TUP'23 - PROYECTO INTEGRADOR

Equipo CodiPro

Integrantes:

- Quinteros, Mariana
- Bazán Azargado, Ma. Emilia
- Torres Urzua, Siro Ezequiel
- Esteche, Yamila Eliana
- Balmaceda, Adriana Verónica
- Martinez, Fabricio
- Juan Beresiarte

Sistema de pago de la tienda "Hola Mundo Animal"

A continuación, se realiza una explicación detallada de la ejecución del código y las instrucciones de uso de la base de datos SQL.

Estructura del código

El código se divide en varias secciones:

1. Definición de modelos de datos (clases `Producto` y `Compra`)
2. Definición de una clase `Query` para interactuar con la base de datos
3. Definición de una clase `TypedQuery` que hereda de `Query` y agrega métodos específicos para interactuar con los modelos de datos
4. Definición de funciones auxiliares para realizar operaciones específicas (e.g. elegir código postal, agregar producto, validar tarjeta de débito)
5. Definición de la función `main` que inicia el programa y llama a las funciones auxiliares
6. Definición de la función `init_db` que crea la base de datos y carga datos iniciales

Ejecución del código

1. La función `main` se llama cuando se ejecuta el programa. Primero, crea una ventana principal con un título y un botón "ACEPTAR".
2. Cuando se presiona el botón "ACEPTAR", se cierra la ventana principal y se llama a la función `elegir_codigo_postal`.

3. La Funcion `elegir_codigo_postal` muestra una ventana emergente que pide al usuario que ingrese un código postal comparándolo con los códigos postales disponibles dentro del sistema de envío. Dichos códigos pueden ser modificados desde la matriz al igual que los precios de cada uno. Como detalle especial, las zonas de San Rafael, Gral Alvear etc no tienen costo de envío agregado por una cortesía de la casa por la cercanía ficticia del local fisico con dichas localidades, dando un ejemplo para zonas que tienen la posibilidad de no tener un precio agregado de envío y el codigo puede continuar perfectamente.Si el codigo postal es valido se muestra un mensaje con el costo de envio a traves de un if que declara el código postal como válido. En caso de un error se encierra dentro de un ciclo que permite al usuario volver a ingresar el código postal y en caso de cerrar se muestra al usuario un mensaje invitando a visitar el local de manera presencial para retirar su producto. En caso de que se ingrese un codigo postal valido el precio se guarda en la variable costo_envio para posteriormente ser sumada al total del pago del producto.
4. Luego, se llama a la función `agregar_producto`, que muestra una lista de productos disponibles y pide al usuario que seleccione uno. Si el producto es seleccionado, se muestra un mensaje con el precio del producto y se guarda en la variable `precio_del_producto`.
5. A continuación, se llama a la función `validar_tarjeta_debito`, que pide al usuario que ingrese los datos de su tarjeta de débito. Si los datos son válidos, se muestra un mensaje de confirmación y se guarda la compra en la base de datos.
6. Finalmente, se cierra la sesión de la base de datos.

Instrucciones de uso de la base de datos SQL

La base de datos se crea utilizando la biblioteca `sqlalchemy` y se configura para utilizar una base de datos SQLite. La base de datos tiene dos tablas: `productos` y `compras`.

Tabla `productos`

- `id`: clave primaria, autoincremental
- `nombre`: nombre del producto
- `precio`: precio del producto
- `cantidad_disponible`: cantidad disponible del producto
- `cantidad_comprada`: cantidad comprada del producto

Tabla `compras`

- `id`: clave primaria, autoincremental
- `nombre_cliente`: nombre del cliente
- `direccion_cliente`: dirección del cliente
- `localidad`: localidad del cliente
- `costo_total`: costo total de la compra
- `pago_realizado`: indica si el pago se realizó con éxito
- `producto_id`: clave foránea que se refiere a la tabla `productos`

La base de datos se crea utilizando la función `init_db`, que crea las tablas y carga datos iniciales. La función `main` utiliza la clase `TypedQuery` para interactuar con la base de datos y realizar operaciones de lectura y escritura.

Observaciones y propuestas de mejora sobre el Proyecto:

Observaciones:

1. La base de datos se crea en memoria, por lo que los datos se perderán cuando se cierra el programa.
2. La función `init_db` carga datos iniciales en la base de datos, pero no proporciona una forma de agregar nuevos productos o editar los existentes.
3. La función `validar_tarjeta_debito` no realiza una validación real de la tarjeta de débito, solo pide al usuario que ingrese los datos y los guarda en la base de datos.
4. El código utiliza la biblioteca `tkinter` para crear una interfaz gráfica de usuario, pero no proporciona una forma de interactuar con la interfaz.

Propuestas casos observaciones:

1. En este caso, al usar SQLite como base de datos, los datos se almacenan en un archivo en el disco duro en lugar de en la memoria. De hecho, la cadena de conexión `sqlite:///database.db` indica que se creará un archivo `database.db` en el directorio actual donde se ejecuta el programa.

Por lo tanto, los datos no se perderán cuando se cierra el programa, ya que se almacenan en un archivo en el disco duro.

Sin embargo, si deseamos que los datos se conserven incluso después de que se cierra el programa, deberemos asegurarnos de que el archivo database.db no se elimine accidentalmente.

Una propuesta podría ser:

Configuración de la ruta del archivo de base de datos: se puede especificar una ruta absoluta para el archivo de base de datos, por ejemplo: `sqlite:////path/to/database.db`. De esta manera, el archivo se creará en la ruta especificada y no se eliminará cuando se cierre el programa.

Crear un archivo de configuración: se puede crear un archivo de configuración que almacene la ruta del archivo de base de datos y otros parámetros de configuración. De esta manera, se podría cargar la configuración al iniciar el programa y asegurarnos de que la base de datos se cree en la ruta correcta.

Utilizar un sistema de archivos persistente: se puede utilizar un sistema de archivos persistente como un servicio de almacenamiento en la nube (como Google Cloud Storage o Amazon S3) o un sistema de archivos en red (como NFS) para almacenar el archivo de base de datos. De esta manera, los datos se conservarán incluso si el programa se cierra o se reinicia.

Deberíamos tener en cuenta que: si se desea utilizar una base de datos más robusta y escalable, como MySQL o PostgreSQL, se debe configurar una conexión a una base de datos externa en lugar de utilizar SQLite.

2. Para agregar la funcionalidad de agregar nuevos productos o editar los existentes a la función `init_db`, se puede modificar de la siguiente manera:

```
def init_db():

    # Si hay productos dentro de la base de datos no se ejecuta la función
```

```
if len(query.obtener_productos()) > 0:
    print("Usando base de datos anterior...")
    return None

print("Base de datos: Creada!")

# Agregar productos iniciales
productos = [
    Producto("Kit descanso perro chico", 3000, 10, 0),
    Producto("Kit descanso perro grande", 4000, 10, 0),
    Producto("Kit baño perro chico", 4000, 13, 0),
    Producto("Kit baño perro grande", 4500, 20, 0),
    Producto("Kit pasea perro chico", 5000, 5, 0),
    Producto("Kit paseo perro grande", 5500, 4, 0),
]

for producto in productos:
    query.add(producto)

# Agregar opción para agregar nuevos productos
while True:
```

```
        respuesta = messagebox.askyesno("Agregar productos", "¿Desea agregar
nuevos productos?")

        if not respuesta:

            break



        nombre_producto = simpledialog.askstring("Nombre del producto",
"Ingrese el nombre del producto")

        precio_producto = float(simpledialog.askstring("Precio del producto",
"Ingrese el precio del producto"))

        cantidad_disponible = int(simpledialog.askstring("Cantidad
disponible", "Ingrese la cantidad disponible del producto"))



        nuevo_producto = Producto(nombre_producto, precio_producto,
cantidad_disponible, 0)

        query.add(nuevo_producto)



    # Agregar opción para editar productos existentes

    while True:

        respuesta = messagebox.askyesno("Editar productos", "¿Desea editar
productos existentes?")

        if not respuesta:

            break



        productos_existentes = query.obtener_productos()
```

```
        lista_productos = "\n".join([f"{i}. {producto.nombre}" for i,
producto in enumerate(productos_existentes)])



        producto_index = simpledialog.askstring("Editar producto",
"Seleccione el producto a editar:\n" + lista_productos)

        if producto_index is None:

            break



        producto_index = int(producto_index)

        producto_a_editar = productos_existentes[producto_index]



        nombre_producto = simpledialog.askstring("Nombre del producto",
"Ingrese el nuevo nombre del producto",
initialvalue=producto_a_editar.nombre)

        precio_producto = float(simpledialog.askstring("Precio del producto",
"Ingrese el nuevo precio del producto",
initialvalue=str(producto_a_editar.precio)))

        cantidad_disponible = int(simpledialog.askstring("Cantidad
disponible", "Ingrese la nueva cantidad disponible del producto",
initialvalue=str(producto_a_editar.cantidad_disponible)))



        producto_a_editar.nombre = nombre_producto

        producto_a_editar.precio = precio_producto

        producto_a_editar.cantidad_disponible = cantidad_disponible



        query.update()
```

Esta modificación agrega dos opciones adicionales a la función `init_db`:

1. Agregar nuevos productos: se pregunta al usuario si desea agregar nuevos productos, y si es así, se le pide ingresar los datos del producto (nombre, precio y cantidad disponible).
2. Editar productos existentes: se pregunta al usuario si desea editar productos existentes, y si es así, se le pide seleccionar el producto a editar y luego ingresar los nuevos datos del producto.

Ambas opciones utilizan la función `simpledialog` para obtener la entrada del usuario y la función `query` para agregar o editar los productos en la base de datos.

3. Para agregar una validación real, debemos utilizar una biblioteca que permita verificar la tarjeta de débito, como `python-creditcard`.

Primero, debemos instalar la biblioteca `python-creditcard` utilizando pip:

```
pip install python-creditcard
```

Luego, podemos modificar la función validar_tarjeta_debito para que realice la validación:

```
import creditcard



def validar_tarjeta_debito():
    messagebox.showinfo(
        "Información", "Recuerde que solo puede pagar con tarjeta de débito"
    )
    precio_final = compra_actual.get_costo_total()
```

```
    messagebox.showinfo("Total a pagar", f"El monto total a pagar es: ${precio_final}")



    while True:

        numero_tarjeta = simpledialog.askstring(

            "Número de tarjeta",

            "Ingrese el número de su tarjeta de débito (deben ser 16 dígitos): ",

        )

        if numero_tarjeta is None:  # Si se presiona "Cancelar"

            mostrar_mensaje_despedida()

            return



        if not creditcard.validate_card_number(numero_tarjeta):

            messagebox.showerror(

                "Error",

                "El número de la tarjeta es inválido. Por favor, inténtelo nuevamente.",

            )

            continue



        while True:
```

```
            codigo_seguridad = simpledialog.askstring(
                "Código de seguridad",
                "Ingrese el código de seguridad de su tarjeta
(son los 3 dígitos que se encuentran en la parte posterior de su
tarjeta): ",
            )
            if codigo_seguridad is None:  # Si se presiona
"Cancelar"
                mostrar_mensaje_despedida()
                return

            if not
creditcard.validate_card_security_code(codigo_seguridad):
                messagebox.showerror(
                    "Error", "El código de seguridad es inválido.
Por favor, inténtelo nuevamente."
                )
                continue

            break

        while True:
            try:
```

```
                mes = int(
                    simpledialog.askstring(
                        "Mes de vencimiento",
                        "Ingrese el mes de vencimiento de su
tarjeta (MM): ",
                    )
                )
                anio = int(
                    simpledialog.askstring(
                        "Año de vencimiento",
                        "Ingrese el año de vencimiento de su
tarjeta en formato (AA): ",
                    )
                )
            except ValueError:
                messagebox.showerror(
                    "Error", "El mes y año de vencimiento deben
ser números enteros"
                )
                continue
```

```
            if mes is None or anio is None:  # Si se presiona
"Cancelar"
                mostrar_mensaje_despedida()
                return

            if not creditcard.validate_card_expiration_date(mes,
anio):
                messagebox.showerror(
                    "Error", "La fecha de vencimiento es
inválida. Por favor, inténtelo nuevamente."
                )
                continue

            break

        compra_actual.set_pago_realizado(True)

        messagebox.showinfo(
            "TIENDA HOLA MUNDO ANIMAL",
            "Finalizado, pago realizado exitosamente ¡Gracias por
su compra!",
        )
```

En este caso, se utiliza la biblioteca `creditcard` para validar el número de tarjeta, el código de seguridad y la fecha de vencimiento. Si alguno de estos campos es inválido, se muestra un mensaje de error y se pide al usuario que ingrese los datos nuevamente.

4. En este caso se propone un código que utiliza la biblioteca `tkinter` para crear una interfaz gráfica de usuario (GUI) e interactuar con una base de datos SQL utilizando la biblioteca `sqlalchemy`.

A continuación, damos una explicación detallada de cómo funcionaría el código con esta propuesta:

Interfaz gráfica de usuario (GUI)

La GUI se crea utilizando la biblioteca `tkinter`. Se define una ventana principal (`ventana_principal`) con un título y un tamaño específico. Dentro de esta ventana, se crean etiquetas (`Label`) y botones (`Button`) para interactuar con el usuario.

Base de datos

La base de datos se crea utilizando la biblioteca `sqlalchemy`. Se define una clase `Base` que hereda de `DeclarativeBase`, que es la clase base para definir modelos de datos en `sqlalchemy`. Luego, se definen dos modelos de datos: `Producto` y `Compra`, que representan las tablas de la base de datos.

Interacción con la base de datos

La interacción con la base de datos se realiza utilizando la clase `TypedQuery`, que es una clase que hereda de `Query`. Esta clase proporciona métodos para interactuar con la base de datos, como `get_all`, `get_by_id`, `add`, `update` y `delete`.

Flujo del programa

El flujo del programa es el siguiente:

1. Se crea la ventana principal y se muestra al usuario.
2. El usuario hace clic en el botón "ACEPTAR" y se cierra la ventana principal.
3. Se llama a la función `elegir_codigo_postal`, que permite al usuario seleccionar un código postal y obtener el costo de envío correspondiente.
4. Se llama a la función `agregar_producto`, que permite al usuario seleccionar un producto y obtener su precio.
5. Se llama a la función `validar_tarjeta_debito`, que permite al usuario ingresar sus datos de tarjeta de débito y realizar el pago.
6. Se registra la compra en la base de datos utilizando la función `registrar_compra`.

Observaciones

- La base de datos del sistema de pago del proyecto se crea utilizando SQLite, pero se puede cambiar a otra base de datos como MySQL si se desea.
- La GUI se crea utilizando `tkinter`, pero se puede cambiar a otra biblioteca como `PyQt` o `wxPython` si se desea utilizar.
- El código utiliza una sesión global para interactuar con la base de datos, lo que puede ser un problema si se necesita manejar varias sesiones simultáneamente.

  `Propuesta para este caso:`

  El código utiliza una sesión global para interactuar con la base de datos, lo que puede ser un problema si se necesita manejar varias sesiones. Una posible solución es crear una clase que maneje la sesión y la conexión a la base de datos, de manera que se pueda crear una instancia de esa clase cada vez que se necesite una sesión.

  Éste es un ejemplo de cómo se podría refactorizar el código:

  ```
  class DatabaseSession:
  ```

```
    def __init__(self, engine):
        self.engine = engine
        self.SessionMaker = sessionmaker(bind=engine)
        self.session = self.SessionMaker()

    def get_session(self):
        return self.session

    def close_session(self):
        self.session.close()

# ...

if __name__ == "__main__":
    engine = create_engine("sqlite:///database.db")
    Base.metadata.create_all(engine)

    db_session = DatabaseSession(engine)
    session = db_session.get_session()
    query = TypedQuery(session)
```

```
# ...



try:

    # ...

finally:

    db_session.close_session()
```

De esta manera, cada vez que se necesite una sesión, se crea una instancia de la clase DatabaseSession y se obtiene la sesión mediante el método `get_session()`.

Al final, se cierra la sesión con el método `close_session()`.

También se podría considerar utilizar un patrón de diseño como el de "Unidad de Trabajo" para manejar las sesiones y las transacciones de manera más elegante y segura.