

# Curso Python 3 – Nivel Inicial

## Informe del Trabajo Practico Final

Realizado por:

- Sebastian Orellana (DNI: 39404862)

Docentes y coordinadores:

- Juan Marcelo Barreto Rodriguez
- Gabriela Verónica Aquino
- Brenda Abigail Barreto Aquino

Fecha de Realización: 22/12/2022

## Desarrollo, objetivo y alcance de la aplicación:

La aplicación que se intenta desarrollar tiene como objetivo ser una herramienta que permita a alguna empresa mayorista, o a alguna empresa que realice encargos, poder administrar pedidos y guardar dicha información en una base de datos. Si bien la aplicación se pensó con el objetivo de que funcione inicialmente para una ferretería, esta puede extenderse a cualquier otro comercio que utiliza el formato de Código, Nombre, Cantidad y Precio para sus productos. Funcionalmente, lo que se busca en la aplicación, es mostrar la información del cliente en todo momento por lo que será necesario ajustar el programa de tal forma que, al abrirlo, se muestren los datos cargados en el caso de que existan, y además de que se actualice la información que ve el usuario cada vez que este cargue un artículo nuevo, borre o modifique alguno ya existente de tal manera de que la información que el usuario ve en todo momento sea actualizada. Además, la aplicación contara con la opción de guardar datos, de tal manera de que toda la información que el usuario cargue, borre o modifique dentro del programa no se guarde hasta que el usuario pulse dicho botón. De tal manera se evita que se guarden datos que fueron cargados, borrados o modificados erróneamente. El guardado únicamente sucederá una vez que el usuario haya confirmado que los datos en la aplicación son los que realmente quiere guardar y por lo tanto haya pulsado el botón de “Guardar Datos”.

Con respecto al boceto original presentado en el curso, se agregó la opción para crear y borrar un cliente (creación y borrado de tablas en la base de datos). Además, se modificaron las opciones de carga y modificación de datos, ya que no se permitía realizar estas opciones en ventanas secundarias. También, se optimizó el espacio de la aplicación, se borró la opción de “Carga Manual” ya que esta opción es la misma que “Carga” y se cambió el botón de Imprimir por el de “Guardar Datos” que coincide con el objetivo de la aplicación.

*Bosquejo inicial:*

CARGAR BORRAR MODIF. CARGA MANUAL IMPRIMIR	LOGO			
	CODIGO	NOMBRE	CANTIDAD	PRECIO/U
	000053	TORNILLOS	20	2.50
	000156	MARTILLO	1	2499.00
	TOTAL:			2549.00

*Aplicación final:*

#	Codigo	Nombre	Cantidad	Precio
1	135	Tornillos	80	320
2	78	-Clavos 10mm, Cromados	50	150

## Desarrollo del código en Python de la aplicación:

A continuación, se explicará y se comentará brevemente todo el contenido del código en Python utilizado para desarrollar la aplicación:

Se comienza importando todas las librerías necesarias para utilizar el programa: las primeras 6 importaciones corresponden a las librerías utilizadas por tkinter, a excepción de la librería “os” que se utiliza para determinar la ruta en donde se encuentra la imagen del logo, que se explicara mas adelante. La librería “sqlite3” es utilizada para el manejo de la base de datos, mientras que la librería “re” se utiliza para verificar las expresiones regulares al momento de cargar y modificar los datos.

```
from tkinter import *
from tkinter import ttk
from tkinter.messagebox import *
from PIL import ImageTk, Image
import os
import tkinter as tk
import sqlite3
import re
```

Comenzando con el logo de la aplicación, se utiliza la librería “os” para determinar su ruta, y luego se utiliza “Image” e “ImageTK” de la librería “PIL” (ya que la imagen se encuentra en formato PNG) para importar la imagen. Luego se la coloca en la parte superior del panel de la aplicacion.

```
# Logo

BASE_DIR = os.path.dirname((os.path.abspath(__file__)))
ruta = os.path.join(BASE_DIR, "logo.png")

logo1 = Image.open(ruta)
logo = ImageTk.PhotoImage(logo1)
panel = tk.Label(main, image=logo)
panel.grid(row=0, column=0, columnspan=7, pady=10)
```

Lo que sigue, es crear el árbol donde se podrán ver los datos cargados por el usuario o que se encuentren en la base de datos y colocarlo en su posición de la grilla. El árbol contara con 5 columnas en total:

- # en donde se ordenan los ítems del árbol
- Código en donde se coloca el código numérico del articulo
- Nombre en donde se coloca una breve descripción del articulo con algunos detalles
- Cantidad en donde se coloca la cantidad de dicho artículo
- Precio en donde se coloca el precio total por la cantidad de dicho articulo

```
# Arbol de datos

tree = ttk.Treeview(main, show="headings", height=14)
tree["columns"] = ("#", "cod", "nombre", "cant", "precio")
tree.column("#", width=50, minwidth=50, anchor=S)
tree.column("cod", width=80, minwidth=50, anchor=S)
tree.column("nombre", width=200, minwidth=200, anchor=S)
tree.column("cant", width=80, minwidth=50, anchor=S)
tree.column("precio", width=80, minwidth=50, anchor=S)

tree.heading("#", text="#")
tree.heading("cod", text="Codigo")
tree.heading("nombre", text="Nombre")
tree.heading("cant", text="Cantidad")
tree.heading("precio", text="Precio")

tree.grid(row=1, column=2, rowspan=14, columnspan=5, padx=20, sticky=N)
```

Tras crear el árbol se crea, se coloca en la esquina inferior derecha el precio total de todos los artículos que se encuentran cargados en el árbol. Esta acción, al igual que la creación del árbol, se realizan al principio del programa debido a que existe la posibilidad de que al abrir el programa ya exista una base de datos previamente creada y cargada, por lo que será necesario mostrarla apenas se abra el programa. Para ello también se definió la función “act\_precio” cuyo objetivo es actualizar dicho precio que se muestra en la esquina inferior de la aplicación. Esta función será llamada luego de cada acción (borrado, alta o modificación de algún artículo) para mantener actualizado y mostrar siempre el precio correcto.

```
# Precio Total

res_total = Label(main, text="TOTAL: $")
res_total.grid(row=14, column=4, sticky=E)

precio = StringVar()

def act_precio():
    total = 0
    for elem in tree.get_children():
        total += int(tree.item(elem, "values")[4])
    precio.set(str(total))

total = Label(main, textvariable=precio)
total.grid(row=14, column=5, sticky=W)
```

En la sección de base de datos, se crean 4 funciones:

- “crear\_base” que simplemente crea una base de datos con el nombre “lista\_clientes\_2022” o en el caso de que esta base de datos ya exista, simplemente se conecta.
- “borrar\_datos” es una función que no está relacionada con la base de datos. Esta función borra los elementos del árbol en el programa. Se define en esta sección ya que es utilizada en la función “carga\_datos”.
- “tabla\_no\_existe” es una función que verifica si ya existe (o no) una tabla con el nombre cliente dentro de la base de datos. Esta función es necesaria ya que en el caso de que se abra una base de datos previamente cargada la tabla cliente existirá, pero si se intentara crear varias veces la tabla nuevamente (pulsando repetidamente el botón “Nuevo Cliente”) será

necesario devolver al usuario un error, indicándole que ya hay una tabla creada y que no es necesario crear una nueva.

- “carga\_datos” en el caso de que la tabla exista, es decir que se ejecute el programa con una base de datos previamente cargada, lo que hace es limpiar el árbol utilizando la función “borrar\_datos” y luego carga todos los elementos de la base de datos dentro del árbol para que puedan ser visibles por el usuario dentro de la aplicación. Si la tabla no existía, no se realiza ninguna acción y más adelante se creará.

```
# Apertura de Base de Datos

def crear_base():
    con = sqlite3.connect("lista_clientes_2022.db")
    return con

def borrar_datos():
    for elem in tree.get_children():
        tree.delete(elem)

def tabla_no_existe():
    sql = "SELECT * FROM sqlite_master WHERE type='table'"
    cursor.execute(sql)
    ex = cursor.fetchall()
    return ex == []
```

```
def carga_datos():
    if not tabla_no_existe():
        borrar_datos()
        sql = "SELECT * FROM cliente"
        cursor.execute(sql)
        datos = cursor.fetchall()
        for i in range(len(datos)):
            tree.insert("", "end", values=(datos[i][0], datos[i][1], datos[i][2], datos[i][3], datos[i][4]))
        act_precio()
```

Por último, en esta sección se ejecutan las funciones dentro del programa para crear una base de datos, definida anteriormente. Adicionalmente se define lo que será el cursor para utilizar en todas las funciones que involucren alguna acción con la base de datos y por ultimo se invoca la función “carga\_datos” para cargar la base de datos, en el caso de que esta ya existiera antes de abrir el programa.

```
con = crear_base()
cursor = con.cursor()
carga_datos()
```

En el caso de que no existiera la base de datos previamente cargada y sea necesario crear una nueva, se definen las siguientes funciones para utilizar:

- “crear\_cliente” primero verifica si no hay una tabla creada utilizando la función definida anteriormente “tabla\_no\_existe”. Si dicha función devuelve el valor FALSE (es decir que la tabla SI existe) se devuelve un mensaje de error indicándole al usuario que ya existe la tabla llamada cliente y no es necesario crear una nueva. En el caso contrario, es decir que se devuelva TRUE, se crea una tabla nueva llamada cliente dentro de la base de datos utilizando el formato de columnas definido anteriormente.

- “borrar cliente” primero confirma si el usuario realmente desea borrar la tabla cliente y no pulso el botón por accidente. En caso de ser afirmativo se verifica al igual que la función anterior si la tabla existe o no utilizando “tabla\_no\_existe” y se borra la tabla en el caso de que esta exista. Se verifica si existe la tabla a la hora de borrar, para evitar el caso de que el usuario abra por primera vez el programa, la tabla no exista, y se intente borrar una tabla inexistente. Luego de borrar el cliente, se actualiza el precio con la función “act\_precio” al valor de cero.

```
# Creacion/Borrado de Clientes

def crear_cliente():
    if tabla_no_existe():
        sql = "CREATE TABLE IF NOT EXISTS cliente(id INTEGER PRIMARY KEY, codigo INTEGER, nombre TEXT, cantidad INTEGER, precio INTEGER)"
        cursor.execute(sql)
        showinfo("Exito al crear", "Cliente creado y cargado al sistema con exito")
    else:
        showerror("Error al crear", "Error: ya hay un cliente creado y cargado en el sistema")

def borrar_cliente():
    confirmar = askyesno("Confirmar", "¿Desea borrar el cliente actual cargado en el sistema?")
    if confirmar:
        if tabla_no_existe():
            showerror("Error al borrar", "Cliente inexistente")
        else:
            sql = "DROP TABLE cliente"
            cursor.execute(sql)
            borrar_datos()
            showinfo("Operacion completada", "Cliente borrado con exito")
    act_precio()
```

Por ultimo se colocan los botones que realizaran las acciones antes mencionadas en la grilla:

```
nuevo_c = Button(main, text="Nuevo cliente", command=crear_cliente)
nuevo_c.grid(row=0, columnspan=2, sticky=S)

borrar_c = Button(main, text="Borrar cliente actual", command=borrar_cliente)
borrar_c.grid(row=1, columnspan=2)
```

## Carga de datos:

Antes de comenzar a definir las funciones y acciones necesarias para cargar un cliente, se definen lo que serán los patrones para verificar las expresiones regulares. El campo de nombre, tanto en la opción de modificar como en la opción de cargar permite solamente caracteres alfanuméricos, además del punto, la coma, el guion medio y los espacios en blanco. Los campos código, cantidad y precio solamente permiten caracteres numéricos enteros, por lo que se excluyen las comas, y los puntos. Adicionalmente se crean todos los botones y campos de entrada para cargar todos los datos. Luego se los coloca en la parte izquierda de la grilla.

```

patronNombre = "[a-zA-Z0-9 ,.-]+$"
patronNumeros = "[0-9]+$"

cargar = Label(main, text="Cargar elemento")
cargar.grid(row=3, columnspan=2, sticky=S,)

carga_codigo = Label(main, text="Codigo")
carga_nombre = Label(main, text="Nombre")
carga_cantidad = Label(main, text="Cantidad")
carga_precio = Label(main, text="Precio($)")
carga_codigo.grid(row=4, column=0, sticky=W, padx=5)
carga_nombre.grid(row=5, column=0, sticky=W, padx=5)
carga_cantidad.grid(row=6, column=0, sticky=W, padx=5)
carga_precio.grid(row=7, column=0, sticky=W, padx=5)

codigo_entry = Entry(main)
nombre_entry = Entry(main)
cantidad_entry = Entry(main)
precio_entry = Entry(main)
codigo_entry.grid(row=4, column=1)
nombre_entry.grid(row=5, column=1)
cantidad_entry.grid(row=6, column=1)
precio_entry.grid(row=7, column=1)

```

La función “verifRegEx” recibe como parámetros, los valores del código, el nombre, la cantidad y el precio del artículo al que se quiere cargar. Esta función verifica si los valores ingresados para cada campo coinciden con los patrones definidos anteriormente. En el caso de que no coincidan, devuelve el respectivo mensaje de error para cada caso. Por último, se regresa un 0 en caso de que todos coincidan, o un 1 en el caso de que haya al menos un valor que no coincidió con su patron. Esto ultimo se hace para evitar que se cargue en el árbol un artículo cuyos valores ingresados no coincidan con los patrones.

```

def verifRegEx(codigo, nombre, cant, precio):
    error = 0
    if (not re.match(patronNombre, nombre)):
        showerror("Error al ingresar los datos", "Error, verifique los datos ingresados en el campo nombre")
        error = 1
    if (not re.match(patronNumeros, codigo)):
        showerror("Error al ingresar los datos", "Error, verifique los datos ingresados en el campo codigo ([0-9])")
        error = 1
    if not re.match(patronNumeros, cant):
        showerror("Error al ingresar los datos", "Error, verifique los datos ingresados en el campo cantidad ([0-9])")
        error = 1
    if not re.match(patronNumeros, precio):
        showerror("Error al ingresar los datos", "Error, verifique los datos ingresados en el campo precio ([0-9])")
        error = 1
    return error

```

Luego se define una variable global llamada “contador” y se le asigna el valor de 1. Esta variable almacenara y se modificara para estar siempre con el valor del numero de la cantidad de elementos cargados y así poder actualizar el contador dentro del árbol (columna #) en el caso de que se borre o se cargue algún elemento. Para realizar esto ultimo se utiliza la función “ajuste\_id” que actualiza y acomoda todos los números de los ID en el árbol, de tal manera de que estos siempre sean incrementales a partir del 1, y evitar números repetidos o que falten números entre dos ID. Esta función también se invoca al finalizar cada acción de Alta, Borrado, o Modificación para mantener la numeración en el árbol de la aplicación.

```

contador = 1

def ajuste_id():
    indice = 1
    for elem in tree.get_children():
        tree.set(elem, '#1', indice)
        indice += 1

```

La función “cargar\_elem” toma los datos ingresados en los campos de entrada de cada columna junto con el valor de la variable “contador” definido anteriormente y carga el artículo dentro del árbol. Primero se obtienen y se guardan los valores de cada entrada dentro de cada variable. Luego se verifica si la función definida anteriormente “verifRegEx” regresa un valor de uno o cero. Si regresa un cero significa que todos los campos fueron completados correctamente y se procede a cargar el elemento al árbol, seguido del ajuste en el número del ID utilizando la función anterior y la actualización del precio total utilizando “act\_precio”. Las últimas 4 líneas dentro de la función solamente borran lo que ingresó el usuario en cada campo tras apretar el botón “Cargar” dentro del programa, para evitar que el usuario tenga que borrar manualmente cada una de las entradas. Por último se crea el botón que cargara los elementos al árbol llamado “Cargar” y se le asigna su posición en la grilla.

```

def cargar_elem():
    global contador
    codigo = codigo_entry.get()
    nombre = nombre_entry.get()
    cant = cantidad_entry.get()
    precio = precio_entry.get()
    if verifRegEx(codigo, nombre, cant, precio) == 0:
        tree.insert("", "end", values=(contador, codigo, nombre, cant, precio))
        contador += 1
        ajuste_id()
        act_precio()
    codigo_entry.delete(0, END)
    nombre_entry.delete(0, END)
    cantidad_entry.delete(0, END)
    precio_entry.delete(0, END)

cargar_boton = Button(main, text="Cargar", width=15, command=cargar_elem)
cargar_boton.grid(row=8, columnspan=2, pady=10)

```

## Borrado de Datos:

Para borrar los datos dentro del árbol se invoca la función “borrar\_elem” cuyo objetivo es borrar el artículo que este seleccionado dentro del árbol. Luego de borrar, se ajusta la cantidad de ítems disminuyendo el valor de la variable “contador”, se ajusta el ID de todos los elementos para que queden correctamente enumerados usando “ajuste\_id” y se actualiza el precio total de la esquina



inferior derecha con “act\_precio”. Adicionalmente se crea el botón de “Borrar” que será el encargado de invocar la función “borrar\_elem” cuando se lo pulse, y se lo coloca en la grilla.

```
def borrar_elem():
    global contador
    item = tree.focus()
    tree.delete(item)
    contador -= 1
    ajuste_id()
    act_precio()

borrar_boton = Button(main, text="Borrar", width=15, command=borrar_elem)
borrar_boton.grid(row=9, columnspan=2)
```

### Modificación de Datos:

Para modificar los datos, se define una variable llamada “modif\_opc” la cual contiene la elección del usuario respecto al dato que quiere modificar (modificar el código, el nombre, la cantidad o el precio del artículo). Luego se define la función “modificar”. Para utilizarla, el usuario primero debe seleccionar una de las 4 opciones en el panel izquierdo y además debe clicar el elemento que desee modificar dentro del árbol. Se utiliza el mismo campo de entrada para todas las opciones, pero dependiendo de la elección del usuario se guarda el valor nuevo a modificar en distintas variables. Para ello se utiliza el match case de Python. En el caso de que el usuario haya escogido la opción número uno (modificar el código) primero se verifica que el nuevo valor a modificar cumpla las condiciones del patron (en el caso del código, precio o cantidad, estos valores ingresados deben ser números que van desde el 0 al 9 sin puntos ni comas, y en el caso de ser nombre del artículo, este debe contener únicamente letras mayúsculas o minúsculas, números, guiones medios, espacios, puntos o comas). En el caso de que se cumpla, se crea un arreglo auxiliar llamado “arr”, al cual se le copian todos los valores del artículo seleccionado del árbol. Luego se le modifica únicamente la columna correspondiente a la opción seleccionada por el usuario y por último se cambian los valores de todas las columnas del artículo seleccionado, por los valores de “arr” que ahora tendrán la información actualizada y modificada. Por último, se borra el valor que ingreso el usuario en el campo de entrada, para evitar que lo tenga que hacer el usuario manualmente, y se actualiza el precio, por si llega a ocurrir el caso en el que el usuario modifiko la cantidad o el precio de un artículo. Para los demás casos (el usuario escoge la opción número 2, 3 o 4 que corresponden a nombre, cantidad o precio respectivamente) se realiza un proceso análogo. En ningún caso se actualiza el ID del artículo ya que el usuario no tiene permitido modificar esa columna desde el programa. Luego se agregan todos los RadioButton necesarios, los campos de entrada y los Labels correspondientes en la grilla.

```

modif_opc = IntVar()

def modificar():
    selec = tree.focus()
    opcion = modif_opc.get()
    match opcion:
        case 1:
            nuevo_cod = nuevo_entry.get()
            if (not re.match(patronNumeros, nuevo_cod)):
                showerror("Error al modificar los datos", "Error, verifique el formato del codigo ([0-9])")
            else:
                valor = tree.item(selec)
                arr = valor["values"]
                arr[1] = nuevo_cod
                tree.item(selec, values=(arr[0],arr[1],arr[2], arr[3], arr[4]))
        case 2:
            nuevo_nombre = nuevo_entry.get()
            if (not re.match(patronNombre, nuevo_nombre)):
                showerror("Error al modificar los datos", "Error, verifique el formato del nombre")
            else:
                valor = tree.item(selec)
                arr = valor["values"]
                arr[2] = nuevo_nombre
                tree.item(selec, values=(arr[0],arr[1],arr[2], arr[3], arr[4]))
        case 3:
            nueva_cant = nuevo_entry.get()
            if (not re.match(patronNumeros, nueva_cant)):
                showerror("Error al modificar los datos", "Error, verifique el formato de la cantidad ([0-9])")
            else:
                valor = tree.item(selec)
                arr = valor["values"]
                arr[3] = nueva_cant
                tree.item(selec, values=(arr[0],arr[1],arr[2], arr[3], arr[4]))

```

```

        case 4:
            nuevo_precio = nuevo_entry.get()
            if (not re.match(patronNumeros, nuevo_precio)):
                showerror("Error al modificar los datos", "Error, verifique el formato del precio ([0-9])")
            else:
                valor = tree.item(selec)
                arr = valor["values"]
                arr[4] = nuevo_precio
                tree.item(selec, values=(arr[0],arr[1],arr[2], arr[3], arr[4]))
    nuevo_entry.delete(0, END)
    act_precio()

```

```

modif = Label(main, text="Modificar valor")
modif.grid(row=10, columnspan=2, sticky=S, pady=5)

opc_codigo = Radiobutton(main, text="Codigo", value=1, variable=modif_opc)
opc_codigo.grid(row=11, column=0, sticky=W, padx=10)
opc_nombre = Radiobutton(main, text="Nombre", value=2, variable=modif_opc)
opc_nombre.grid(row=11, column=1, sticky=W, padx=20)
opc_cantidad = Radiobutton(main, text="Cantidad", value=3, variable=modif_opc)
opc_cantidad.grid(row=12, column=0, sticky=S, padx=10)
opc_precio = Radiobutton(main, text="Precio", value=4, variable=modif_opc)
opc_precio.grid(row=12, column=1, sticky=W, padx=20)

nuevo_val = Label(main, text="Nuevo Valor")
nuevo_val.grid(row=13, column=0, pady=8)
nuevo_entry = Entry(main)
nuevo_entry.grid(row=13, column=1, pady=8)

modif_boton = Button(main, text="Modificar", width=15, command=modificar)
modif_boton.grid(row=14, columnspan=2, pady=10)

```

## Consultar datos:

Como el usuario dispone en todo momento de la información en el árbol, no es necesario realizar consultas sobre alguna información de los artículos. Por lo tanto, cuando el usuario presione el botón “Consultar Información” verá un mensaje que muestre la siguiente información sobre la base de datos. Es importante remarcar que la información que se ve en el árbol no es necesariamente igual a la información de la base de datos, ya que la información de la base de datos solamente se actualizará cada vez que el usuario presione el botón “Guardar Datos”. Si el usuario agrega, borra o modifica artículos del árbol y no guarda su información, la base de datos no cambiará.

Información que verá el cliente al consultar:

- Cantidad de artículos distintos (numero de filas en la base de datos)
- Cantidad total de artículos cargados (sumatoria de todos los elementos de la columna cantidad dentro de la base de datos)
- Precio total de los elementos (sumatoria de todos los elementos de la columna precio dentro de la base de datos)

Para esta última se creó una función auxiliar llamada “consulta\_precio\_total” que realiza dicha sumatoria de todos los elementos de la columna precios en la base de datos. Para mostrar toda la información listada anteriormente en conjunto se utiliza la función “consulta” que realiza y muestra la suma de artículos distintos, artículos cargados y del precio total. Por último se coloca el botón llamado “Consultar Información” en la parte inferior de la aplicación.

```
# Consultar

def consulta_precio_total():
    sql = "SELECT precio FROM cliente"
    cursor.execute(sql)
    res = cursor.fetchall()
    precio_total = 0
    for i in res:
        precio_total += i[0]
    return precio_total

def consulta():
    sql = "SELECT COUNT(*) FROM cliente"
    cursor.execute(sql)
    res = cursor.fetchone()
    cant_art_dist = res[0]
    sql = "SELECT cantidad FROM cliente"
    cursor.execute(sql)
    res = cursor.fetchall()
    cant_art_total = 0
    for i in res:
        cant_art_total += i[0]
    precio_total = consulta_precio_total()
    showinfo("Resultado de la Consulta", "Estado actual de la base de datos del cliente:\n\nCantidad de articulos distintos cargados: " + str(cant_art_dist) +
            "\nCantidad total de articulos cargados: " + str(cant_art_total) + "\nPrecio total de los elementos: $" + str(precio_total))

consul_boton = Button(main, text="Consultar Informacion", command=consulta)
consul_boton.grid(row=14, column=2)
```

## Guardado de información:

Para guardar todos los cambios realizados en el árbol, dentro de la base de datos se utiliza el botón “Guardar Datos” colocado junto al botón “Guardar Información”. Para guardar los datos se define una lista vacía llamada “lista\_elem” que funcionará como lista auxiliar y estará dentro de la función “guardar\_datos”. Utilizando esta función, primero se limpia completamente la base de datos. Esto se realiza con el objetivo de evitar que se sobreponga la información y aparezcan datos repetidos o que

se modificaron anteriormente. No hay riesgo de perder la información ya que todos los datos están cargados y son visibles en el árbol, antes de realizar el borrado. Una vez que se limpio la base de datos, se utiliza un ciclo for para cargar todos los elementos del árbol dentro de la lista auxiliar "lista\_elem". Una vez finalizado este proceso, se cargan todos los elementos dentro de "lista\_elem" en la base de datos utilizando otro ciclo for. Al finalizar estos procesos se tendrá toda la información copiada en el árbol, la lista auxiliar y la base de datos. Por ultimo se muestra un mensaje confirmando que se guardaron todos los datos. En las últimas líneas de código del programa, se coloca el botón "Guardar Datos" en su posición en la grilla, por ultimo se cierra la base de datos y se define el loop de tkinter.

```
# Guardar datos del cliente

def guardar_datos():
    lista_elem = []
    sql = "DELETE FROM cliente"
    cursor.execute(sql)
    for elem in tree.get_children():
        lista_elem.append(tuple(tree.item(elem)["values"]))
    for elem in lista_elem:
        sql = "INSERT INTO cliente(id, codigo, nombre, cantidad, precio) VALUES (?, ?, ?, ?, ?)"
        cursor.execute(sql, elem)
    con.commit()
    showinfo("Archivo Guardado", "Datos guardados con exito")

guardar = Button(main, text="Guardar Datos", command=guardar_datos)
guardar.grid(row=14, column=3)

main.mainloop()
con.close()
```

## Ejemplo de prueba de la aplicación:

Se inicia la aplicación a partir de los siguientes 3 archivos:

Nombre	Fecha de modificación	Tipo	Tamaño
FTools1_0	22/12/2022 22:10	Archivo de origen ...	11 KB
lista_clientes_2022	22/12/2022 22:09	SQLite	8 KB
logo	9/12/2022 14:35	Archivo PNG	12 KB

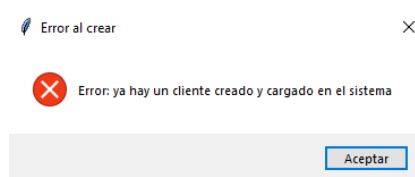
Como lista\_clientes\_2022 ya existía y se cargo un valor previamente a modo de prueba, al abrir el programa por primera vez se verá lo siguiente:

#	Codigo	Nombre	Cantidad	Precio
1	156	Pilas AA	2	150

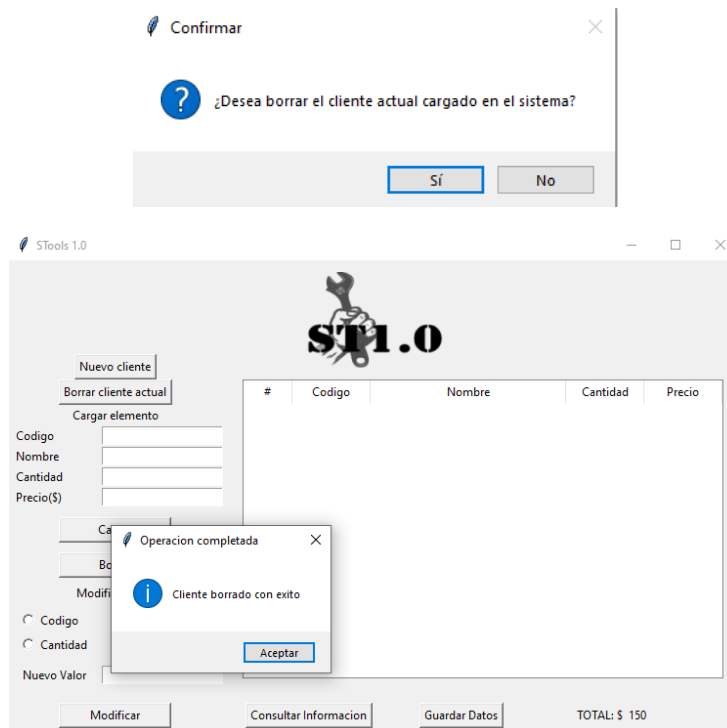
La base de datos dentro de SQLiteStudio se vera de la siguiente manera:

id	codigo	nombre	cantidad	precio
1	1	156 Pilas AA	2	150

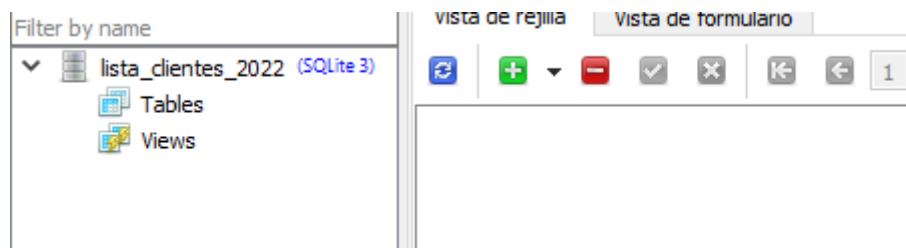
Si intentamos crear un nuevo cliente, esto no será posible ya que actualmente tenemos y estamos viendo un cliente. Por lo tanto, veremos un mensaje de error al pulsar el botón “Nuevo cliente”:



Si deseamos borrar el cliente se vería lo siguiente:



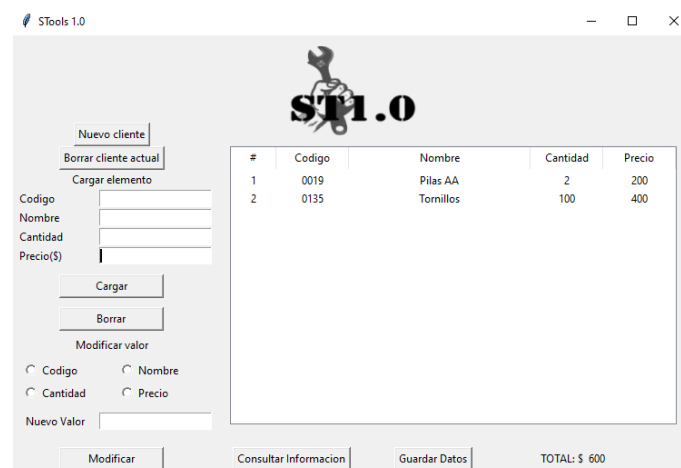
La base de datos ahora se vería sin tablas, ya que acabamos de borrar el cliente:



Si creamos un cliente nuevo, y le cargamos dos artículos:

- Código: 0019, Nombre: Pilas AA, Cantidad: 2, Precio: 200
- Código: 0135, Nombre: Tornillos, Cantidad: 100, Precio: 400

Tras pulsar "Guardar Datos" veriamos lo siguiente:



Filter by name

- lista\_cientes\_2022 (SQLite 3)
  - Tables (1)
    - cliente
  - Views

Vista de rejilla    Vista de formulario

	id	codigo	nombre	cantidad	precio
1	1	19	Pilas AA	2	200
2	2	135	Tornillos	100	400

Supongamos ahora que vamos a borrar el primer artículo, y modificaremos la cantidad del segundo artículos para que quede finalmente en 80, y por lo tanto también modificaremos su precio final en 320. Tras guardar los datos:

STools 1.0

**ST1.0**

Cargar elemento

Codigo   
 Nombre   
 Cantidad   
 Precio(\$)

Modificar valor

☐ Codigo    ☐ Nombre  
☐ Cantidad    ☒ Precio

Nuevo Valor

TOTAL: \$ 320

#	Codigo	Nombre	Cantidad	Precio
1	135	Tornillos	80	320

Filter by name

- lista\_cientes\_2022 (SQLite 3)
  - Tables (1)
    - cliente
  - Views

Vista de rejilla    Vista de formulario

	id	codigo	nombre	cantidad	precio
1	1	135	Tornillos	80	320


Por último, se agregarán tres artículos más:


- Código: 0078, Nombre: -Clavos 10mm. Cromados, Cantidad: 50, Precio: 150
- Código: 25-31, Nombre: Arandelas, Cantidad: 100, Precio: 150
- Código: 0199, Nombre: <Rodillo para paredes>, Cantidad: 1, Precio: 900

Tras cargar uno por uno, veremos lo siguiente dentro del programa:

#	Codigo	Nombre	Cantidad	Precio
1	135	Tornillos	80	320
2	0078	-Clavos 10mm. Cromados	50	150


Al cargar el segundo artículo:


 Error al ingresar los datos

 Error, verifique los datos ingresados en el campo codigo ([0-9])

Aceptar


Al cargar el tercer artículo:

 Error al ingresar los datos

 Error, verifique los datos ingresados en el campo nombre

Aceptar

Por último, se guardarán los datos con solamente los dos artículos que se cargaron correctamente:



Nuevo cliente  
 Borrar cliente actual  
 Cargar elemento  
 Codigo  
 Nombre  
 Cantidad  
 Precio(\$)  
 Cargar  
 Borrar  
 Modificar valor  
☐ Codigo ☐ Nombre  
☐ Cantidad ☒ Precio  
 Nuevo Valor

#	Codigo	Nombre	Cantidad	Precio
1	135	Tornillos	80	320
2	0078	-Clavos 10mm. Cromados	50	150

Modificar Consultar Informacion Guardar Datos TOTAL: \$ 470

Filter by name
 

- lista\_clientes\_2022 (SQLite 3)
  - Tables (1)
    - cliente
  - Views

vista de rejilla vista de formulario
 

	id	codigo	nombre	cantidad	precio
1	1	135	Tornillos	80	320
2	2	78	-Clavos 10mm. Cromados	50	150



Y se realizara una consulta de la información de la base de datos:

