigo):
    for producto in productos:
        if producto['codigo'] == codigo:
            productos.remove(producto)
            return True
    return False
```

Explicación del código:

1. **Búsqueda de Producto:** La función comienza tomando un parámetro, codigo, que representa el código numérico del producto que se desea eliminar. Este código es utilizado para identificar de manera única el producto que se desea eliminar.
2. **Recorrido de la Lista de Productos:** Luego, se inicia un bucle ***for*** que recorre la lista de productos, que contiene todos los productos almacenados en la aplicación.
3. **Verificación de Código:** En cada iteración del bucle, se verifica si el código del producto actual (contenido en el diccionario **producto**) coincide con el código proporcionado como argumento. Si se encuentra un producto con el código correspondiente, se procede a la eliminación.
4. **Eliminación del Producto:** Cuando se encuentra un producto con el código coincidente, se utiliza el método ***remove*** para eliminar ese producto específico de la lista de productos. Esto se logra al pasar el objeto producto como argumento al método ***remove***.
5. **Finalización del Bucle:** Dado que los códigos son únicos y no deberían haber duplicados, una vez que se elimina el producto, la función sale del bucle de búsqueda utilizando ***return True***. Esto indica que se ha encontrado y eliminado un producto con éxito.
6. **Producto no Encontrado:** Si el bucle de búsqueda se completa sin encontrar un producto que coincida con el código proporcionado, la función regresa ***False***. Esto indica que no se ha eliminado ningún producto porque no se encontró un producto con el código dado.

En resumen: Esta función es útil para mantener actualizada la lista de productos y permitir a los usuarios eliminar productos no deseados o incorrectos de la aplicación de gestión de inventario. Al eliminar productos, se asegura que la información sea precisa y refleje con precisión el inventario disponible.

Ejemplo del uso de las funciones implementadas

```
# Agregamos productos a la lista:
agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500, 'teclado.jpg',
101)
agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg', 102)
agregar_producto(3, 'Monitor LCD 22 pulgadas', 15, 52500,
'monitor22.jpg', 103)
agregar_producto(4, 'Monitor LCD 27 pulgadas', 25, 78500,
'monitor27.jpg', 104)
agregar_producto(5, 'Pad mouse', 5, 500, 'padmouse.jpg', 105)
agregar_producto(3, 'Parlantes USB', 4, 2500, 'parlantes.jpg', 105) # No
es posible agregarlo, mismo código que el producto 3.

# Listamos todos los productos en pantalla
listar_productos()

# Consultar un producto por su código
cod_prod = int(input("Ingrese el código del producto: "))
producto = consultar_producto(cod_prod)
if producto:
    print(f"Producto encontrado: {producto['codigo']} -
{producto['descripcion']}")
else:
```

```
    print(f'Producto {cod_prod} no encontrado.')

# Modificar un producto por su código
modificar_producto(1, 'Teclado mecánico 62 teclas', 20, 34000,
'tecladomecanico.jpg', 106)

# Listamos todos los productos en pantalla
listar_productos()

# Eliminamos un producto del inventario
eliminar_producto(5)

# Listamos todos los productos en pantalla
listar_productos()
```

ETAPA 2: CONVERSIÓN A CLASES Y OBJETOS

El objetivo de esta etapa es convertir las funciones vistas en la etapa anterior en objetos y clases. Para ello vamos a adaptar el código desarrollado antes al paradigma de **objetos** en Python. Para ello, crearemos una **clase** llamada **Catálogo** que encapsulará los datos y las operaciones relacionadas con los productos, trabajadas anteriormente mediante funciones.
Las clases nos permitirán crear objetos para nuestro proyecto.

Clase CATÁLOGO

Definiremos una clase llamada `Catalogo`, que se utiliza para administrar un catálogo de productos. Cada producto en el catálogo se representa como un diccionario que contiene información sobre el producto, como su código, descripción, cantidad en stock, precio, imagen y proveedor. La clase `Catalogo` ofrece métodos para agregar, consultar, modificar, listar y eliminar productos en el catálogo, así como para mostrar los detalles de un producto específico.
El código para inicializar la clase es el siguiente:

```
class Catalogo:
    productos = []
```

Esta clase posee un **atributo de clase** llamado **productos**, una lista que almacena los productos. Al ser un atributo de clase es compartido por todas las instancias de la clase `**Catalogo**`. Esta lista se inicializa vacía y se llena luego con diccionarios que representan productos.
Veamos los métodos de la Clase `Catalogo`:

agregar_producto(self, codigo, descripcion, cantidad, precio, imagen, proveedor)

Este método permite agregar productos al catálogo asegurándose de que no haya duplicados en función de su código. Si el producto no existe previamente, se crea un nuevo diccionario para representar el producto y se agrega a la lista de productos. El método devuelve ***True*** si el producto se agrega exitosamente y ***False*** si ya existe un producto con el mismo código. Veamos cómo funciona:
Parámetros:

- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.

- **codigo**: Un número entero que representa el código numérico del producto.
- **descripcion**: Una cadena de texto que proporciona una descripción alfabética del producto.
- **cantidad**: Un número entero que indica la cantidad en stock del producto.
- **precio**: Un número decimal (punto flotante) que representa el precio de venta del producto.
- **imagen**: Una cadena de texto que especifica el nombre de la imagen del producto.
- **proveedor**: Un número entero que identifica al proveedor del producto.

Retorna:

- **Valor booleano: True** si el producto se agregó exitosamente al arreglo y **False** si ya existe un producto con el mismo código y no se agrega el nuevo producto.

```
    def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        if self.consultar_producto(codigo):
            return False

        nuevo_producto = {
            'codigo': codigo,
            'descripcion': descripcion,
            'cantidad': cantidad,
            'precio': precio,
            'imagen': imagen,
            'proveedor': proveedor
        }

        self.productos.append(nuevo_producto)
        return True
```

Explicación del código:

1. **Verificación de Duplicados:** Antes de agregar un nuevo producto, el método realiza una verificación para asegurarse de que no exista ya un producto con el mismo código. Esto se hace llamando al método **self.consultar_producto(codigo)**.
2. **Método `consultar_producto`:** Este método se utiliza para verificar si ya existe un producto en el catálogo con el mismo código. Si se encuentra un producto con el mismo código, esto significa que no se debe agregar el nuevo producto y se devuelve ***False*** para indicar que no se realizó la operación. De lo contrario, se continúa con la adición del nuevo producto.
3. **Creación del Nuevo Producto:** Si no se encuentra un producto con el mismo código, se procede a crear un nuevo producto en forma de diccionario. Los atributos del nuevo producto, como 'codigo', 'descripcion', 'cantidad', 'precio', 'imagen' y 'proveedor', se establecen según los valores proporcionados en los parámetros del método.
4. **Agregar el Nuevo Producto al Catálogo:** Una vez creado el diccionario que representa el nuevo producto, se agrega a la lista de productos del catálogo (**self.productos**) utilizando el método **append**. Esto aumenta la lista de productos con el nuevo producto.
5. **Valor de Retorno:** Finalmente, el método devuelve ***True*** para indicar que la operación se completó con éxito y que el producto se agregó al catálogo.

consultar_producto(self, codigo)

Este método se encarga de **buscar y recuperar** la información de un producto en el catálogo de acuerdo a su código. Si se encuentra un producto con el código proporcionado, se devuelve un diccionario con los datos del producto. Si no se encuentra ningún producto con ese código, se

devuelve ***False*** para indicar que el producto no está en el catálogo. Este método le permite al método **agregar_producto** verificar si un producto no fue agregado previamente al catálogo, evitando así la duplicación de registros. Veamos cómo funciona:

Parámetros:
- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se quiere consultar.

Retorna:
- Si el producto fue encontrado retorna un **diccionario con los datos del producto**. Si no se encontró retorna el valor booleano ***False***.

```
    def consultar_producto(self, codigo):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                return producto
        return False
```

Explicación del código:
1. **Búsqueda en el Catálogo**: El método recorre la lista de productos en el catálogo (**self.productos**) utilizando un bucle ***for***. Para cada producto en la lista, comprueba si el valor de la clave '**codigo**' en el diccionario del producto coincide con el código proporcionado como parámetro.
2. **Coincidencia de Códigos:** Si se encuentra un producto cuyo código coincide con el código proporcionado, se devuelve ese producto en forma de diccionario utilizando la instrucción ***return producto***.
3. **Producto No Encontrado:** Si el bucle de búsqueda finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False*** para indicar que el producto no se encontró en el catálogo.

modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor)

Este método permite actualizar los datos de un producto existente en el catálogo utilizando su código. Si se encuentra un producto con el código proporcionado, se actualizan sus datos y se devuelve ***True*** para indicar una modificación exitosa. Si no se encuentra ningún producto con ese código, se devuelve ***False*** para indicar que el producto no existe en el catálogo. Veamos cómo funciona:

Parámetros:
- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea modificar.
- **nueva_descripcion:** Una cadena de texto que representa la nueva descripción alfabética del producto.
- **nueva_cantidad:** Un número entero que representa la nueva cantidad en stock del producto.
- **nuevo_precio:** Un número de punto flotante que representa el nuevo precio de venta del producto.

- **nueva_imagen:** Una cadena de texto que representa la nueva imagen del producto.
- **nuevo_proveedor:** Un número entero que representa el nuevo número de proveedor del producto.

Retorna:
- **Valor booleano: True** si el producto fue modificado exitosamente y **False** si no pudo realizarse la modificación.

```
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                producto['descripcion'] = nueva_descripcion
                producto['cantidad'] = nueva_cantidad
                producto['precio'] = nuevo_precio
                producto['imagen'] = nueva_imagen
                producto['proveedor'] = nuevo_proveedor
                return True
        return False
```

Explicación del código:
1. **Búsqueda en el Catálogo**: El método recorre la lista de productos en el catálogo (**self.productos**) utilizando un bucle ***for***. Para cada producto en la lista, comprueba si el valor de la clave '**codigo**' en el diccionario del producto coincide con el código proporcionado como parámetro.
2. **Modificación de Datos:** Si se encuentra un producto cuyo código coincide con el código proporcionado, se actualizan los datos del producto con los nuevos valores proporcionados para descripción, cantidad, precio, imagen y proveedor.
3. **Resultado de la Modificación:** El método devuelve ***True*** para indicar que los datos del producto se han modificado con éxito.
4. **Producto No Encontrado:** Si el bucle de búsqueda finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False*** para indicar que el producto no existe en el catálogo y, por lo tanto, no se pueden realizar modificaciones.

listar_productos(self)

Este método permite mostrar en pantalla un listado detallado de todos los productos que se encuentran en el catálogo. Veamos cómo funciona:

Parámetros: No requiere.
Retorna: No retorna valores.

```
    def listar_productos(self):
        print("-" * 50)
        for producto in self.productos:
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen']}")
            print(f"Proveedor..: {producto['proveedor']}")
```

```
            print("-" * 50)
```

Explicación del código:

1. **Separador visual de inicio:** Cuando el método se invoca, inmediatamente muestra una línea divisoria hecha de guiones para separar visualmente los distintos productos en el listado. Esto mejora la presentación en pantalla.
2. **Bucle de recorrido y visualización de datos:** Luego, utiliza un bucle ***for*** para recorrer la lista de productos en el catálogo (***self.productos***). Para cada producto en la lista, realiza las siguientes acciones:
   a. **Muestra de Datos:** El método imprime en pantalla información detallada sobre cada producto, incluyendo los siguientes datos:
      - **Código**: El código numérico del producto (`producto['codigo']`).
      - **Descripción:** La descripción alfabética del producto (`producto['descripcion']`).
      - **Cantidad:** La cantidad en stock del producto (`producto['cantidad']`).
      - **Precio:** El precio de venta del producto (`producto['precio']`).
      - **Imagen:** El nombre de la imagen asociada al producto (`producto['imagen']`).
      - **Proveedor:** El número de proveedor del producto (`producto['proveedor']`).
   b. **Separador visual de cierre:** Después de mostrar los datos de cada producto, se imprime nuevamente una línea de guiones (-). Esto crea una división visual entre los productos y facilita la lectura y comprensión de la lista.

El método ***listar_productos*** ofrece una representación visual clara de todos los productos presentes en el catálogo. Para cada producto, muestra sus datos clave en un formato estructurado. Esto facilita la revisión y gestión de los productos almacenados en el catálogo.

eliminar_producto(self, codigo)

Este método permite eliminar un producto específico del catálogo según su código. Veamos cómo funciona:

Parámetros:

- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea eliminar.

Retorna: No retorna valores.

```
    def eliminar_producto(self, codigo):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                self.productos.remove(producto)
                return True
        return False
```

Explicación del código:

1. **Bucle de recorrido y comparación de códigos:** El método utiliza un bucle ***for*** utiliza para recorrer la lista de productos en el catálogo (***self.productos***). En cada iteración del bucle, compara el código del producto actual (`**producto['codigo']**`) con el código proporcionado como argumento (`**codigo**`).
2. **Eliminación del Producto:** Si el código del producto coincide con el código proporcionado, significa que se ha encontrado el producto que se desea eliminar. En este caso, el método

elimina el producto de la lista del catálogo utilizando el método `**remove**` de las listas de Python.

3. **Valor de Retorno:** Una vez que se ha eliminado el producto, el método devuelve ***True*** para indicar que la operación se completó con éxito. Esto permite saber que el producto se eliminó correctamente.
4. **Producto No Encontrado:** Si el bucle de recorrido finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False***. Esto indica que el producto no existe en el catálogo y, por lo tanto, no se pudo eliminar.

El método ***eliminar_producto*** proporciona una forma efectiva de gestionar la eliminación de productos en el catálogo. Al especificar el código del producto que se desea eliminar, se elimina de la lista y se confirma el éxito de la operación mediante el valor de retorno. Esto facilita la gestión del catálogo y la eliminación de productos no deseados.

mostrar_producto(self, codigo)

Este método tiene como objetivo mostrar los datos de un producto específico del catálogo según su código. Veamos cómo funciona:

Parámetros:

- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea eliminar.

Retorna:

- **Valor booleano: True** si el producto se eliminó exitosamente del arreglo y **False** si no fue posible eliminar el producto.

```
    def mostrar_producto(self, codigo):
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 50)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 50)
        else:
            print("Producto no encontrado.")
```

Explicación del código:

1. **Consulta de Producto:** El método utiliza la función **consultar_producto(codigo)** de la misma clase Catalogo para buscar un producto con el código proporcionado. El resultado se almacena en la variable **producto**.
2. **Verificación de Existencia del Producto:** Se verifica si **producto** es diferente de *False*, lo que significa que se ha encontrado un producto con el código especificado. Si se encontró el producto, el método procede a mostrar sus detalles.

3. **Mostrar Datos del Producto:** Si se encuentra el producto, se muestra un bloque de información detallada sobre el producto en la pantalla. Esto incluye su código, descripción, cantidad en stock, precio, imagen y proveedor. La presentación se realiza utilizando declaraciones `print`.
4. **Producto No Encontrado:** Si no se encuentra ningún producto con el código proporcionado, se muestra un mensaje que indica "Producto no encontrado".

El método ***mostrar_producto*** es útil para visualizar los detalles de un producto en el catálogo. Si el producto existe, se muestran todos sus atributos. Si no se encuentra el producto, se informa al usuario que el producto no se encuentra en el catálogo. Este método es una herramienta eficaz para acceder a información específica sobre productos en el catálogo de una manera organizada y legible.

Ejemplo del uso de las funciones implementadas

```
catalogo = Catalogo()
catalogo.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500,
'teclado.jpg', 101)
catalogo.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg',
102)
print()
print("Listado de productos:")
catalogo.listar_productos()
print()
print("Datos de un producto:")
catalogo.mostrar_producto(1)
catalogo.eliminar_producto(1)
print()
print("Listado de productos:")
catalogo.listar_productos()
```

ETAPA 3: CREACIÓN DE LA BASE DE DATOS SQL

Utilizar un sistema gestor de bases de datos MySQL para lograr la persistencia de los datos

La clase "**Catalogo**" proporciona una interfaz sencilla para administrar productos en una base de datos MySQL. Cada método realiza una operación específica, como agregar, consultar, modificar, listar o eliminar productos. Esta implementación es útil para aquellos que deseen crear una aplicación de gestión de inventario o catálogo de productos utilizando Python y MySQL.
Es importante tener en cuenta que se requiere la instalación del paquete **mysql-connector-python** para utilizar esta clase y que se deben proporcionar credenciales válidas para acceder a una base de datos MySQL.
El paquete **mysql-connector-python** es un conector oficial de MySQL para Python. Se utiliza para conectarse y comunicarse con bases de datos MySQL desde aplicaciones Python.

Instalación de MySQL:

Para instalar **mysql-connector-python**, puedes utilizar **pip**, que es el administrador de paquetes de Python. Abre una terminal o línea de comandos y ejecuta el siguiente comando:

```
pip install mysql-connector-python
```

Este comando descargará e instalará el paquete **mysql-connector-python** y todas sus dependencias en tu entorno de Python.
Una vez instalado, puedes utilizar este paquete para conectar tu aplicación Python a una base de datos MySQL y realizar operaciones de base de datos, como consultas SQL, inserciones, actualizaciones y eliminaciones de registros.
Es importante mencionar que debes proporcionar credenciales válidas para conectarte a la base de datos, como el nombre de usuario, la contraseña, la dirección del servidor de la base de datos y el nombre de la base de datos a la que deseas acceder. El paquete **mysql-connector-python** facilita la conexión y la interacción con bases de datos MySQL desde Python, lo que lo hace útil para desarrollar aplicaciones que requieran almacenar y gestionar datos de manera persistente en bases de datos MySQL.

Introducción

El código proporcionado es una implementación de una clase llamada "Catalogo", que se utiliza para administrar un catálogo de productos almacenados en una base de datos MySQL. Este documento detalla la funcionalidad de cada método de la clase. Para acceder a los datos se utilizan algunos elementos que debemos conocer.

Conector:
Un "conector" se refiere a un conjunto de herramientas o una biblioteca que facilita la interacción entre un lenguaje de programación (como Python) y un sistema de gestión de bases de datos (como MySQL), manejando aspectos como la conexión, la ejecución de consultas y el manejo de transacciones.
En el contexto de la programación y las bases de datos, un "**conector**" es un software o una biblioteca que permite a un programa (como una aplicación escrita en Python) conectarse y comunicarse con un sistema de gestión de bases de datos (como MySQL). En nuestro código, el término "conector" se refiere específicamente a la biblioteca ***mysql.connector***, que proporciona funcionalidades para interactuar con bases de datos MySQL desde Python. Otras características son:

1. **Interfaz entre Python y MySQL:** ***mysql.connector*** es un controlador que proporciona una interfaz entre Python y una base de datos MySQL. Permite que tu aplicación Python ejecute operaciones de base de datos como consultas, actualizaciones y transacciones.
2. **Establecimiento de la Conexión:** El conector se utiliza para establecer una conexión con la base de datos MySQL. Esto se hace especificando los detalles necesarios como el host, el nombre de usuario, la contraseña y el nombre de la base de datos. En tu código, esto se realiza a través de:

```
self.conn = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)
```

3. **Manejo de Sesiones de Base de Datos:** Una vez establecida la conexión, el conector administra la sesión de la base de datos, permitiéndote realizar operaciones como ejecutar comandos SQL, manejar transacciones y cerrar la conexión.

4. **Compatibilidad y Estándares:** Los conectores, como ***mysql.connector*** para Python, generalmente son compatibles con los estándares de la industria, como el API de Base de Datos de Python (DB-API). Esto hace que sea más fácil para los desarrolladores trabajar con diferentes bases de datos de manera consistente.
5. **Gestión de Recursos y Errores:** El conector también gestiona aspectos importantes como el pooling de conexiones, el manejo de errores y excepciones, y la conversión entre tipos de datos de Python y SQL.
6. **Ejecución de Consultas y Recuperación de Resultados:** Aunque el trabajo de ejecutar consultas y recuperar resultados se realiza a través de los cursores, el conector es responsable de crear estos cursores y mantener la conexión que los respalda.

Cursor:
En el contexto de bases de datos, particularmente en nuestro código que utiliza MySQL con Python, un **"cursor"** es un objeto que se utiliza para interactuar con el sistema de gestión de la base de datos. Es fundamental para ejecutar consultas SQL y recuperar datos de la base de datos. Esto es lo que debes saber sobre los cursores:

1. **Intermediario entre Python y la Base de Datos:** El cursor actúa como un intermediario entre tu programa Python y la base de datos MySQL. Permite ejecutar comandos SQL (como SELECT, INSERT, UPDATE, DELETE) desde Python.
2. **Ejecución de Consultas:** Utilizas el cursor para ejecutar consultas SQL. Por ejemplo, *cursor.execute("SELECT * FROM tabla")* ejecuta una consulta SQL para seleccionar todos los registros de una tabla.
3. **Recuperación de Datos:** Después de ejecutar una consulta SELECT, el cursor puede usarse para obtener los resultados. Puedes iterar sobre el cursor o usar métodos como *fetchone()*, *fetchall()* para recuperar filas de la base de datos.
4. **Manejo de Transacciones:** El cursor es utilizado para manejar transacciones con la base de datos. Por ejemplo, después de insertar o actualizar datos, necesitas hacer un ***commit*** de la transacción para que los cambios se guarden permanentemente en la base de datos. En tu código, esto se hace mediante *self.conn.commit()*.
5. **Configuración del Cursor:** En tu código, el cursor se crea con la opción ***dictionary=True***, lo que significa que los resultados de las consultas se devolverán como diccionarios de Python en lugar de tuplas, permitiendo un acceso más fácil y legible a los datos por nombre de columna, en lugar de solo por índice.
6. **Cierre del Cursor:** Es una buena práctica cerrar el cursor con ***cursor.close()*** cuando ya no se necesita, para liberar recursos del sistema de gestión de la base de datos.

El **cursor** es esencial para ejecutar consultas SQL, manejar resultados y controlar transacciones con la base de datos MySQL.

Definición de la base de datos

Los datos de productos se almacenan en la base de datos MySQL que definimos a continuación.

Tabla de Productos (productos):

- **codigo:** Un número único que identifica cada producto.
- **descripcion:** Una cadena de texto que describe el producto.
- **cantidad:** Un número que representa la cantidad de productos disponibles en el inventario.
- **precio:** Un número que representa el precio del producto.
- **imagen_url:** Una cadena de texto que almacena la URL de la imagen del producto.
- **proveedor:** Un número que representa al proveedor del producto.

Crear la base de datos y sus tablas

Para crear la base de datos y las tablas en **XAMPP**, primero debes asegurarte de que el **servidor MySQL** esté en funcionamiento. Luego, puedes utilizar una herramienta como **phpMyAdmin** o ejecutar comandos SQL directamente en la consola de MySQL.
Puedes (casi siempre) acceder a phpMyAdmin desde http://127.0.1.1/phpmyadmin/
Aquí tienes un script SQL que puedes utilizar para crear la base de datos y las tablas, asegurándote de que las claves primarias se definan correctamente y que las tablas se creen si no existen:

```
-- Crear la base de datos si no existe
CREATE DATABASE IF NOT EXISTS miapp;
-- Usar la base de datos
USE miapp;
-- Crear la tabla de Productos si no existe
CREATE TABLE IF NOT EXISTS productos (
codigo INT,
descripcion VARCHAR(255) NOT NULL,
cantidad INT(4) NOT NULL,
precio DECIMAL(10, 2) NOT NULL,
imagen_url VARCHAR(255),
proveedor INT(2));
```

Clase Catalogo

La clase **Catalogo** tiene como utilidad principal gestionar un catálogo de productos almacenados en una base de datos MySQL. Proporciona una interfaz para realizar operaciones comunes en un catálogo de productos, como agregar nuevos productos, consultar información sobre productos, modificar productos existentes, listar todos los productos en el catálogo, eliminar productos y mostrar detalles de un producto en particular.
Esta clase facilita la administración de productos en una base de datos MySQL a través de los siguientes métodos:

Constructor: def __init__(self, host, user, password, database):

Este método es el constructor de la clase. Iicializa una instancia de **Catalogo** y crea una conexión a la base de datos. Toma cuatro argumentos: `host`, `user`, `password`, y `database`, que se utilizan para establecer una conexión con la base de datos.
Dentro del constructor, se crea una conexión a la base de datos MySQL y se configura un cursor para que devuelva resultados en forma de diccionarios.
Luego, se verifica si la tabla **"productos"** existe en la base de datos. Si no existe, se crea la tabla con las columnas necesarias.

Argumentos (Args):

- **host (str):** La dirección del servidor de la base de datos.
- **user (str):** El nombre de usuario para acceder a la base de datos.
- **password (str):** La contraseña del usuario.
- **database (str):** El nombre de la base de datos.

Este es el código completo de la clase **Catálogo**, hasta el momento:

```
import mysql.connector

class Catalogo:

    def __init__(self, host, user, password, database):

        self.conn = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
        )

        self.cursor = self.conn.cursor(dictionary=True)
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
            codigo INT,
            descripcion VARCHAR(255) NOT NULL,
            cantidad INT(4) NOT NULL,
            precio DECIMAL(10, 2) NOT NULL,
            imagen_url VARCHAR(255),
            proveedor INT(2))''')
        self.conn.commit()
```

Método Agregar Producto: def agregar_producto(self, codigo, descripcion, cantidad, precio, imagen, proveedor):

Este método tiene como objetivo principal agregar un nuevo producto a una base de datos. Para llevar a cabo esta tarea, se requieren varios parámetros que describen las características del producto a agregar. A continuación, se detalla el funcionamiento de este método:

Argumentos (Args):

- **codigo (int):** El código del producto. Debe ser un número entero.
- **descripcion (str):** La descripción del producto. Se espera una cadena de texto que describa el producto de manera clara.
- **cantidad (int):** La cantidad en stock del producto. Debe ser un número entero que representa la cantidad de unidades disponibles.
- **precio (float):** El precio del producto. Este valor es un número en formato decimal que refleja el costo del producto.
- **imagen (str):** La URL de la imagen del producto. Debe ser una cadena que contenga una URL válida que apunte a la imagen del producto.
- **proveedor (int):** El código del proveedor del producto. Se asume que es un número entero que identifica al proveedor en la base de datos.

Retorno (Returns):

- **bool:** El método retorna un valor booleano. Si el producto se agrega con éxito a la base de datos, devuelve True. En caso de que ya exista un producto con el mismo código en la base de datos, el método retorna False.

```
   def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        # Verificamos si ya existe un producto con el mismo código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        producto_existe = self.cursor.fetchone()
        if producto_existe:
            return False

        # Si no existe, agregamos el nuevo producto a la tabla
        sql = f"INSERT INTO productos \
               (codigo, descripcion, cantidad, precio, imagen_url,
proveedor) \
               VALUES \
               ({codigo}, '{descripcion}', {cantidad}, {precio},
'{imagen}', {proveedor})"
        self.cursor.execute(sql)
        self.conn.commit()
        return True
```

Descripción del Funcionamiento:

1. El método comienza verificando si ya existe un producto con el mismo código en la base de datos. Esto se realiza mediante una consulta SQL que busca productos en la tabla productos que tengan el mismo código que el proporcionado como argumento. Si se encuentra un producto con el mismo código, se almacena esta información en la variable ***producto_existe***.
2. Luego, se verifica el valor de ***producto_existe***. Si esta variable contiene información (lo que significa que ya existe un producto con el mismo código), el método retorna ***False***, indicando que no se puede agregar un nuevo producto con un código duplicado.
3. En caso de que no exista un producto con el mismo código, se procede a agregar el nuevo producto a la base de datos. Se construye una consulta SQL que inserta una nueva fila en la tabla productos con los detalles del producto, como código, descripción, cantidad, precio, imagen y proveedor.
4. Después de crear la consulta SQL, se ejecuta mediante el método ***execute*** en el cursor (***self.conector***). La base de datos guarda el nuevo producto en la tabla productos.
5. Finalmente, se confirma la transacción con ***self.conn.commit()*** para asegurar que los cambios se guarden de manera permanente en la base de datos. Si la operación de inserción fue exitosa, el método retorna ***True*** para indicar que el producto se agregó con éxito.

En resumen: este método permite agregar nuevos productos a una base de datos MySQL después de verificar que no existan productos con el mismo código. En caso de duplicados, retorna ***False***, y en caso de éxito en la inserción, retorna ***True***.

Probaremos si estos primeros dos métodos funcionan a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo**:

```
# Programa principal
catalogo = Catalogo(host='localhost', user='root', password='',
database='miapp')

# Agregamos productos a la tabla
```

```
catalogo.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500,
'teclado.jpg', 101)
catalogo.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg',
102)
catalogo.agregar_producto(3, 'Monitor LED', 5, 25000, 'monitor.jpg', 102)
```

Si todo ha funcionado bien, en phpMyAdmin debería aparecer algo como esto:

codigo	descripcion	cantidad	precio	imagen_url	proveedor
1	Teclado USB 101 teclas	10	4500.00	teclado.jpg	101
2	Mouse USB 3 botones	5	2500.00	mouse.jpg	102
3	Monitor LED	5	25000.00	monitor.jpg	102

Método Consultar Producto: def consultar_producto(self, codigo):

Este método tiene como propósito principal consultar un producto específico en la base de datos a partir de su código. Aquí se explica en detalle cómo funciona:

Argumentos (Args):

- **codigo (int):** El código del producto a consultar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns):

- **dict:** El método retorna un diccionario que contiene la información del producto consultado. Si el producto se encuentra en la base de datos, el diccionario contendrá los detalles del producto, como código, descripción, cantidad, precio, URL de la imagen y proveedor. Si no se encuentra ningún producto con el código proporcionado, el método retorna **False**.

```
    def consultar_producto(self, codigo):
        # Consultamos un producto a partir de su código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        return self.cursor.fetchone()
```

Descripción del Funcionamiento:

1. El método inicia ejecutando una consulta SQL en la base de datos para buscar un producto específico. La consulta se realiza en la tabla productos y se seleccionan todos los campos de la fila que coincidan con el código proporcionado como argumento.
2. La ejecución de la consulta se lleva a cabo mediante el cursor de la base de datos (***self.cursor***). La base de datos busca un producto con el código especificado y recupera sus datos.
3. El resultado de la consulta se almacena en un diccionario, que contiene la información del producto. Este diccionario se genera automáticamente en formato clave-valor, donde las claves son los nombres de las columnas en la tabla productos, y los valores son los datos correspondientes al producto consultado.
4. Finalmente, el método retorna el diccionario que contiene la información del producto consultado. Si no se encuentra ningún producto con el código especificado, el método retorna ***False***, lo que indica que no se encontró ningún producto con ese código en la base de datos.

En resumen: este método es utilizado para recuperar la información detallada de un producto en la base de datos, dada su identificación única (código). Si el producto se encuentra, se devuelve un ***diccionario*** con sus detalles; de lo contrario, se retorna ***False***.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos que permitían agregar los productos):

```
# Consultamos un producto y lo mostramos
producto = catalogo.consultar_producto(1)
if producto:
    print(f"Producto encontrado: {producto['descripcion']}")
else:
    print("Producto no encontrado.")
```

Método Modificar Producto: def modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):

Este método tiene la función de actualizar los datos de un producto específico en la base de datos a partir de su código. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto que se va a modificar. Debe ser un número entero que identifica de manera única al producto en la base de datos.
- **nueva_descripcion (str):** La nueva descripción que se asignará al producto.
- **nueva_cantidad (int):** La nueva cantidad en stock del producto.
- **nuevo_precio (float):** El nuevo precio del producto.
- **nueva_imagen (str):** La nueva URL de la imagen del producto.
- **nuevo_proveedor (int):** El nuevo código del proveedor.

Retorno (Returns):

- **bool:** El método retorna True si la modificación se realizó con éxito. Si no se encontró ningún producto con el código proporcionado, el método retorna False.

```
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        # Modificamos los datos de un producto a partir de su código
        sql = f"UPDATE productos SET \
                    descripcion = '{nueva_descripcion}', \
                    cantidad = {nueva_cantidad}, \
                    precio = {nuevo_precio}, \
                    imagen_url = '{nueva_imagen}', \
                    proveedor = {nuevo_proveedor} \
                WHERE codigo = {codigo}"
        self.cursor.execute(sql)
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Descripción del Funcionamiento:

1. El método inicia construyendo una consulta SQL para actualizar un producto en la base de datos. La consulta se construye mediante una cadena de formato (f-string) que incluye todos los datos que deben ser actualizados. Se utiliza la información proporcionada como argumentos para actualizar la descripción, cantidad, precio, URL de la imagen y el código del proveedor del producto identificado por su código.
2. La consulta se ejecuta utilizando el cursor de la base de datos (***self.cursor***). La base de datos busca un producto con el código especificado y aplica las modificaciones definidas en la consulta SQL.
3. Después de la ejecución de la consulta, se realiza una confirmación de la transacción con **self.conn.commit()**. Esta confirmación asegura que los cambios se almacenan de manera permanente en la base de datos.
4. El método verifica el número de filas afectadas por la actualización a través de **self.cursor.rowcount**. Si se encontró un producto con el código especificado y se realizaron modificaciones, **self.cursor.rowcount** será mayor que 0, y el método retorna ***True***. De lo contrario, si no se encontró el producto, se retorna ***False***.

En resumen: este método permite actualizar los detalles de un producto en la base de datos a través de su código. Devuelve ***True*** si se realizó la modificación con éxito y ***False*** si no se encontró ningún producto con el código especificado.

Método Mostrar Producto: def mostrar_producto(self, codigo):

Este método tiene como objetivo mostrar en la consola los datos de un producto a partir de su código. Aquí está una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto a mostrar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns): este método no retorna valores.

```
    def mostrar_producto(self, codigo):
        # Mostramos los datos de un producto a partir de su código
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 40)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen_url']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 40)
        else:
            print("Producto no encontrado.")
```

Descripción del Funcionamiento:

1. El método toma un único argumento, que es el código del producto que se desea mostrar. El código se pasa como el parámetro **codigo**.

2. En la primera parte del método, se utiliza el método **consultar_producto(codigo)** para obtener un diccionario con la información del producto que corresponde al código proporcionado. Esto se hace llamando al método **consultar_producto** que ya hemos explicado previamente.
3. Se verifica si el producto fue encontrado en la base de datos. Si producto es un diccionario (lo que significa que se encontró un producto con el código especificado), se procede a mostrar los detalles del producto.
4. Si el producto se encontró, se muestra un encabezado visualizado con guiones (-) para separar claramente los detalles del producto. Luego, se imprimen en la consola los siguientes datos del producto:
   - Código
   - Descripción
   - Cantidad en stock
   - Precio
   - URL de la imagen
   - Código del proveedor
5. Después de mostrar los detalles del producto, se imprime otro encabezado de guiones para una mejor visualización.
6. Si no se encuentra ningún producto con el código proporcionado, se imprime **"Producto no encontrado"** en la consola.

En resumen: Este método permite mostrar de manera detallada la información de un producto específico en la consola. Si el producto con el código especificado se encuentra en la base de datos, sus detalles se muestran en la consola. Si no se encuentra ningún producto con ese código, se muestra un mensaje indicando que el producto no fue encontrado. Este método es útil para obtener información detallada sobre un producto específico en el catálogo.

Comprobaremos el funcionamiento de los dos métodos para modificar el producto y mostrarlo a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Modificamos un producto y lo mostramos
catalogo.mostrar_producto(1)
catalogo.modificar_producto(1, 'Teclado mecánico', 20, 34000,
'tecmec.jpg', 106)
catalogo.mostrar_producto(1)
```

Método Listar Productos: def listar_productos(self):

Este método tiene la finalidad de mostrar en pantalla un listado de todos los productos almacenados en la tabla de la base de datos. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args): Este método no requiere argumentos.
Retorno (Returns): Este método no tiene valor de retorno.

```
    def listar_productos(self):
        # Mostramos en pantalla un listado de todos los productos en la
tabla
        self.cursor.execute("SELECT * FROM productos")
        productos = self.cursor.fetchall()
        print("-" * 40)
        for producto in productos:
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen_url']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 40)
```

Descripción del Funcionamiento:

1. El método comienza ejecutando una consulta SQL en la base de datos para seleccionar todos los registros de la tabla "productos" mediante la instrucción **self.cursor.execute("SELECT * FROM productos")**.
2. Luego, utiliza **self.cursor.fetchall()** para recuperar todos los resultados de la consulta. Estos resultados se almacenan en la variable **productos**, que es una lista de diccionarios. Cada diccionario representa un producto y contiene información detallada sobre el mismo.
3. A continuación, el método imprime una línea de guiones "-" repetidos 40 veces para crear un encabezado visual en la salida. Esto se hace con **print("-" * 40)**.
4. Después, el método recorre la lista de productos con un **bucle for**. En cada iteración, toma un diccionario que representa un producto y muestra su información en pantalla de manera legible.
    - La línea **print(f"Código.....: {producto['codigo']}")** muestra el código del producto.
    - La línea **print(f"Descripción: {producto['descripcion']}")** muestra la descripción del producto.
    - La línea **print(f"Cantidad...: {producto['cantidad']}")** muestra la cantidad en stock del producto.
    - La línea **print(f"Precio.....: {producto['precio']}")** muestra el precio del producto.
    - La línea **print(f"Imagen.....: {producto['imagen_url']}")** muestra la URL de la imagen del producto.
    - La línea **print(f"Proveedor..: {producto['proveedor']}")** muestra el código del proveedor del producto.
    - Al final de cada iteración del bucle, se muestra otra línea de guiones para separar visualmente cada producto.

En resumen: este método recopila información sobre todos los productos de la base de datos y la muestra de manera organizada en la salida. Esto facilita la visualización y la gestión de todos los productos en el catálogo. Cada producto se muestra con su código, descripción, cantidad, precio, imagen y proveedor, lo que permite una rápida revisión de la información de la base de datos.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Listamos todos los productos
catalogo.listar_productos()
```

Método Eliminar Producto:

Este método tiene como objetivo eliminar un producto de la tabla de la base de datos a partir de su código. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto a eliminar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns):

- **bool:** True si se eliminó el producto con éxito, False si no se encontró el producto.

```
    def eliminar_producto(self, codigo):
        # Eliminamos un producto de la tabla a partir de su código
        self.cursor.execute(f"DELETE FROM productos WHERE codigo =
{codigo}")
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Descripción del Funcionamiento:

1. El método toma un único argumento, que es el código del producto a eliminar. Este código se pasa como el parámetro **codigo**.
2. En la primera parte del método, se ejecuta una consulta SQL utilizando **self.cursor.execute(f"DELETE FROM productos WHERE codigo = {codigo}")**. Esta consulta se encarga de eliminar el registro de la tabla "productos" que coincida con el código proporcionado. En otras palabras, elimina el producto que tiene el código especificado.
3. Luego, se llama a **self.conn.commit()** para confirmar los cambios en la base de datos. Esto es importante porque las modificaciones en la base de datos, como la eliminación de registros, no se hacen efectivas hasta que se confirman.
4. Finalmente, el método evalúa si la eliminación fue exitosa. Lo hace revisando el valor de **self.cursor.rowcount**. Si es mayor que 0, **significa que se eliminó al menos un registro de la base de datos y el método devuelve True**, indicando que la eliminación se realizó con éxito. Si self.cursor.rowcount es igual a 0, significa que no se encontró ningún registro con el código proporcionado y el método devuelve **False**.

En resumen: Este método permite eliminar un producto de la base de datos a partir de su código. Si el producto con el código especificado existe en la base de datos y se elimina correctamente, el método devuelve ***True***. Si no se encuentra ningún producto con ese código, devuelve ***False***. Este método es útil para gestionar la base de datos y mantener actualizado el catálogo de productos, permitiendo la eliminación de productos que ya no están disponibles.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Eliminamos un producto
catalogo.eliminar_producto(2)
```

```
catalogo.listar_productos()
```

Conclusiones

Hemos explorado detalladamente la funcionalidad de la **clase Catalogo** y sus métodos para administrar un catálogo de productos almacenados en una base de datos MySQL. Cada uno de los métodos de esta clase tiene un propósito específico y proporciona una funcionalidad esencial para trabajar con productos. A continuación, destacamos algunas conclusiones clave:

- **Facilidad de Uso:** La clase Catalogo se ha diseñado para ser accesible y fácil de usar, incluso para personas con conocimientos básicos de programación en Python. Los métodos están bien estructurados y cuentan con documentación que describe claramente sus propósitos y cómo usarlos.
- **Administración de Productos:** Los métodos agregar_producto, consultar_producto, modificar_producto, listar_productos, eliminar_producto, y mostrar_producto brindan un conjunto completo de funcionalidades para agregar, consultar, actualizar, eliminar y mostrar productos. Esto permite una gestión completa del catálogo de productos.
- **Mecanismo de Base de Datos:** La clase utiliza una base de datos MySQL para almacenar y recuperar información sobre los productos. Esta elección de base de datos es escalable y segura, lo que la hace adecuada para la gestión de productos a nivel empresarial.
- **Evitar Duplicados:** El método agregar_producto verifica si ya existe un producto con el mismo código antes de agregarlo a la base de datos. Esto garantiza que no haya duplicados en el catálogo.
- **Detalles de Productos:** El método mostrar_producto permite ver en detalle la información de un producto específico, lo que puede ser útil para inspeccionar detalles o tomar decisiones de compra.

En resumen: la clase **Catalogo** proporciona una solución sólida y amigable para administrar un catálogo de productos a través de una base de datos MySQL. Los métodos ofrecen una gama completa de funcionalidades y están diseñados para ser fáciles de entender y utilizar, lo que la hace valiosa tanto para principiantes como para aquellos con experiencia en Python y bases de datos.

La aplicación de estos conceptos y métodos brinda una plataforma robusta para la administración eficiente de inventarios y catálogos de productos en entornos empresariales y otros.