Actualizador POSManager

El Actualizador POSManager es un conjunto de scripts diseñados para facilitar el procesamiento y actualización de archivos relacionados con productos, ofertas y códigos de barras. Su objetivo es automatizar la integración de datos provenientes de diferentes fuentes, como archivos CSV y Excel, realizar cálculos de precios basados en descuentos y generar archivos finales organizados y listos para ser utilizados en el sistema POSManager.

Este conjunto de herramientas permite la fusión de datos entre múltiples archivos, la normalización de información, la asignación de valores calculados y la creación de archivos de salida en formatos específicos, como CSV o TXT. Además, garantiza que los archivos se guarden con nombres basados en la fecha, asegurando la trazabilidad de los procesos.

Con la interfaz gráfica de Tkinter, el usuario puede interactuar fácilmente con el sistema para seleccionar archivos de entrada y definir la ubicación de los archivos de salida, mientras que el manejo de errores y las validaciones aseguran que el proceso se realice de manera segura y sin contratiempos.

Explicación del archivo `main.py`

Este archivo configura la interfaz gráfica y maneja la conexión a la base de datos de una aplicación utilizando `tkinter` y multihilos. A continuación, se detalla cada bloque de código con su propósito:

---

1. Configuración inicial de la ventana principal

```
root = tk.Tk()
root.title("Procesamiento de Archivos para POSManager")
```

- **Propósito**: Se crea la ventana principal de la aplicación utilizando `tkinter` y se le asigna un título que aparece en la barra de la ventana.

---

2. Configuración del logger

```
configurar_logger(root)
actualizar_log = get_logger()
```

- **Propósito**: Se configura un sistema de logging para la aplicación. La función `configurar_logger` inicializa el logger, mientras que `get_logger` obtiene una referencia al logger configurado, lo que permite registrar eventos durante la ejecución de la aplicación.

---

3. Importaciones posteriores a la configuración inicial

```
from ui.inputs import crear_inputs
from ui.buttons import crear_botones
from config.db_config import DBConfig
```

- **Propósito**: Se importan módulos específicos para la interfaz de usuario y la configuración de la base de datos, lo que modulariza el código y mejora su organización.

---

4. Configuración de la ruta del archivo `config.json`

```
current_dir = os.path.dirname(os.path.abspath(__file__))
config_path = os.path.join(current_dir, 'config.json')
```

- **Propósito**: Se obtiene la ruta del directorio actual donde se encuentra el archivo `main.py`, y se genera la ruta completa al archivo `config.json` que contiene la configuración de la base de datos.

---

5. Creación de la instancia de configuración de la base de datos

```
db_config = DBConfig(config_path)
```

- **Propósito**: Se crea una instancia de la clase `DBConfig`, que se encarga de gestionar la configuración de la base de datos. Utiliza la ruta al archivo `config.json` para leer los parámetros de configuración de la base de datos.

---

6. Definición de la variable global `conexion_en_proceso`

```
conexion_en_proceso = False
```

- **Propósito**: Se define una variable global que indicará si la conexión a la base de datos está en proceso o no. Esto ayuda a controlar el flujo de la aplicación y evitar múltiples intentos simultáneos de conexión.

---

7. Función para intentar conectar a la base de datos

```
def connect_to_db():
    connection = db_config.create_connection()
    if connection:
        connection.close()
```

- **Propósito**: Se define una función que intenta establecer una conexión a la base de datos utilizando los parámetros definidos en `DBConfig`. Si la conexión es exitosa, se cierra inmediatamente para evitar mantenerla abierta innecesariamente.

---

8. Función para manejar la conexión a la base de datos en un hilo separado

```
def db_connection_thread():
    actualizar_log("Probando conexion a la base de datos...")
    connect_to_db()
```

- **Propósito**: Esta función ejecuta `connect_to_db()` en un hilo separado. Registra un mensaje de prueba en el log y llama a la función para verificar la conexión a la base de datos sin bloquear la interfaz gráfica de usuario.

---

9. Función para manejar el cierre de la ventana

```
def on_closing():
    respuesta = messagebox.askquestion("Confirmar cierre", "¿Estás
seguro de que deseas cerrar la ventana?")

    if respuesta == 'yes':
        actualizar_log("Finalizando Procesos")
        root.destroy()
```

- **Propósito**: Esta función maneja el evento de cierre de la ventana. Al intentar cerrar la ventana, se muestra un cuadro de confirmación. Si el usuario confirma, se registra un mensaje en el log y se cierra la ventana.

---

10. Configuración de la ventana para ejecutar `on_closing` al cerrarla

```
root.protocol("WM_DELETE_WINDOW", on_closing)
```

- **Propósito**: Se configura para que, cuando el usuario intente cerrar la ventana principal, se ejecute la función `on_closing` que maneja la confirmación y el cierre adecuado de la aplicación.

---

11. Creación de los inputs y botones de la interfaz

```
entry_archivo2, entry_propuesta = crear_inputs(root)
button_procesar = crear_botones(root, entry_archivo2, entry_propuesta)
```

- **Propósito**: Se crean los elementos de la interfaz gráfica, como los campos de entrada (`entry_archivo2` y `entry_propuesta`) y el botón para procesar la información. Estas funciones se importan desde los módulos `inputs.py` y `buttons.py`, lo que permite modularizar el código.

---

12. Inicialización de la aplicación y manejo de hilos

```
actualizar_log("Aplicación iniciada")
threading.Thread(target=db_connection_thread, daemon=True).start()
root.mainloop()
```

- **Propósito**: Se inicia el registro de la aplicación en el log y se ejecuta un hilo para probar la conexión a la base de datos sin bloquear la interfaz gráfica. Finalmente, se arranca el bucle principal de la aplicación con `root.mainloop()`, que mantiene la ventana abierta y interactiva.

Resumen General:

Este archivo configura la ventana principal de la aplicación con `tkinter`, maneja la conexión a la base de datos utilizando un hilo separado para no bloquear la interfaz, y define el comportamiento de cierre adecuado de la ventana. Además, establece un sistema de logging para el seguimiento de eventos y errores dentro de la aplicación.

Explicación del archivo `db_config.py`

Este archivo contiene la clase `DBConfig`, que se encarga de gestionar la configuración y la conexión a la base de datos MySQL. A continuación se detalla el funcionamiento de cada bloque de código.

---

1. Importaciones y configuración del logger

```
import json
import mysql.connector
from mysql.connector import Error
from ui.logs import get_logger
import os

actualizar_log = get_logger()
```

- **Propósito**:
  - Se importan las bibliotecas necesarias: `json` para manejar archivos JSON, `mysql.connector` para interactuar con la base de datos MySQL y `Error` para manejar excepciones.
  - Se importa `get_logger` desde el módulo de logs, lo que permite registrar eventos dentro de la aplicación.
  - Se importa `os` para gestionar rutas de archivos y verificar la existencia de archivos en el sistema.

---

2. Definición de la clase `DBConfig`

```
class DBConfig:
    def __init__(self, config_path):
        self.config_path = config_path
        self.config = None
        self.connection = None
        self.cursor = None
```

- **Propósito**:
  - La clase `DBConfig` gestiona la configuración y las operaciones relacionadas con la base de datos.
  - El constructor `__init__` recibe la ruta del archivo de configuración `config_path`, donde se almacenan las credenciales para conectar a la base de datos.

- Se inicializan las variables `config`, `connection` y `cursor`, que se utilizarán más adelante para almacenar la configuración cargada, la conexión activa y el cursor de la base de datos, respectivamente.

---

3. Carga del archivo de configuración

```
def load_config(self):
    """Carga el archivo JSON con las credenciales de la base de
datos."""
    if not os.path.exists(self.config_path):
        raise FileNotFoundError(f"Archivo de configuración no
encontrado: {self.config_path}")

    with open(self.config_path, "r") as file:
        try:
            self.config = json.load(file)
            required_keys = ["DB_HOST", "DB_USER", "DB_PASSWORD",
"DB_DATABASE"]
            for key in required_keys:
                if key not in self.config:
                    raise KeyError(f"Falta la clave {key} en el archivo
de configuración.")
        except json.JSONDecodeError as e:
            raise ValueError(f"Error al leer el archivo de
configuración: {e}")
```

- **Propósito**:
  - La función `load_config` carga el archivo de configuración en formato JSON.
  - Primero verifica que el archivo de configuración exista, lanzando un error si no se encuentra.
  - Luego, lee el archivo y carga su contenido en el atributo `self.config`.
  - Se valida que el archivo contenga las claves necesarias para la conexión a la base de datos (`DB_HOST`, `DB_USER`, `DB_PASSWORD`, `DB_DATABASE`).
  - Si hay un error al leer el archivo JSON o falta alguna clave, se lanza una excepción correspondiente.

---

4. Creación de la conexión a la base de datos

```
def create_connection(self):
    """Crea y devuelve una conexión a la base de datos MySQL usando el
archivo de configuración."""
    try:
```

```
        if self.config is None:
            self.load_config()  # Cargar configuración si no está
cargada

        self.connection = mysql.connector.connect(
            host=self.config["DB_HOST"],
            user=self.config["DB_USER"],
            password=self.config["DB_PASSWORD"],
            database=self.config["DB_DATABASE"],
            port=self.config["DB_PORT"]
        )

        if self.connection.is_connected():
            actualizar_log("Conexión exitosa!")
            return self.connection
    except (Error, FileNotFoundError, ValueError, KeyError) as e:
        actualizar_log(f"Ha ocurrido un error en la conexion: {e}")
        actualizar_log("Comprueba tu configuración.")
        return None
```

- **Propósito**:
  - La función `create_connection` intenta establecer una conexión con la base de datos MySQL utilizando los parámetros de configuración.
  - Si la conexión es exitosa, registra un mensaje en el log y devuelve la conexión.
  - Si ocurre algún error (como problemas con la configuración o la base de datos), se captura y se registra el error en el log, luego devuelve `None` para indicar que la conexión falló.

---

5. Verificación de la conexión activa

```
def check_connection(self):
    """Verifica si la conexión está activa."""
    try:
        if self.connection and self.connection.is_connected():
            return True
        else:
            actualizar_log("Conexión no activa. Reconectando")
            self.create_connection()  # Reintenta conectar
            return self.connection and self.connection.is_connected()
    except Error as e:
        actualizar_log(f"Error al verificar la conexión: {e}")
        return False
```

- **Propósito**:
    - La función `check_connection` verifica si la conexión actual a la base de datos está activa.
    - Si la conexión no está activa, intenta reconectar y devuelve `True` si la conexión es exitosa.
    - Si se produce un error al verificar la conexión, se captura y se registra en el log, y se devuelve `False`.

---

6. Cierre de la conexión

```
def close_connection(self):
    if self.connection:
        self.connection.close()
```

- **Propósito**:
    - La función `close_connection` cierra la conexión actual a la base de datos si está activa.
    - Es importante cerrar las conexiones para liberar recursos.

---

7. Apertura del cursor para consultas

```
def open_cursor(self):
    if self.check_connection():
        try:
            self.cursor = self.connection.cursor(dictionary=True)
            actualizar_log("cursor abierto para realizar consulta")
            return self.cursor
        except Error as e:
            actualizar_log(f"Ocurrio un error al querer abrir el cursor
{e}")
            raise e
    else:
        actualizar_log("La conexión no está activa. No se pudo abrir el
cursor.")
        raise ConnectionError("No se pudo abrir el cursor porque la
conexión no está activa.")
```

- **Propósito**:
    - La función `open_cursor` abre un cursor de base de datos, que se utiliza para ejecutar consultas SQL.
    - Primero verifica si la conexión está activa con `check_connection`. Si es así, abre el cursor.

- Si hay un error al abrir el cursor, se captura y se registra en el log.
- Si la conexión no está activa, se lanza un error.

Resumen General:

El archivo `db_config.py` gestiona la configuración de la conexión a la base de datos MySQL, carga las credenciales desde un archivo JSON, establece y verifica la conexión, y maneja el cursor para ejecutar consultas. Utiliza un sistema de logging para registrar las operaciones y errores, lo que facilita el seguimiento de la actividad de la base de datos en la aplicación.

Explicación del archivo `file_controller.py`

Este archivo contiene varias funciones relacionadas con el manejo de archivos en el sistema, como guardar resultados en formato CSV, gestionar configuraciones de consultas, y copiar archivos procesados. A continuación se describe cada una de las funciones con su propósito y funcionamiento:

---

1. Importaciones y configuración del logger

```
import os
import subprocess
import datetime
import shutil
import json
import csv
from datetime import datetime
from ui.logs import get_logger

actualizar_log = get_logger()
```

- **Propósito**:
  - Se importan diversas bibliotecas necesarias para manejar archivos, directorios, fechas, procesos y datos en formato JSON y CSV.
  - `get_logger` se importa desde el módulo de logs para permitir el registro de eventos y errores en el sistema.

---

2. Función `guardar_resultados_como_csv`

```
def guardar_resultados_como_csv(results, final_path, name):
    """
    Guarda los resultados de una consulta en un archivo CSV dentro de la
carpeta de salida predefinida.
    """
    if not results:
        actualizar_log("No hay resultados para guardar.")
        return

    # Definir la ruta del directorio de salida
    output_dir =
os.path.expanduser(f'~\\Documents\\PM-offer-updater\\{final_path}')

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
        actualizar_log(f"Directorio creado: {output_dir}")

    fecha_hoy = datetime.today().strftime('%Y-%m-%d')
    output_file = os.path.join(output_dir, f"{name}-{fecha_hoy}.csv")

    encabezados = results[0].keys()

    try:
        with open(output_file, mode='w', encoding='utf-8', newline='')
as archivo_csv:
            writer = csv.DictWriter(archivo_csv, fieldnames=encabezados,
delimiter=';')
            writer.writeheader()
            writer.writerows(results)

        actualizar_log(f"Resultados guardados exitosamente en
{output_file}.")
        return output_file
    except Exception as e:
        actualizar_log(f"Error al guardar resultados en CSV: {e}")
        raise e
```

- **Propósito**:
    - Esta función guarda los resultados de una consulta (en forma de lista de diccionarios) en un archivo CSV en un directorio predefinido.
    - Si no existen resultados, se registra un mensaje y la función termina.
    - Si el directorio de salida no existe, se crea. Luego se genera el archivo CSV con la fecha de hoy en su nombre.

- La función usa `csv.DictWriter` para escribir los encabezados y los resultados.
- Si hay un error al guardar el archivo, se captura y se registra.

---

3. Función `save_query_config`

```
def save_query_config(data):
    """
    Guarda la configuración de los filtros de consulta en un archivo
JSON.
    """
    output_dir =
os.path.expanduser('~\\Documents\\PM-offer-updater\\config')

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
        actualizar_log(f"Directorio creado: {output_dir}")

    archivo_configuracion = os.path.join(output_dir,
"config_query_filters.json")

    with open(archivo_configuracion, 'w') as json_file:
        json.dump(data, json_file, indent=4)

    actualizar_log(f"Configuración guardada en:
{archivo_configuracion}")
```

- **Propósito**:
  - Guarda la configuración de los filtros de consulta (pasada como `data`) en un archivo JSON dentro de la carpeta `config`.
  - Si el directorio `config` no existe, se crea. Luego, se guarda el archivo `config_query_filters.json` con el contenido proporcionado.
  - El archivo JSON se guarda con una indentación de 4 espacios para facilitar su lectura.

---

4. Función `read_query_config`

```
def read_query_config():
    """
    Lee la configuración de los filtros de consulta desde un archivo
JSON.
    """
```

```
    configuraciones_dir =
os.path.expanduser('~\\Documents\\PM-offer-updater\\config')
    archivo_configuracion = os.path.join(configuraciones_dir,
"config_query_filters.json")

    if not os.path.exists(archivo_configuracion):
        actualizar_log(f"El archivo de configuración no existe:
{archivo_configuracion}")
        return None

    try:
        with open(archivo_configuracion, 'r') as json_file:
            configuracion = json.load(json_file)

        actualizar_log(f"Configuración cargada desde:
{archivo_configuracion}")
        return configuracion

    except Exception as e:
        actualizar_log(f"Error al leer el archivo de configuración:
{str(e)}")
        return None
```

- **Propósito**:
  - Lee la configuración de los filtros de consulta desde el archivo `config_query_filters.json`.
  - Si el archivo no existe, se registra un error y se retorna `None`.
  - Si el archivo existe, se carga el contenido JSON y se devuelve como un diccionario.
  - Si ocurre algún error al leer el archivo, se captura y se registra.

---

5. Función `save_processed_files`

```
def save_processed_files():
    """
    Copia los archivos procesados a la carpeta de resultados.
    """
    fecha_hoy = datetime.today().strftime('%Y-%m-%d')
    output_dir_codebars =
os.path.expanduser('~\\Documents\\PM-offer-updater\\processed-files\\Cod
ebars')
    output_dir_items =
os.path.expanduser('~\\Documents\\PM-offer-updater\\processed-files\\cal
```

```
culated-items')

    file_path =
os.path.expanduser(f'~\\Documents\\PM-offer-updater\\processed-files\\Re
sults\\{fecha_hoy}')

    if not os.path.exists(file_path):
        os.makedirs(file_path)

    def copiar_archivo(archivo_origen, destino, nuevo_nombre):
        if os.path.exists(archivo_origen):
            shutil.copy(archivo_origen, os.path.join(destino,
nuevo_nombre))
            actualizar_log(f"Archivo copiado: {nuevo_nombre} a
{destino}")
        else:
            actualizar_log(f"Archivo no encontrado: {archivo_origen}")

    items_file = os.path.join(output_dir_items,
f"calc-items-{fecha_hoy}.txt")
    copiar_archivo(items_file, file_path, "Items.txt")

    codebars_file = os.path.join(output_dir_codebars,
f"CodBarras-{fecha_hoy}.txt")
    copiar_archivo(codebars_file, file_path, "CodBarras.txt")

    actualizar_log("Archivos copiados correctamente a la carpeta
Results.")
    return file_path
```

- **Propósito**:
  - Copia los archivos procesados de las carpetas `Codebars` y `calculated-items` a una carpeta de resultados (`Results`), utilizando la fecha de hoy en el nombre de los archivos.
  - Si el directorio `Results` no existe, se crea.
  - Usa la función `copiar_archivo` para copiar los archivos a la nueva ubicación. Si el archivo no se encuentra, se registra un error.

---

6. Función `open_file`

```
def open_file(file_path):
    """
    Abre la carpeta en el explorador de archivos del sistema.
```

```
    """
    if os.path.exists(file_path):
        if os.name == 'nt':  # Para sistemas Windows
            subprocess.run(['explorer', file_path])
        elif os.name == 'posix':  # Para sistemas Unix/Linux/Mac
            subprocess.run(['open', file_path])
        actualizar_log(f"Carpeta con resultados abierta: {file_path}")
    else:
        actualizar_log("La carpeta de resultados no existe.")
```

- **Propósito**:
  - Abre el directorio especificado en el explorador de archivos del sistema operativo.
  - Dependiendo del sistema operativo (`Windows` o `Unix/Linux/Mac`), utiliza el comando adecuado (`explorer` o `open`).
  - Si la carpeta no existe, se registra un mensaje de error.

Resumen General:

El archivo `file_controller.py` gestiona la creación, copia y apertura de archivos y directorios relacionados con los resultados de las consultas. También maneja la configuración de filtros de consulta, guardándola y leyéndola en formato JSON. Se emplean logs para registrar las acciones realizadas y se gestionan errores de forma adecuada, asegurando una operación eficiente y sin interrupciones.

Explicación del archivo `process_controller.py`

Este archivo gestiona el proceso completo de procesamiento de archivos dentro de la aplicación, que incluye la ejecución de múltiples pasos, como el procesamiento de archivos de artículos y códigos de barras, cálculo de ofertas y la normalización de datos. A continuación, se detalla cada parte del código y su funcionamiento:

---

1. Importaciones y configuración del logger

```
from ui.logs import get_logger
from controllers.file_controller import read_query_config
from libs.orquestators.quantio_items import process_file as
process_items
```

```
from libs.orquestators.quantio_barcodes import process_file as
process_barcodes
from libs.update_normalizer import procesar_archivos
from libs.offer_calculator import calcular_ofertas
from libs.barcode_selector import seleccionar_barcodes
from controllers.file_controller import save_processed_files, open_file
from tkinter import messagebox

# Obtener la función para actualizar logs
actualizar_log = get_logger()
```

- **Propósito**:
  - Se importan diversos módulos y funciones necesarias para el procesamiento de los archivos, incluyendo la configuración de logs, funciones para procesar artículos y códigos de barras, calcular ofertas y seleccionar los códigos de barras, así como la capacidad para leer y guardar archivos.
  - Se utiliza el logger `get_logger` para registrar los eventos durante el procesamiento.

---

2. Función `process`

```
def process(file_path2, file_propuesta):
    if not file_path2 or not file_propuesta:
        messagebox.showwarning("Advertencia", "Por favor, seleccione
todos los archivos antes de continuar.")
        actualizar_log("Seleccione el / los archivos faltantes.")
        return

    config = read_query_config()

    try:
        query_file_items = process_items(config['dias'])
        output_file = procesar_archivos(query_file_items, file_path2)
        items_file = calcular_ofertas(output_file, file_propuesta)
        query_file_barcodes = process_barcodes()
        codebars_file = seleccionar_barcodes(output_file,
query_file_barcodes)

        if items_file and codebars_file:
            res = save_processed_files()
            actualizar_log("Proceso completado")
            open_file(res)

    except ValueError as e:
```

```
        messagebox.showerror("Error", str(e))
        actualizar_log(f"Error: {str(e)}")
```

- **Propósito**:
    - Esta es la función principal que orquesta todo el proceso de procesamiento de archivos.
    - **Validación de archivos**:
        - Primero, verifica si los archivos necesarios (`file_path2` y `file_propuesta`) están presentes. Si alguno falta, se muestra una advertencia al usuario y se registra el mensaje en el log.
    - **Lectura de configuración**:
        - Luego, carga la configuración de la consulta utilizando la función `read_query_config` que lee los filtros de consulta desde un archivo JSON.
    - **Procesamiento de datos**:
        - Llama a varias funciones que procesan los archivos:
            - `process_items(config['dias'])`: Procesa los artículos utilizando el valor de días desde la configuración.
            - `procesar_archivos(query_file_items, file_path2)`: Procesa el archivo de artículos.
            - `calcular_ofertas(output_file, file_propuesta)`: Calcula las ofertas en base al archivo de salida y la propuesta de archivo.
            - `process_barcodes()`: Procesa los códigos de barras.
            - `seleccionar_barcodes(output_file, query_file_barcodes)`: Selecciona los códigos de barras relevantes a partir de los datos procesados.
    - **Guardar y abrir archivos procesados**:
        - Si los archivos de artículos y códigos de barras se procesan correctamente (es decir, ambos son válidos), guarda los archivos procesados con `save_processed_files()`.
        - Luego, registra que el proceso se completó exitosamente y abre la carpeta con los resultados mediante `open_file(res)`.
    - **Manejo de errores**:
        - Si ocurre una excepción de tipo `ValueError` durante el proceso, se muestra un mensaje de error y se registra en los logs.

Resumen General:

El archivo `process_controller.py` coordina el flujo completo de procesamiento de archivos. Empieza por verificar que los archivos necesarios estén presentes y válidos. Luego, procesa los archivos de acuerdo con una serie de pasos que incluyen el manejo de

artículos, códigos de barras y ofertas. Una vez completado, guarda los resultados y abre la carpeta correspondiente. Si ocurre algún error durante el proceso, se captura y se notifica al usuario. La implementación del logger asegura que se mantenga un registro detallado de todo el proceso.

Explicación del archivo `query_controller.py`

Este archivo gestiona la ejecución de consultas SQL a la base de datos, específicamente relacionadas con productos y códigos de barras. A continuación se describe el funcionamiento de cada parte del código:

---

1. Importaciones y configuración del logger

```
import mysql.connector
import os
from ui.logs import get_logger
from config.db_config import DBConfig
from queries.quantio import cod1, cod2, Q_PRODUCTS, Q_BARCODES

actualizar_log = get_logger()
```

- **Propósito**:
  - Se importan las bibliotecas necesarias, incluyendo `mysql.connector` para interactuar con MySQL, `os` para gestionar rutas de archivos, y los módulos personalizados que permiten gestionar logs y configuraciones de la base de datos.
  - Se define el logger `actualizar_log` para registrar los eventos durante el proceso de consulta.

---

2. Función `quantio_updated_products(day_filter)`

```
def quantio_updated_products(day_filter):
    connection = None
    cursor = None
    try:
        # Obtener el directorio donde se encuentra el script actual
(query_controller.py)
        current_dir = os.path.dirname(os.path.abspath(__file__))
```

```
        # Ruta completa al archivo config.json (supongamos que está en
el directorio raíz o donde se llama)
        config_path = os.path.join(current_dir, '..', 'config.json')
        # Crear una nueva conexión a la base de datos
        db_config = DBConfig(config_path)  # Asegúrate de poner la ruta
correcta al archivo de configuración
        connection = db_config.create_connection()  # Establece la
conexión

        if connection:
            cursor = connection.cursor(dictionary=True)
            actualizar_log("Cursor abierto para realizar consulta")

            # Ejecutar las sentencias SET sin usar multi=True
            cursor.execute(cod1)  # Ejecuta la primera sentencia SET
            cursor.execute(cod2)  # Ejecuta la segunda sentencia SET

            # Ejecutar la consulta SELECT con el parámetro day_filter
            cursor.execute(Q_PRODUCTS, {"day_filter": day_filter})

            # Verificar si la consulta principal devuelve resultados
            if cursor.with_rows:  # Esta propiedad es True si hay un
conjunto de resultados
                resultados = cursor.fetchall()
                actualizar_log("Consulta realizada correctamente")
                return resultados
            else:
                actualizar_log("No hay resultados para la consulta.")
                return []
        else:
            actualizar_log("No se pudo establecer la conexión a la base
de datos.")
            return []

    except mysql.connector.Error as e:
        actualizar_log(f"Error ejecutando la consulta: {e}")
        return []

    finally:
        # Asegurarse de que el cursor y la conexión se cierren después
de la consulta
        if cursor:
            cursor.close()
            actualizar_log("Cursor cerrado después de la consulta.")
        if connection:
            connection.close()  # Cerrar la conexión
```

```
            actualizar_log("Conexión cerrada.")
```

- **Propósito**:
    - **Realizar una consulta SQL sobre productos**:
        - Esta función se encarga de realizar una consulta SQL a la base de datos para obtener los productos actualizados según un filtro de días (`day_filter`).
    - **Conexión a la base de datos**:
        - Obtiene el archivo de configuración para establecer la conexión con la base de datos utilizando el objeto `DBConfig`.
        - Si la conexión es exitosa, se abre un cursor para ejecutar las consultas SQL.
    - **Ejecutar sentencias SET**:
        - Antes de ejecutar la consulta principal, se ejecutan dos sentencias `SET` (almacenadas en las variables `cod1` y `cod2`).
    - **Consulta principal**:
        - Ejecuta una consulta `SELECT` definida en `Q_PRODUCTS`, pasando `day_filter` como parámetro.
    - **Manejo de resultados**:
        - Si la consulta retorna resultados, estos se obtienen y retornan como una lista de diccionarios.
        - Si no se encuentran resultados, se devuelve una lista vacía.
    - **Manejo de errores**:
        - Si ocurre un error en cualquier parte del proceso, se captura y registra el error.
    - **Cierre de recursos**:
        - Asegura que el cursor y la conexión a la base de datos se cierren al finalizar la operación.

---

3. Función `quantio_updated_barcodes()`

```
def quantio_updated_barcodes():
    connection = None
    cursor = None
    try:
        # Obtener el directorio donde se encuentra el script actual
(query_controller.py)
        current_dir = os.path.dirname(os.path.abspath(__file__))

        # Ruta completa al archivo config.json (supongamos que está en
el directorio raíz o donde se llama)
        config_path = os.path.join(current_dir, '..', 'config.json')
        # Crear una nueva conexión a la base de datos
```

```
        db_config = DBConfig(config_path)  # Asegúrate de poner la ruta
correcta al archivo de configuración
        connection = db_config.create_connection()  # Establece la
conexión

        if connection:
            cursor = connection.cursor(dictionary=True)
            actualizar_log("Realizando consulta de codigos de barra a la
base de datos")

            # Ejecutar la consulta SELECT con el parámetro day_filter
            cursor.execute(Q_BARCODES)

            # Verificar si la consulta principal devuelve resultados
            if cursor.with_rows:  # Esta propiedad es True si hay un
conjunto de resultados
                resultados = cursor.fetchall()
                actualizar_log("Consulta realizada correctamente")
                return resultados
            else:
                actualizar_log("No hay resultados para la consulta.")
                return []
        else:
            actualizar_log("No se pudo establecer la conexión a la base
de datos.")
            return []

    except mysql.connector.Error as e:
        actualizar_log(f"Error ejecutando la consulta: {e}")
        return []

    finally:
        # Asegurarse de que el cursor y la conexión se cierren después
de la consulta
        if cursor:
            cursor.close()
        if connection:
            connection.close()  # Cerrar la conexión
            actualizar_log("Conexión cerrada.")
```

- **Propósito**:
    - **Realizar una consulta SQL sobre códigos de barras**:
        - Similar a la función anterior, esta función realiza una consulta SQL para obtener los códigos de barras actualizados desde la base de datos.

- **Conexión a la base de datos**:
  - Se conecta a la base de datos utilizando el archivo de configuración y crea un cursor para ejecutar la consulta.
- **Consulta de códigos de barras**:
  - Ejecuta una consulta SQL definida en `Q_BARCODES` para obtener los códigos de barras.
- **Manejo de resultados**:
  - Si se obtienen resultados, los devuelve como una lista de diccionarios; de lo contrario, devuelve una lista vacía.
- **Manejo de errores y cierre de recursos**:
  - Similar a la función anterior, maneja cualquier error y asegura el cierre adecuado del cursor y la conexión.

Resumen General:

El archivo `query_controller.py` se encarga de gestionar la conexión a la base de datos y ejecutar consultas SQL para obtener información relacionada con productos y códigos de barras. Utiliza un archivo de configuración para establecer la conexión y manejar los errores que puedan surgir durante el proceso de consulta. Además, asegura el cierre adecuado de la conexión y el cursor para evitar fugas de recursos. El uso del logger permite un seguimiento detallado de los eventos y errores ocurridos durante las consultas.

Explicación del archivo `selector.py`

Este archivo se encarga de realizar el cruce de información entre dos archivos CSV y generar un nuevo archivo que contiene la relación entre productos y códigos de barras. A continuación se describe el funcionamiento del código:

---

1. Importaciones y configuración del logger

```
import pandas as pd
import os
from datetime import datetime
from tkinter import messagebox
from ui.logs import get_logger

# Obtener la función para actualizar logs
actualizar_log = get_logger()
```

- **Propósito**:
    - **Pandas**: Para leer y manipular archivos CSV.
    - **OS**: Para gestionar rutas de archivos y directorios.
    - **Datetime**: Para obtener la fecha actual y usarla en la creación del archivo de salida.
    - **Messagebox**: Para mostrar alertas al usuario en caso de error.
    - `get_logger`: Se utiliza para registrar los eventos en el log.

---

2. Función `seleccionar_barcodes(output_file, barcode_query)`

Esta función realiza un cruce entre dos archivos CSV: uno con información de productos y otro con códigos de barras. Luego, genera un nuevo archivo con los resultados.

```
def seleccionar_barcodes(output_file, barcode_query):
    try:
        #Lectura de archivos
        try:
            df1 = pd.read_csv(output_file)
            df2 = pd.read_csv(barcode_query, sep=';',
encoding='utf-8-sig', on_bad_lines='skip')
        except Exception as e:
            messagebox.showerror("Error", f"Error al leer los archivos:
{e}")
            actualizar_log(f"Error al leer los archivos: {e}")
            return
```

- **Propósito**:
    - Se leen dos archivos CSV: `output_file` (que contiene información de productos) y `barcode_query` (que contiene información sobre códigos de barras).
    - Si ocurre un error al leer los archivos, se muestra un mensaje de error y se registra en los logs.

---

3. Limpiar nombres de columnas y verificar columnas necesarias

```
 # Limpiar los nombres de las columnas eliminando espacios en blanco
        df1.columns = df1.columns.str.strip()
        df2.columns = df2.columns.str.strip()

        # Verificar que las columnas necesarias existen en ambos
DataFrames
```

```
        required_columns_df1 = ['IDProducto']
        required_columns_df2 = ['IDProducto']

        missing_columns_df1 = [col for col in required_columns_df1 if
col not in df1.columns]
        missing_columns_df2 = [col for col in required_columns_df2 if
col not in df2.columns]

        if missing_columns_df1:
            actualizar_log(f"Error: Las siguientes columnas no se
encuentran en df1: {', '.join(missing_columns_df1)}")
            return

        if missing_columns_df2:
            actualizar_log(f"Error: Las siguientes columnas no se
encuentran en df2: {', '.join(missing_columns_df2)}")
            return
```

- **Propósito**:
    - Se eliminan los espacios en blanco de los nombres de las columnas en ambos DataFrames para evitar problemas durante el procesamiento.
    - Se verifican las columnas necesarias (`IDProducto`) en ambos DataFrames.
    - Si falta alguna columna, se registra un error y la función termina.

---

4. Realizar el cruce de los DataFrames

```
 # Realizar el "merge" entre las dos tablas usando 'CodigoERP', con un
'left join' para incluir filas de df1 que no están en df2
        merged_df = pd.merge(df1[['IDProducto', 'codigoInterno']],
df2[['IDProducto', 'Codebar']], on='IDProducto', how='left')

        # Reordenar las columnas según la solicitud (codigoInterno,
Codebar)
        result = merged_df[['codigoInterno', 'Codebar']]

        actualizar_log("Cruce de archivo de codigo de barras realizado")
```

- **Propósito**:
    - Se realiza un **merge (unión)** entre los dos DataFrames `df1` y `df2`, utilizando la columna `IDProducto` como clave de unión.
    - Se utiliza un `left join` para incluir todas las filas de `df1` y solo las filas coincidentes de `df2`.

- Se seleccionan las columnas `codigoInterno` y `Codebar` para el resultado final.

---

5. Guardar el resultado en un archivo

```
# OUTPUT

        # Carpeta de salida
        output_dir =
os.path.expanduser('~\\Documents\\PM-offer-updater\\processed-files\\Cod
ebars')

        # Verificar si la carpeta existe, si no, crearla
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Crear el nombre de archivo con la fecha de hoy
        fecha_hoy = datetime.today().strftime('%Y-%m-%d')
        output_file = os.path.join(output_dir,
f"CodBarras-{fecha_hoy}.txt")

        # Guardar el resultado
        result.to_csv(output_file, sep='\t', index=False, header=False)

        # Aviso del archivo creado
        actualizar_log(f"Archivo procesado y guardado en {output_file}")
        actualizar_log("---------- Proceso de preparacion de codigo de
barras Terminado ----------")

        return output_file
```

- **Propósito**:
  - Se define el directorio de salida donde se guardará el archivo procesado.
  - Si el directorio no existe, se crea.
  - Se genera un nombre para el archivo de salida utilizando la fecha actual.
  - El resultado final se guarda en un archivo `.txt` con los datos de los productos y sus códigos de barras.
  - Se registra el éxito de la operación en el log.

---

6. Manejo de excepciones

```
except Exception as e:
        messagebox.showerror("Error", f"Ocurrió un error: {e}")
        actualizar_log(f"Ocurrió un error: {e}")
        return False
```

- **Propósito**:
    - Si ocurre algún error inesperado en cualquier parte del proceso, se captura la excepción y se muestra un mensaje de error al usuario.
    - Se registra el error en los logs y se devuelve `False` indicando que el proceso no se completó correctamente.

Resumen General:

El archivo `selector.py` se encarga de procesar dos archivos CSV: uno con productos y otro con códigos de barras. Después de realizar un cruce entre ambos archivos basado en la columna `IDProducto`, genera un archivo de salida con la relación entre los códigos internos de los productos y sus respectivos códigos de barras.

- **Flujo principal**:
    1. Lee los archivos CSV.
    2. Verifica que las columnas necesarias existan en ambos archivos.
    3. Realiza el cruce de los datos (`merge`).
    4. Guarda los resultados en un archivo de texto con fecha.
    5. Maneja cualquier error que ocurra durante el proceso y proporciona retroalimentación al usuario.

Este flujo de trabajo facilita la integración y la preparación de los datos de productos y códigos de barras para su uso posterior.

Explicación del archivo `calculator.py`

Este archivo se encarga de realizar los cálculos relacionados con las ofertas, aplicando descuentos a los productos y generando un archivo de salida con los resultados. A continuación se describe el funcionamiento del código:

---

1. Importaciones y configuración del logger

```
import pandas as pd
```

```
import os
from datetime import datetime
import tkinter as tk
from tkinter import messagebox
from ui.logs import get_logger
import unicodedata

# Obtener la función para actualizar logs
actualizar_log = get_logger()
```

- **Propósito**:
  - **Pandas**: Para manejar y procesar archivos CSV y Excel.
  - **OS**: Para gestionar directorios y rutas de archivos.
  - **Datetime**: Para obtener la fecha actual y usarla al crear el archivo de salida.
  - **Messagebox**: Para mostrar alertas en la interfaz gráfica si ocurre un error.
  - `get_logger`: Función para registrar eventos en los logs.

---

2. Función `normalizar_texto(texto)`

```
def normalizar_texto(texto):
    if isinstance(texto, str):
        return ''.join(
            c for c in unicodedata.normalize('NFKD', texto)
            if not unicodedata.combining(c)
        )
    return texto
```

- **Propósito**:
  - Normaliza el texto en las celdas de los archivos CSV o Excel eliminando caracteres de combinación (como acentos) y asegurando que el texto sea limpio y fácil de procesar.
  - La función utiliza el módulo `unicodedata` para realizar esta normalización.

---

3. Función `calcular_ofertas(output_file, archivo_propuesta)`

Esta es la función principal, y realiza el cálculo de las ofertas para los productos, aplicando descuentos desde un archivo de propuesta.

```
def calcular_ofertas(output_file, archivo_propuesta):
    try:
        # Cargar archivos Excel con manejo de errores
```

```
        try:
            items_df = pd.read_csv(output_file)
            propuesta_df = pd.read_excel(archivo_propuesta)
        except Exception as e:
            messagebox.showerror("Error", f"Error al leer los archivos:
{e}")
            actualizar_log("Error al leer los archivos para calcular")
            return
```

- **Propósito**:
  - Se cargan dos archivos: el archivo de productos `output_file` (CSV) y el archivo de propuestas `archivo_propuesta` (Excel).
  - Si ocurre un error en la lectura de los archivos, se muestra un mensaje de error y se registra en los logs.

---

4. Normalización y limpieza de datos

```
 # Normalizar texto en todas las columnas
        items_df = items_df.applymap(normalizar_texto)
        propuesta_df = propuesta_df.applymap(normalizar_texto)

        # Limpiar y convertir la columna de precios a float
        items_df[items_df.columns[8]] =
items_df[items_df.columns[8]].replace({',': '.'},
regex=True).astype(float)

        # Verificar existencia de columnas
        if "IDProducto" not in items_df.columns or
propuesta_df.columns[0] != "Id Quantio":
            messagebox.showerror("Error", "¡Faltan las columnas
IDProducto o Id Quantio o su nombre ha cambiado")
            actualizar_log("Faltan las columnas IDProducto o Id Quantio
o su nombre ha cambiado. Revisalo y vuelve a intentarlo")
            actualizar_log(f"Columnas del archivo normalizado: [
{items_df} ]")
            actualizar_log(f"Columnas del archivo de propuesta: [
{propuesta_df} ]")
            return
```

- **Propósito**:
  - Se normalizan los textos en todas las columnas de ambos DataFrames.

- Se limpia y convierte la columna de precios a tipo `float` para evitar problemas de formato.
- Se verifican las columnas necesarias (`IDProducto` en `items_df` y `Id Quantio` en `propuesta_df`).
- Si alguna columna falta o su nombre ha cambiado, se muestra un error y se registran los detalles en los logs.

---

5. Verificación de la columna de oferta

```
 # Asegurar que la columna Precio_de_Oferta_Etiquetas esté en la
posición 23
        if "Precio_de_Oferta_Etiquetas" not in items_df.columns:
            messagebox.showerror("Error", "¡La columna
Precio_de_Oferta_Etiquetas no existe en el archivo Items!")
            actualizar_log("¡La columna Precio_de_Oferta_Etiquetas no
existe en el archivo Items!")
            actualizar_log(f"Columnas del archivo normalizado: [
{items_df} ]")
            return
```

- **Propósito**:
  - Se asegura que la columna `Precio_de_Oferta_Etiquetas` esté presente en el archivo `items_df` y esté en la posición esperada.
  - Si la columna no está, se muestra un error y se registra en los logs.

---

6. Función `calcular_precio_final(id_producto, precio_final)`

```
 def calcular_precio_final(id_producto, precio_final):
            try:
                # Asegurarse de que precio_final sea un número flotante
                precio_final = float(precio_final)  # Convertir
precio_final a float

                # Buscar el descuento correspondiente en el archivo de
propuesta
                if id_producto in propuesta_df.iloc[:, 0].values:
                    descuento_str =
propuesta_df.loc[propuesta_df.iloc[:, 0] == id_producto,
propuesta_df.columns[1]].values[0]
```

```
                    # Convertir el descuento a float, manejando posibles
errores
                    descuento = float(descuento_str)

                    # Verificar si el descuento es un porcentaje o un
precio fijo
                    if descuento <= 1:  # Descuento como porcentaje
                        return precio_final * (1 - descuento), True
                    else:  # Precio fijo
                        return descuento, True
                return "", False  # Si no se encuentra el producto,
devolver vacío
            except ValueError:
                # Manejar si no se puede convertir el descuento o el
precio a un número flotante
                actualizar_log(f"Error al convertir el descuento de
{id_producto} o el precio a flotante.")
                return "", False
```

- **Propósito**:
  - Esta función se utiliza para calcular el precio final de un producto, dependiendo de si el descuento proporcionado en el archivo de propuesta es un porcentaje o un valor fijo.
  - Si el producto tiene un descuento, se calcula el precio final. Si no tiene descuento, se devuelve un valor vacío.
  - Se maneja la conversión de valores a tipo `float` para asegurar que los descuentos se puedan aplicar correctamente.

---

7. Aplicar descuentos y crear archivo de salida

```
 # Calcular precios y agregar indicador de descuento
        descuentos = items_df.apply(
            lambda row: calcular_precio_final(row["IDProducto"],
row[items_df.columns[8]]), axis=1
        )
        items_df.iloc[:, 23] = [precio for precio, _ in descuentos]

        # Verificar si la columna de índice 35 existe, y agregarla si no
tiene encabezado
        if len(items_df.columns) <= 35:
            items_df.insert(35, 'Es Oferta', "")  # Agregar una nueva
columna llamada 'Es Oferta' en el índice 35
```

```
        items_df.iloc[:, 35] = ["S" if descuento else "N" for _,
descuento in descuentos]

        # Reemplazar comas por puntos en las columnas y convertirlas a
numérico
        items_df.iloc[:, [8, 22, 23, 24]] = items_df.iloc[:, [8, 22, 23,
24]].replace(",", ".", regex=True)
        items_df.iloc[:, [8, 22, 23, 24]] = items_df.iloc[:, [8, 22, 23,
24]].apply(pd.to_numeric, errors='coerce')

        # Rellenar valores NaN con 0 (opcional)
        items_df.iloc[:, [8, 22, 23, 24]] = items_df.iloc[:, [8, 22, 23,
24]].fillna(0)

        # Exportar archivo actualizado como TXT Unicode
        # Definir la ruta del directorio de salida
        output_dir =
os.path.expanduser('~\\Documents\\PM-offer-updater\\processed-files\\cal
culated-items')

        # Verificar si la carpeta existe, si no, crearla
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Obtener la fecha de hoy
        fecha_hoy = datetime.today().strftime('%Y-%m-%d')

        # Crear el nombre de archivo con la fecha de hoy
        output_file = os.path.join(output_dir,
f"calc-items-{fecha_hoy}.txt")

        # Guardar el resultado en el archivo
        items_df.to_csv(output_file, index=False, header=False,
sep='\t', encoding='utf-16', float_format="%.2f")

        # Registrar la acción
        actualizar_log(f"Archivo guardado correctamente en:
{output_file}")
        actualizar_log("---------- Proceso de cálculo de oferta
Terminado ----------")

        return True
```

- **Propósito**:

- Se calculan los precios finales y se agrega un indicador de si el producto está en oferta o no.
- Se verifica la existencia de ciertas columnas y se agregan si es necesario.
- Se prepara el archivo de salida y se guarda en formato `.txt` con codificación `utf-16`.
- Se registra en los logs que el archivo fue guardado correctamente.

---

8. Manejo de excepciones

```
except Exception as e:
        messagebox.showerror("Error", f"Ocurrió un error: {e}")
        actualizar_log(f"Error {e}")
        return False
```

- **Propósito**:
  - Si ocurre cualquier error no esperado durante el proceso, se muestra un mensaje de error y se registra en los logs.

Este archivo es una parte crucial para el cálculo y generación de los archivos de ofertas en tu sistema. Asegura que se apliquen los descuentos correctamente y guarda los resultados de forma segura.

Explicación del archivo `normalizer.py`

Este archivo está diseñado para leer y procesar dos archivos de entrada, normalizarlos y realizar un "merge" de los datos, asegurando que las columnas necesarias estén presentes y generando un archivo de salida con los datos normalizados y combinados. A continuación se detalla su funcionamiento:

---

1. Importaciones y configuración del logger

```
import pandas as pd
import os
from datetime import datetime
from ui.logs import get_logger

# Obtener la función para actualizar logs
```

```
actualizar_log = get_logger()
```

- **Propósito**:
    - **Pandas**: Para manipular y procesar los archivos CSV.
    - **OS**: Para manejar rutas de archivos y crear directorios.
    - **Datetime**: Para obtener la fecha actual y usarla en el nombre del archivo de salida.
    - `get_logger`: Permite registrar los logs de las actividades.

---

2. Función `procesar_archivos(file_path1, file_path2)`

Esta es la función principal que procesa los archivos de entrada. A continuación, se describe su flujo de trabajo:

```
def procesar_archivos(file_path1, file_path2):
    # Intentar leer los archivos
    try:
        # Leer el archivo df1 (CSV delimitado por tabulaciones)
        df1 = pd.read_csv(file_path1, sep=';', encoding='utf-8-sig',
on_bad_lines='skip')
    except pd.errors.ParserError as e:
        actualizar_log(f"Error de análisis al leer el archivo
{file_path1}: {e}")
        return
    except FileNotFoundError as e:
        actualizar_log(f"Archivo no encontrado: {file_path1}")
        return
    except Exception as e:
        actualizar_log(f"Hubo un error al leer el archivo {file_path1}:
{e}")
        return

    try:
        # Leer el archivo df2 (TXT delimitado por tabulaciones)
        df2 = pd.read_csv(file_path2, sep='\t', encoding='latin1',
on_bad_lines='skip')
    except pd.errors.ParserError as e:
        actualizar_log(f"Error de análisis al leer el archivo
{file_path2}: {e}")
        return
    except FileNotFoundError as e:
        actualizar_log(f"Archivo no encontrado: {file_path2}")
        return
```

```
    except Exception as e:
        actualizar_log(f"Hubo un error al leer el archivo {file_path2}:
{e}")
        return
```

- **Propósito**:
    - Intenta leer dos archivos (uno en formato CSV y el otro en formato TXT).
    - Si hay errores al leer los archivos (por ejemplo, el archivo no existe, tiene un formato incorrecto o cualquier otro tipo de error), se registra en los logs y la función retorna sin continuar.

---

3. Normalización de los DataFrames

```
 # NORMALIZACION Y JOIN

    # Limpiar los nombres de las columnas eliminando espacios en blanco
    df1.columns = df1.columns.str.strip()
    df2.columns = df2.columns.str.strip()
```

- **Propósito**:
    - Elimina espacios en blanco de los nombres de las columnas para evitar problemas al acceder a ellas.

---

4. Verificación de las columnas necesarias

```
 # Verificar que las columnas necesarias existen en ambos DataFrames
    required_columns_df1 = [ 'IDProducto' ]
    required_columns_df2 = ['CodigoERP' ]

    missing_columns_df1 = [col for col in required_columns_df1 if col
not in df1.columns]
    missing_columns_df2 = [col for col in required_columns_df2 if col
not in df2.columns]

    if missing_columns_df1:
        actualizar_log(f"Error: Las siguientes columnas no se encuentran
en df1: {', '.join(missing_columns_df1)}")
        return

    if missing_columns_df2:
```

```
        actualizar_log(f"Error: Las siguientes columnas no se encuentran
en df2: {', '.join(missing_columns_df2)}")
        return
```

- **Propósito**:
    - Se asegura que las columnas necesarias estén presentes en ambos DataFrames (`IDProducto` en `df1` y `CodigoERP` en `df2`).
    - Si alguna de estas columnas falta, se registra un error y la función termina sin continuar.

---

5. Renombrado y combinación de los DataFrames

```
 # Renombrar la columna 'IDProducto' de df1 a 'CodigoERP'
    df2 = df2.rename(columns={'CodigoERP': 'IDProducto'})

    # Realizar el "merge" entre las dos tablas usando 'CodigoERP', con
un 'left join' para incluir filas de df1 que no están en df2
    merged_df = pd.merge(df1, df2[['IDProducto', 'IdItem']],
on='IDProducto', how='left')
```

- **Propósito**:
    - Renombra la columna `CodigoERP` en `df2` a `IDProducto` para hacerla consistente con `df1`.
    - Realiza un "merge" entre ambos DataFrames utilizando `IDProducto` como clave. Utiliza un "left join" para incluir todos los registros de `df1` aunque no haya una coincidencia en `df2`.

---

6. Reordenación de las columnas

```
 # Lista de columnas a mostrar en el orden especificado
    columnas_a_mostrar = [ ... ]  # Las columnas a mostrar en el archivo
final

    # Reordenar las columnas según la solicitud
    result = merged_df[columnas_a_mostrar]
```

- **Propósito**:

- Define un orden específico para las columnas en el DataFrame resultante, asegurando que se muestre según las necesidades de la aplicación.

---

7. Generación de nuevos IDs

```
# Crear un conjunto de todos los valores existentes en df2['IdItem']
para evitar duplicados
    existing_ids = set(df2['IdItem'].dropna().astype(int))

    # Identificar los índices donde 'idItem' está vacío
    empty_id_indices = result[result['IdItem'].isna()].index

    # Asignamos nuevos valores a las filas faltantes
    for index in empty_id_indices:
        codigo_interno = result.at[index, 'codigoInterno']

        if pd.notna(codigo_interno) and int(codigo_interno) < 8000:
            # Si codigoInterno es menor a 8000, asignamos ese valor a
IdItem
            result.at[index, 'IdItem'] = int(codigo_interno)
        else:
            # De lo contrario, asignamos un nuevo ID único
            result.at[index, 'IdItem'] = next(new_ids)
```

- **Propósito**:
  - Identifica las filas en las que el campo `IdItem` está vacío y genera nuevos IDs para esas filas.
  - Si el `codigoInterno` es menor a 8000, lo asigna como `IdItem`; si no, genera un nuevo ID único.

---

8. Preparación del archivo de salida

```
 # Carpeta de salida
    output_dir =
os.path.expanduser('~\\Documents\\PM-offer-updater\\processed-files\\Ite
ms')

    # Verificar si la carpeta existe, si no, crearla
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
```

```
    # Crear el nombre de archivo con la fecha de hoy
    fecha_hoy = datetime.today().strftime('%Y-%m-%d')
    output_file = os.path.join(output_dir, f"Items-{fecha_hoy}.csv")

    # Guardar el resultado
    result.to_csv(output_file, index=False)

    # Aviso del archivo creado
    actualizar_log(f"Archivo procesado y guardado en {output_file}")
    actualizar_log("---------- Proceso de normalizacion Terminado
----------")

    return output_file
```

- **Propósito**:
    - Crea una carpeta de salida si no existe, construye un nombre de archivo basado en la fecha actual y guarda el DataFrame procesado en un archivo CSV.
    - Registra la ubicación del archivo de salida en los logs.

Resumen

El archivo `normalizer.py` toma dos archivos de entrada (un CSV y un TXT), los procesa, verifica que tengan las columnas necesarias, realiza un "merge" entre ellos, genera nuevos IDs cuando es necesario y guarda el archivo combinado y normalizado en una ubicación específica. Además, registra las actividades y errores en los logs para facilitar el seguimiento de los procesos.

Explicación del archivo `quantio_barcode.py`

Este archivo está diseñado para procesar los códigos de barras de Quantio utilizando las funciones de otros módulos como `query_controller` y `file_controller`. A continuación, se describe su flujo de trabajo:

---

1. Importaciones y configuración del logger

```
from controllers.query_controller import quantio_updated_barcodes
from controllers.file_controller import guardar_resultados_como_csv
from ui.logs import get_logger
```

```
actualizar_log = get_logger()
```

- **`quantio_updated_barcodes`**: Esta función probablemente se encarga de obtener los datos actualizados de los códigos de barras.
- **`guardar_resultados_como_csv`**: Esta función probablemente guarda los resultados obtenidos de la consulta en un archivo CSV.
- **`get_logger`**: Se utiliza para obtener la función de logging y registrar las actividades del proceso.

---

2. Definición de las variables

```
file_path = "raw\\quantio\\barcodes"
name = "Barcodes"
```

- **`file_path`**: Especifica la ruta donde se guardará el archivo resultante.
- **`name`**: Define el nombre base del archivo de salida, en este caso, "Barcodes".

---

3. Función `process_file()`

Esta es la función principal que orquesta el procesamiento del archivo de códigos de barras de Quantio. A continuación, se describe su flujo:

```
def process_file():
    try:
        data = quantio_updated_barcodes()
        output_file = guardar_resultados_como_csv(data, file_path, name)
        actualizar_log("El archivo Barcodes Quantio se proceso
correctamente")
        return output_file
    except Exception as e:
        actualizar_log(f"Ocurrio un Error en el proceso de la consulta:
{e}")
        raise e
```

- **Propósito**:
    - **Obtención de datos**: Llama a la función `quantio_updated_barcodes()` para obtener los datos actualizados de los códigos de barras.

- **Guardado de los datos**: Llama a la función `guardar_resultados_como_csv()` para guardar esos datos en un archivo CSV en la ubicación especificada.
- **Log de éxito**: Si todo se ejecuta correctamente, se registra un mensaje indicando que el archivo se procesó con éxito.
- **Manejo de errores**: Si ocurre un error en el proceso, se captura la excepción y se registra el error en los logs.

Resumen

El archivo `quantio_barcode.py` se encarga de:

1. Consultar los códigos de barras actualizados de Quantio.
2. Guardar esos datos en un archivo CSV.
3. Registrar los mensajes de éxito o error en los logs.

Este archivo utiliza funciones externas para la obtención de los datos y su almacenamiento, y se asegura de manejar cualquier posible error durante el proceso.

Explicación del archivo `quantio_items.py`

El archivo `quantio_items.py` es similar al archivo `quantio_barcode.py`, pero está enfocado en el procesamiento de los productos de Quantio. A continuación, se desglosa su flujo de trabajo:

---

1. Importaciones y configuración del logger

```
from controllers.query_controller import quantio_updated_products
from controllers.file_controller import guardar_resultados_como_csv
from ui.logs import get_logger

actualizar_log = get_logger()
```

- **`quantio_updated_products`**: Esta función parece encargarse de obtener los productos actualizados de Quantio, posiblemente filtrados por un criterio de fecha.
- **`guardar_resultados_como_csv`**: Función para guardar los resultados obtenidos en un archivo CSV.
- **`get_logger`**: Se obtiene la función de logging para registrar mensajes de éxito o error.

---

2. Definición de las variables

```
file_path = "raw\\quantio\\items"
name = "Items"
```

- `file_path`: Define la ruta donde se guardará el archivo con los datos de los productos de Quantio.
- `name`: El nombre base para el archivo de salida, en este caso "Items".

---

3. Función `process_file(day_filter)`

Esta es la función principal que gestiona el procesamiento de los productos de Quantio. Se toma un filtro de fecha como parámetro:

```
def process_file(day_filter):
    try:
        data = quantio_updated_products(day_filter)
        output_file = guardar_resultados_como_csv(data, file_path, name)
        actualizar_log("El archivo Items Quantio se proceso
correctamente")
        return output_file
    except Exception as e:
        actualizar_log(f"Ocurrio un Error en el proceso de la consulta:
{e}")
        raise e
```

- **Propósito**:
  - **Obtención de datos**: Llama a la función `quantio_updated_products(day_filter)` pasando un filtro de fecha (probablemente para obtener productos actualizados en un día específico o dentro de un rango de fechas).
  - **Guardado de los datos**: Llama a la función `guardar_resultados_como_csv()` para guardar los datos obtenidos en un archivo CSV en la ubicación especificada.
  - **Log de éxito**: Si el proceso se realiza sin errores, se registra un mensaje indicando que el archivo de productos se procesó correctamente.
  - **Manejo de errores**: Si ocurre un error en cualquier parte del proceso, se captura la excepción y se registra en los logs con un mensaje de error detallado.

Resumen

El archivo `quantio_items.py` tiene como objetivo:

1. Consultar los productos actualizados de Quantio usando un filtro de fecha.
2. Guardar estos datos en un archivo CSV.
3. Registrar en los logs el éxito o el error durante el proceso.

Este archivo utiliza funciones externas para obtener los productos y guardar los resultados, asegurando que cualquier error que ocurra durante el proceso se registre adecuadamente.

Explicación de `queries.py`

El archivo `queries.py` contiene dos consultas SQL que se utilizan para obtener información sobre productos y códigos de barras desde una base de datos. Ambas consultas están diseñadas para ejecutarse en un sistema de base de datos (probablemente MySQL, dado el uso de funciones como `IFNULL` y `CURDATE()`).

1. Variables de configuración de código interno

```
cod1 = 'SET @codigoInterno1 = 1;'
cod2 = 'SET @codigoInterno2 = 8001;'
```

Estas variables inicializan dos contadores, `@codigoInterno1` y `@codigoInterno2`, que se utilizarán para generar códigos internos únicos para productos según su tipo o condiciones en la consulta SQL.

---

2. Consulta SQL: `Q_PRODUCTS`

```
Q_PRODUCTS = """
    SELECT
        CASE
            WHEN productos.idItem IN (5, 7) AND
LENGTH(LTRIM(productos.Codebar)) <= 5 THEN
                CAST(LTRIM(productos.Codebar) AS UNSIGNED)
            WHEN productos.idItem IN (5, 7) AND
LENGTH(LTRIM(productos.Codebar)) > 5 THEN
                (@codigoInterno1 := @codigoInterno1 + 1)
            ELSE
                (@codigoInterno2 := @codigoInterno2 + 1)
        END AS codigoInterno,
```

```
        ...
    FROM
        productos
        LEFT JOIN Subrubros
        ON productos.IDSubRubro = subrubros.IDSubRubro
    WHERE productos.Activo = 's'
    AND productos.FechaModificacion >= CURDATE() - %(day_filter)s
    AND productos.idTipoIVA != 1
    GROUP BY codigoInterno;
"""
```

Propósito:

- **Generación de código interno**:
    - Se utiliza una lógica condicional (`CASE`) para asignar un valor a `codigoInterno`.
    - Si el producto tiene un `idItem` de 5 o 7 y su `Codebar` es pequeño (5 caracteres o menos), se usa el valor de `Codebar`.
    - Si el `Codebar` es mayor a 5 caracteres, se incrementa el contador `@codigoInterno1`.
    - En otros casos, se incrementa el contador `@codigoInterno2`.
- **Campos seleccionados**:
    - Combina varios campos de la tabla `productos` y realiza cálculos, como el precio final del producto, considerando diferentes tipos de IVA y márgenes.
    - Incluye el nombre y descripción del producto (`Nom Reducido`), fechas, unidad de medida, entre otros.
- **Condiciones**:
    - La consulta filtra solo los productos activos (`productos.Activo = 's'`).
    - Los productos deben haber sido modificados en los últimos días (`productos.FechaModificacion >= CURDATE() - %(day_filter)s`), donde `day_filter` es un parámetro que se pasa al ejecutar la consulta.
    - Los productos con `idTipoIVA = 1` son excluidos.
- **Agrupación**:
    - Los resultados se agrupan por `codigoInterno`.

---

3. Consulta SQL: `Q_BARCODES`

```
Q_BARCODES = """
    SELECT productoscodebars.IDProducto AS IDProducto,
           IFNULL(productoscodebars.codebar, '0') AS Codebar
    FROM productoscodebars
    LEFT JOIN productos
    ON productos.IDProducto = productoscodebars.IDProducto
    WHERE productos.Activo = 's'
    UNION ALL

    SELECT productos.IDProducto AS IDProducto,
           IFNULL(productos.Codebar, '0') AS Codebar
    FROM productos
    WHERE productos.Activo = 's'
"""
```

Propósito:

- **Obtención de códigos de barras**:
    - La consulta obtiene los códigos de barras (`Codebar`) de los productos.
    - La primera parte de la consulta selecciona los códigos de barras desde la tabla `productoscodebars`, uniendo con la tabla `productos` para obtener los productos correspondientes.
    - La segunda parte obtiene directamente el código de barras de la tabla `productos` si no se encuentra en `productoscodebars`.
- **Uso de `IFNULL`**:
    - Se utiliza la función `IFNULL` para asegurarse de que, si un `Codebar` es nulo (no disponible), se reemplace por `'0'`.
- **Condiciones**:
    - Filtra solo los productos activos (`productos.Activo = 's'`).
    - Utiliza `UNION ALL` para combinar ambos conjuntos de resultados, asegurando que todos los códigos de barras sean incluidos.

Resumen

El archivo `queries.py` contiene dos consultas SQL esenciales para el proceso:

1. **`Q_PRODUCTS`**: Esta consulta obtiene información detallada sobre productos, como el código interno, nombre, precio final, etc., y aplica una lógica para generar códigos internos únicos.

2. **`Q_BARCODES`**: Esta consulta recoge los códigos de barras de los productos, considerando que algunos productos pueden no tener un código de barras asociado.

Ambas consultas están diseñadas para ser ejecutadas con un parámetro de filtro de fecha (`day_filter`), lo que permite obtener solo los productos modificados recientemente.

Explicación de `buttons.py`

El archivo `buttons.py` contiene funciones relacionadas con la creación y manejo de botones en una interfaz gráfica con Tkinter. A continuación, te explico lo que hace cada sección del código:

1. Importaciones

```
from tkinter import ttk
from ui.logs import get_logger
from controllers.process_controller import process  # Importa la función desde el orquestador
```

- **`ttk`**: Se importa desde Tkinter para usar widgets avanzados (como los botones estilizados).
- **`get_logger`**: Se importa para obtener una función que maneje los logs.
- **`process`**: Se importa desde `controllers.process_controller`, que parece ser la función encargada de orquestar el procesamiento de los archivos.

---

2. Función `procesar`

```
def procesar(entry_archivo2, entry_propuesta):
    file_path2 = entry_archivo2.get()
    file_propuesta = entry_propuesta.get()

    # Llamar al orquestador para procesar los archivos
    process(file_path2, file_propuesta)
```

- **Propósito**: Esta función es llamada cuando el usuario hace clic en el botón "Procesar Archivos".
- **Parámetros**: Recibe dos parámetros (`entry_archivo2` y `entry_propuesta`), que son entradas de texto en la interfaz (probablemente campos donde el usuario ingresa rutas de archivos).
- **Acción**:

- Obtiene las rutas de archivo desde las entradas de texto.
- Llama a la función `process` para procesar los archivos.

---

3. Función `crear_botones`

```
def crear_botones(root, entry_archivo2, entry_propuesta):
    # Botón para procesar
    button_procesar = ttk.Button(
        root,
        text="Procesar Archivos",
        command=lambda: procesar(entry_archivo2, entry_propuesta)
    )
    button_procesar.grid(row=4, column=0, columnspan=3, pady=20,
padx=10)

    return button_procesar
```

- **Propósito**: Esta función crea el botón "Procesar Archivos" en la interfaz gráfica y lo posiciona en el grid.
- **Parámetros**:
  - `root`: El contenedor principal de la ventana de Tkinter (probablemente la ventana principal de la interfaz).
  - `entry_archivo2` y `entry_propuesta`: Son las entradas de texto que contienen las rutas de archivo.
- **Acción**:
  - Crea un botón de la clase `ttk.Button` con el texto "Procesar Archivos".
  - Al hacer clic en el botón, se ejecuta la función `procesar(entry_archivo2, entry_propuesta)`, gracias al uso de `lambda`.
  - El botón se coloca en la posición especificada dentro del grid (`row=4, column=0`) y se le aplica algo de margen con `pady` y `padx`.

Resumen

- **`crear_botones`**: Crea el botón "Procesar Archivos" en la interfaz y lo vincula a la función `procesar` para ejecutar el procesamiento de archivos cuando se hace clic.
- **`procesar`**: Obtiene las rutas de archivo desde las entradas y llama al orquestador (`process`) para procesar los archivos.

Este código ayuda a integrar la lógica de procesamiento de archivos con la interfaz gráfica de usuario, permitiendo que el usuario interactúe fácilmente con la aplicación.

Explicación de `inputs.py`

El archivo `inputs.py` maneja la interfaz gráfica para seleccionar archivos en una aplicación de escritorio utilizando **Tkinter**. El código define las funciones necesarias para interactuar con el usuario, seleccionar archivos y llenar los campos correspondientes. Aquí tienes un resumen detallado de lo que hace cada sección del código:

1. Importaciones

```
import tkinter as tk
import os
from tkinter import ttk
from tkinter import filedialog
from ui.logs import get_logger
from ui.windows import ventana_query_quantio, config
from controllers.file_controller import read_query_config
```

- **`tkinter`**: La biblioteca estándar para interfaces gráficas en Python.
- **`os`**: Utilizada para interactuar con el sistema de archivos.
- **`ttk`**: Estilos mejorados para los widgets de Tkinter.
- **`filedialog`**: Un cuadro de diálogo para abrir archivos.
- **`get_logger`**: Función para registrar logs.
- **`ventana_query_quantio`** y **`config`**: Se importan desde `ui.windows`, probablemente para abrir una ventana de consulta y gestionar configuraciones.
- **`read_query_config`**: Una función importada, aunque no se usa en el código proporcionado.

---

2. Funciones para Seleccionar Archivos

`seleccionar_archivo_entrada2`

```
def seleccionar_archivo_entrada2(entry_archivo2):
    file_path = filedialog.askopenfilename(title="Seleccionar la lista
de Items de POSManager", filetypes=[("Archivos TXT", "*.txt"),
("Archivos CSV", "*.csv"), ("Archivos Excel", "*.xlsx;*.xls")])
    if file_path:
        entry_archivo2.delete(0, tk.END)
        entry_archivo2.insert(0, file_path)
        actualizar_log(f"Lista de Items de POSManager seleccionada:
{file_path}")
```

- **Propósito**: Permite al usuario seleccionar un archivo (TXT, CSV o Excel).
- **Acción**:
    - Muestra un cuadro de diálogo para seleccionar el archivo.
    - Si el archivo es seleccionado, el valor de la entrada `entry_archivo2` se actualiza con la ruta del archivo seleccionado y se registra en los logs.

`seleccionar_archivo_propuesta`

```
def seleccionar_archivo_propuesta(entry_propuesta):
    file_path = filedialog.askopenfilename(title="Seleccionar archivo de
Propuesta", filetypes=[("Archivos Excel", "*.xlsx;*.xls")])
    if file_path:
        entry_propuesta.delete(0, tk.END)
        entry_propuesta.insert(0, file_path)
        actualizar_log(f"Archivo de Propuesta seleccionado:
{file_path}")
```

- **Propósito**: Permite seleccionar un archivo Excel para la propuesta.
- **Acción**: Similar a la función anterior, pero restringida a archivos Excel.

`seleccionar_archivo_codebars`

```
def seleccionar_archivo_codebars(entry_codebars):
    file_path = filedialog.askopenfilename(title="Seleccionar archivo de
código de barras", filetypes=[("Archivos CSV", "*.csv"),("Archivos
Excel", "*.xlsx;*.xls")])
    if file_path:
        entry_codebars.delete(0, tk.END)
        entry_codebars.insert(0, file_path)
        actualizar_log(f"Archivo de código de barras seleccionado:
{file_path}")
```

- **Propósito**: Permite seleccionar archivos de códigos de barras (CSV o Excel).
- **Acción**: Similar a las funciones anteriores, con la diferencia de los tipos de archivo permitidos.

---

3. Función `crear_inputs`

```
def crear_inputs(root):
    default_propuesta =
os.path.expanduser('~\\Documents\\PM-offer-updater\\import\\Propuesta.xl
sx')
```

```
    # Inputs y botones para los archivos
    label_archivo1 = ttk.Label(root, text="Items actualizados
recientemente (Consulta a la base de datos):")
    label_archivo1.grid(row=0, column=0, padx=10, pady=10)

    label2_archivo1 = ttk.Label(root, text=f"Cantidad de dias
contemplados: {config['dias']}")
    label2_archivo1.grid(row=0, column=1, padx=10, pady=10)

    button_query = ttk.Button(root, text="Filtro", command=lambda:
ventana_query_quantio(root))
    button_query.grid(row=0, column=2, padx=10, pady=10)

    label_archivo2 = ttk.Label(root, text="Seleccionar la lista de Items
de POSManager (TXT delimitado por comas):")
    label_archivo2.grid(row=1, column=0, padx=10, pady=10)

    entry_archivo2 = ttk.Entry(root, width=50)
    entry_archivo2.grid(row=1, column=1, padx=10, pady=10)

    button_archivo2 = ttk.Button(root, text="Buscar", command=lambda:
seleccionar_archivo_entrada2(entry_archivo2))
    button_archivo2.grid(row=1, column=2, padx=10, pady=10)

    label_propuesta = ttk.Label(root, text="Seleccionar archivo de
Propuesta (Excel):")
    label_propuesta.grid(row=2, column=0, padx=10, pady=10)

    entry_propuesta = ttk.Entry(root, width=50)
    entry_propuesta.grid(row=2, column=1, padx=10, pady=10)

    button_propuesta = ttk.Button(root, text="Buscar", command=lambda:
seleccionar_archivo_propuesta(entry_propuesta))
    button_propuesta.grid(row=2, column=2, padx=10, pady=10)

    label_codebars = ttk.Label(root, text="Codigos de barras
actualizados (Consulta a la base de datos)")
    label_codebars.grid(row=3, column=0, padx=10, pady=10)

    entry_propuesta.insert(0, default_propuesta)

    return entry_archivo2, entry_propuesta
```

- **Propósito**: Crea los inputs y botones en la ventana principal.
- **Acción**:

- Define varias etiquetas (`ttk.Label`) y campos de entrada (`ttk.Entry`).
- Los botones permiten al usuario seleccionar archivos usando las funciones de selección definidas anteriormente.
- Inserta un valor por defecto en el campo `entry_propuesta` para la propuesta.
- Cada input y botón está organizado usando el método `grid` para definir su posición en la interfaz.

Resultado: Esta función configura la interfaz gráfica para que el usuario seleccione los archivos necesarios para el procesamiento.

Resumen General

- Este archivo crea una interfaz gráfica para seleccionar archivos y configurar ciertos parámetros.
- Permite seleccionar archivos de diferentes tipos (TXT, CSV, Excel) mediante cuadros de diálogo.
- Se actualizan los campos de entrada con las rutas de los archivos seleccionados y se registra la acción en los logs.

El archivo `logs.py` gestiona el sistema de logging para la aplicación de escritorio utilizando **Tkinter**. A continuación te explico cómo funciona:

1. Variables Globales

```
log_area = None
actualizar_log = None
```

- Estas son variables globales que se usan para almacenar el área de texto donde se mostrarán los logs (`log_area`) y la función para actualizar los logs (`actualizar_log`).

2. Clase `PrintRedirector`

```
class PrintRedirector:
    """
    Redirige la salida de `print` a un widget de texto.
    """
    def __init__(self, widget):
        self.widget = widget
```

```
    def write(self, message):
        if message.strip():  # Evitar mensajes vacíos
            self.widget.insert(tk.END, message + '\n')
            self.widget.yview(tk.END)

    def flush(self):
        pass
```

- **Propósito**: Esta clase redirige cualquier salida de `print` hacia un widget de texto en la interfaz gráfica. Esto es útil si quieres mostrar los mensajes de consola en la ventana de la aplicación.
- **Método `write`**: Este método maneja el mensaje que se pasa a `print`. Lo inserta en el `Text` widget (`log_area`) y hace que el área de texto se desplace hacia el final para mostrar el mensaje más reciente.
- **Método `flush`**: Este método no hace nada, ya que no es necesario en este caso (es parte de la interfaz de `file-like` objects).

3. Función `configurar_logger`

```
def configurar_logger(root):
    """
    Configura el logger global y define las funciones necesarias.
    """
    global log_area, actualizar_log

    # Crear el área de logs
    log_area = tk.Text(root, height=10, width=100, wrap=tk.WORD,
state=tk.DISABLED)
    log_area.grid(row=5, column=0, columnspan=3, pady=10)

    # Definir la función para actualizar los logs
    def log(message):
        try:
            log_area.config(state=tk.NORMAL)
            log_area.insert(tk.END, message + '\n')
            log_area.yview(tk.END)
            log_area.config(state=tk.DISABLED)
            root.update()
        except:
            tk.TclError
            pass

    # Asignar la función al logger global
```

```
    actualizar_log = log
```

- **Propósito**: Esta función configura el sistema de logging en la interfaz gráfica.
- **Acción**:
    - Crea un widget de texto (`log_area`) para mostrar los mensajes de log. Este widget está inicialmente en modo `DISABLED` para evitar que el usuario edite el área de texto.
    - Define una función interna `log`, que se encarga de insertar mensajes en el widget de texto y mantenerlo actualizado.
    - La función `log` cambia el estado del widget de `DISABLED` a `NORMAL` para insertar el mensaje, luego vuelve a `DISABLED` para evitar que el usuario modifique el texto.
    - La función `root.update()` asegura que la interfaz gráfica se actualice después de insertar el mensaje.
    - Asigna la función `log` a la variable global `actualizar_log`.

4. Función `get_logger`

```
def get_logger():
    """
    Devuelve la función `actualizar_log` para usar en otros módulos.
    """
    if actualizar_log is None:
        raise RuntimeError("El logger no ha sido configurado. Llama a `configurar_logger` primero.")
    return actualizar_log
```

- **Propósito**: Esta función devuelve la función global `actualizar_log`, que es utilizada en otros módulos para escribir en el área de logs.
- **Verificación**: Si `actualizar_log` no ha sido configurado previamente (es decir, si `configurar_logger` no ha sido llamada), se lanza un `RuntimeError` para evitar que el código intente usar el logger sin configurar.

Resumen General

El archivo **`logs.py`** proporciona una funcionalidad para mostrar logs de la aplicación en un widget de texto de **Tkinter**.

- Usa la clase `PrintRedirector` para redirigir la salida de `print` a este widget.
- La función `configurar_logger` configura este sistema de logging en la interfaz.

- La función `get_logger` permite acceder al logger desde otros módulos, garantizando que los logs se registren correctamente en la ventana de la aplicación.

Este sistema es útil para monitorear el flujo de trabajo de la aplicación, mostrando información o errores importantes directamente en la interfaz gráfica.

Explicación de `windows.py`

El archivo `windows.py` define una ventana secundaria de configuración, que permite al usuario seleccionar una fecha y guardar un valor basado en esa selección. Aquí está la descripción detallada de cada sección del código:

1. Importaciones

```
import tkinter as tk
from tkinter import ttk, Frame
from tkcalendar import DateEntry
from datetime import datetime
from controllers.file_controller import save_query_config,
read_query_config
from ui.logs import get_logger
```

- Se importan las bibliotecas necesarias para la creación de la interfaz gráfica con **Tkinter**, el calendario con **`tkcalendar`**, y el manejo de fechas con **`datetime`**.
- **`save_query_config`** y **`read_query_config`** son funciones para guardar y leer la configuración del archivo de consulta.
- **`get_logger`** permite obtener el logger para registrar mensajes.

2. Función `ventana_query_quantio`

Esta función crea una ventana secundaria (`Toplevel`) para que el usuario ingrese y guarde una configuración de fecha.

Definición de la ventana y sus elementos:

```
def ventana_query_quantio(root):
```

- **`ventana_query_quantio`** es la función principal que genera la ventana secundaria donde el usuario interactúa.

Función `save_config`: Guarda la configuración de la fecha seleccionada.

```
def save_config():
    global config
    # Obtener la fecha seleccionada
    fecha_seleccionada = str(calendar.get_date())

    # Convertir la fecha seleccionada a un objeto datetime
    fecha_seleccionada = datetime.strptime(fecha_seleccionada,
'%Y-%m-%d')

    # Obtener la fecha actual
    fecha_actual = datetime.now()

    # Calcular la diferencia en días
    dias = (fecha_actual - fecha_seleccionada).days

    # Guardar la cantidad de días en una variable
    configuracion = {
        'dias': dias
    }

    save_query_config(configuracion)
```

- **`save_config`** obtiene la fecha seleccionada por el usuario a través del `DateEntry`, la convierte a un objeto `datetime`, calcula la diferencia en días con respecto a la fecha actual, y guarda este valor en la configuración utilizando `save_query_config`.

Creación de la ventana `ventana_query_quantio`:

```
ventana_query_quantio = tk.Toplevel(root)
ventana_query_quantio.title("Query Quantio")
ventana_query_quantio.geometry("400x400")
```

- Crea una nueva ventana secundaria (`Toplevel`), con un título "Query Quantio" y un tamaño de 400x400 píxeles.

Frames para organizar los widgets:

```
frame_buttons = Frame(ventana_query_quantio)
frame_buttons.pack(side="bottom", anchor="se", fill="x")

frame_top_label = Frame(ventana_query_quantio)
frame_top_label.pack(side="top", anchor="w", fill="x")
```

```
frame_filters = Frame(ventana_query_quantio)
frame_filters.pack(side="top", fill="both")

sub_frame_date = Frame(frame_filters)
sub_frame_date.pack(side="top", fill="x")
```

- Los **frames** se utilizan para organizar los widgets dentro de la ventana. Esto hace que la disposición de los elementos sea más flexible.
- **`frame_buttons`** contiene los botones.
- **`frame_top_label`** contiene el título de la ventana.
- **`frame_filters`** contiene el campo para la selección de la fecha.
- **`sub_frame_date`** es un subframe dentro de `frame_filters`, específicamente para colocar el calendario de selección de fecha.

Widgets en la ventana secundaria:

```
label = tk.Label(frame_top_label, text="Control de Query")
label.pack(side="left", pady=10, padx=10)

label_date = tk.Label(sub_frame_date, text="Selecciona una fecha:")
label_date.pack(side="left", padx=10, pady=5)

calendar = DateEntry(sub_frame_date, date_pattern="mm/dd/yyyy",
width=12, maxdate=datetime.now())
calendar.pack(side="left", pady=10, padx=10)
```

- **Etiqueta de control de la query**: Muestra un texto en la parte superior.
- **Etiqueta de fecha**: Describe el propósito del campo de fecha.
- **`DateEntry`**: Un campo de calendario que permite al usuario seleccionar una fecha. El formato es **`mm/dd/yyyy`** y la fecha máxima es la fecha actual.

Botones:

```
guardar_button = ttk.Button(frame_buttons, text="Guardar",
command=lambda: save_config())
guardar_button.pack(side="right", pady=10, padx=10)

cerrar_button = ttk.Button(frame_buttons, text="Cerrar", command=lambda:
ventana_query_quantio.destroy())
cerrar_button.pack(side="right", pady=10, padx=10)
```

- **Botón "Guardar"**: Llama a `save_config()` para guardar la cantidad de días calculada.
- **Botón "Cerrar"**: Cierra la ventana secundaria cuando se hace clic.

Retorno de la función:

```
return ventana_query_quantio
```

- La función retorna el objeto de la ventana secundaria creada (`ventana_query_quantio`), lo que permite interactuar con ella desde otras partes del código si es necesario.

Resumen

La función **`ventana_query_quantio`** crea una ventana secundaria con los siguientes componentes:

- Un calendario (`DateEntry`) para que el usuario seleccione una fecha.
- Un botón para guardar la diferencia en días entre la fecha seleccionada y la fecha actual.
- Un botón para cerrar la ventana secundaria.

El valor de los días calculado se guarda en un archivo de configuración mediante la función `save_query_config`.

Este sistema permite configurar la cantidad de días que se utilizará en las consultas posteriores.

Flujo General del Proyecto Modificado

---

1. Selección de Archivos por el Usuario (Interfaz Gráfica)

El proceso comienza cuando el usuario selecciona los archivos que contienen los datos a procesar. Esta operación se realiza en la interfaz gráfica utilizando Tkinter.

- **Archivo de Items (POSManager)**: El usuario selecciona un archivo que contiene los items de POSManager (puede ser un archivo `.txt`, `.csv` o `.xlsx`).
- **Archivo de Propuesta**: El usuario selecciona el archivo de propuesta, generalmente un archivo Excel (.xlsx o .xls).
- **Archivo de Códigos de Barras**: **Ya no es necesario** que el usuario seleccione este archivo. En lugar de eso, los códigos de barras se consultan directamente desde la base de datos.

Código relacionado:

- En `inputs.py`, la función `seleccionar_archivo_codebars` ya no es necesaria, por lo que debería ser eliminada.

---

2. Configuración del Filtro de Fechas

El siguiente paso sigue siendo configurar un filtro de fechas. El usuario puede seleccionar una fecha en la ventana de configuración, que determinará el rango de días a considerar para las consultas a la base de datos. Este valor se calcula como la diferencia entre la fecha seleccionada y la fecha actual.

Código relacionado:

- En `windows.py`, la función `ventana_query_quantio` permite al usuario seleccionar una fecha desde un calendario y guarda la cantidad de días en el archivo de configuración (usando la función `save_config`).
- Esta cantidad de días se lee en `config['dias']` y se utiliza en las consultas a la base de datos.

---

3. Consultas a la Base de Datos

Una vez que el usuario ha configurado los filtros y seleccionado los archivos necesarios, se realizan las consultas a la base de datos para obtener los datos correspondientes.

- **Consulta de Productos (`Q_PRODUCTS`)**: Se realiza una consulta a la base de datos para obtener información sobre los productos, como códigos internos, precios, descripciones, y otros detalles relevantes. Esta consulta incluye un filtro para los productos que han sido modificados dentro del rango de días especificado por el usuario (`config['dias']`).

- **Consulta de Códigos de Barras (`Q_BARCODES`)**: **Ya no es necesario seleccionar un archivo de códigos de barras.** Ahora, esta consulta se ejecuta directamente sobre la base de datos para obtener los códigos de barras asociados a los productos.

Código relacionado:

- En `queries.py`, las consultas SQL están definidas bajo las variables `Q_PRODUCTS` y `Q_BARCODES`. La consulta de códigos de barras (`Q_BARCODES`) ahora se ejecuta directamente sobre la base de datos, sin necesidad de cargar un archivo.

---

4. Normalización y Cálculos

Los datos obtenidos de las consultas a la base de datos se normalizan y se les aplican cálculos para ajustarlos a la estructura y formato necesario para el archivo de salida.

- **Normalización**: En las consultas SQL, se utilizan funciones como `IFNULL` para manejar valores nulos y asegurar que los datos estén completos y sean coherentes. Además, se realiza un ajuste en la longitud de los códigos de barras y otros campos que podrían variar.

- **Cálculos**: Se calculan campos como el precio final de los productos, tomando en cuenta el costo, el IVA y los márgenes. Esto se realiza dentro de las consultas SQL, específicamente en la columna `precio_final` de `Q_PRODUCTS`.

Código relacionado:

- En `queries.py`, la consulta de productos (`Q_PRODUCTS`) aplica los cálculos relacionados con los precios finales, tomando en cuenta los valores de IVA y los márgenes de ganancia.

---

5. Guardado de los Resultados

Una vez que los datos han sido procesados (normalizados y calculados), se guardan en un archivo en formato CSV o Excel.

- **Guardar Archivos CSV**: Los datos procesados se guardan en archivos CSV, con una ruta y nombre específicos. Los archivos generados son los archivos de productos y códigos de barras actualizados, como los llamados "Items" y "Barcodes" en la carpeta de salida.

Código relacionado:

- En `quantio_items.py` y `quantio_barcode.py`, las funciones `process_file` manejan la generación de los archivos CSV utilizando la función `guardar_resultados_como_csv`, que guarda los datos procesados en la ubicación especificada.
- Los resultados de las consultas a la base de datos son pasados como parámetros a la función `guardar_resultados_como_csv`, que se encarga de formatear y guardar los datos en un archivo CSV.

Ruta de los archivos:

- Los archivos resultantes se guardan en una carpeta específica definida en `file_path` (por ejemplo, `"raw\\quantio\\items"` para los items y `"raw\\quantio\\barcodes"` para los códigos de barras).

---

6. Registro de Logs

Durante todo el proceso, se actualizan los logs para informar al usuario sobre el progreso y cualquier error que ocurra.

- **Registro de Logs**: A lo largo del flujo, se registran mensajes en el área de logs de la interfaz gráfica para que el usuario vea el estado de la ejecución y los errores si los hay.

Código relacionado:

- En `logs.py`, la función `configurar_logger` configura el área de logs donde se muestran los mensajes, y `get_logger` permite acceder a la función `actualizar_log` desde otras partes del código para registrar mensajes.
- La función `actualizar_log` es llamada en varias partes del proyecto, por ejemplo, cuando un archivo es seleccionado, cuando el proceso de la base de datos es exitoso, o cuando ocurre un error.

Resumen del Flujo Modificado

1. **Selección de Archivos**: El usuario selecciona solo los archivos de entrada (Items y Propuesta). Los códigos de barras ya no se seleccionan, sino que se consultan directamente desde la base de datos.
2. **Configuración del Filtro de Fechas**: El usuario selecciona una fecha y el sistema calcula la cantidad de días para filtrar los productos modificados.
3. **Consultas a la Base de Datos**: Se ejecutan las consultas SQL para obtener los productos y códigos de barras actualizados dentro del rango de días especificado. La consulta de códigos de barras se realiza directamente en la base de datos.
4. **Normalización y Cálculos**: Se normalizan los datos y se realizan los cálculos necesarios (como el precio final).
5. **Guardado de Archivos**: Los datos procesados se guardan en archivos CSV en las ubicaciones específicas.
6. **Registro de Logs**: Se mantienen registros del progreso y errores durante el proceso.

Este flujo actualizado elimina la necesidad de seleccionar un archivo de códigos de barras y mejora la eficiencia al realizar la consulta de códigos de barras directamente desde la base de datos.