DB-Visualiser App

Documentation

Abhishek Sebastian

Major Modules

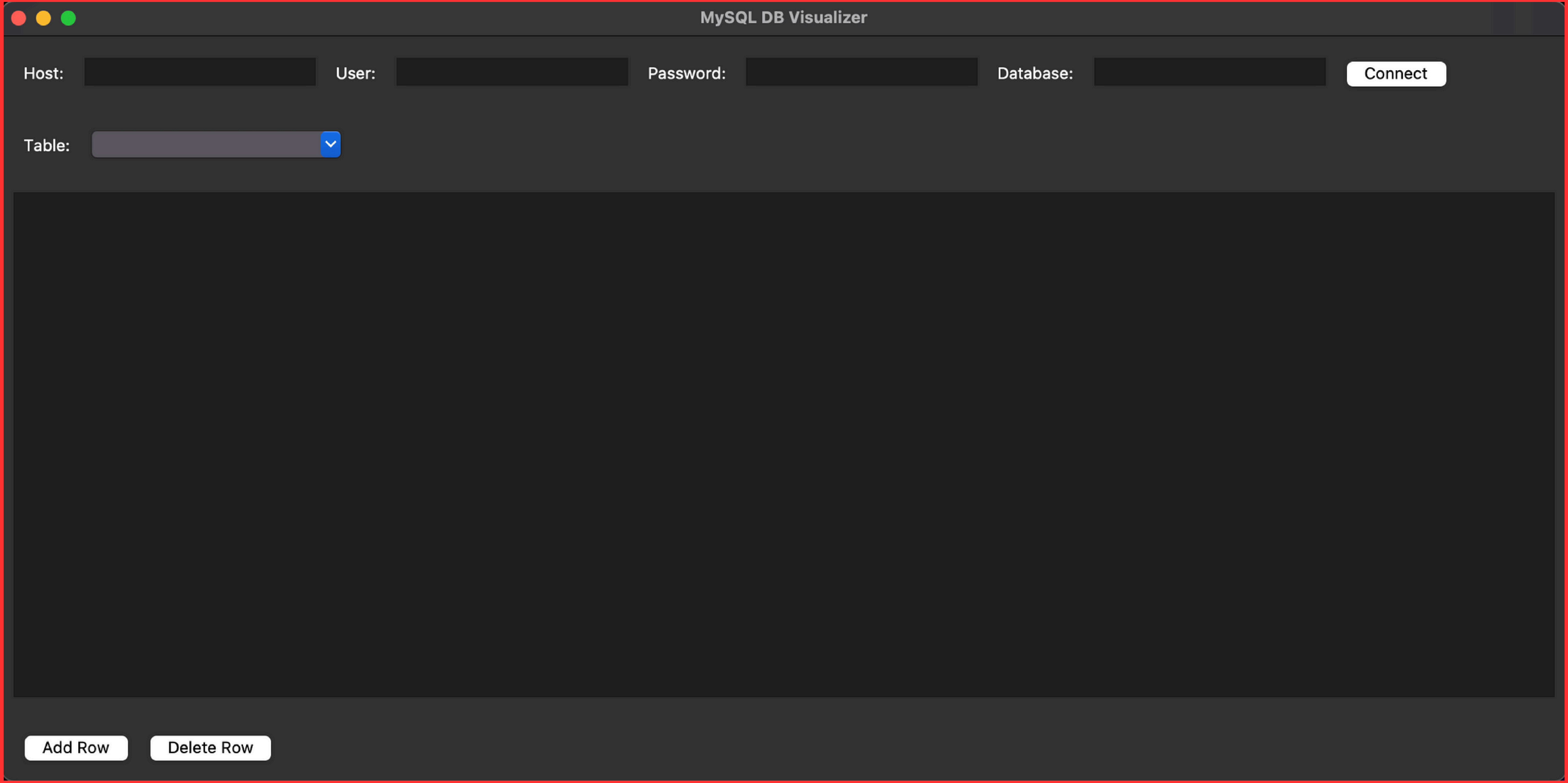
Using Tkinter

- Main APP Window
  - Login Details
  - Table Select Dropdown
  - DB Visualiser
  - DB Action Buttons

Using MYSQL Connector

- Functional Modules
  - Connect to DB
  - Select table
  - Retrieve table
  - ADD/DEL/EDIT Rows

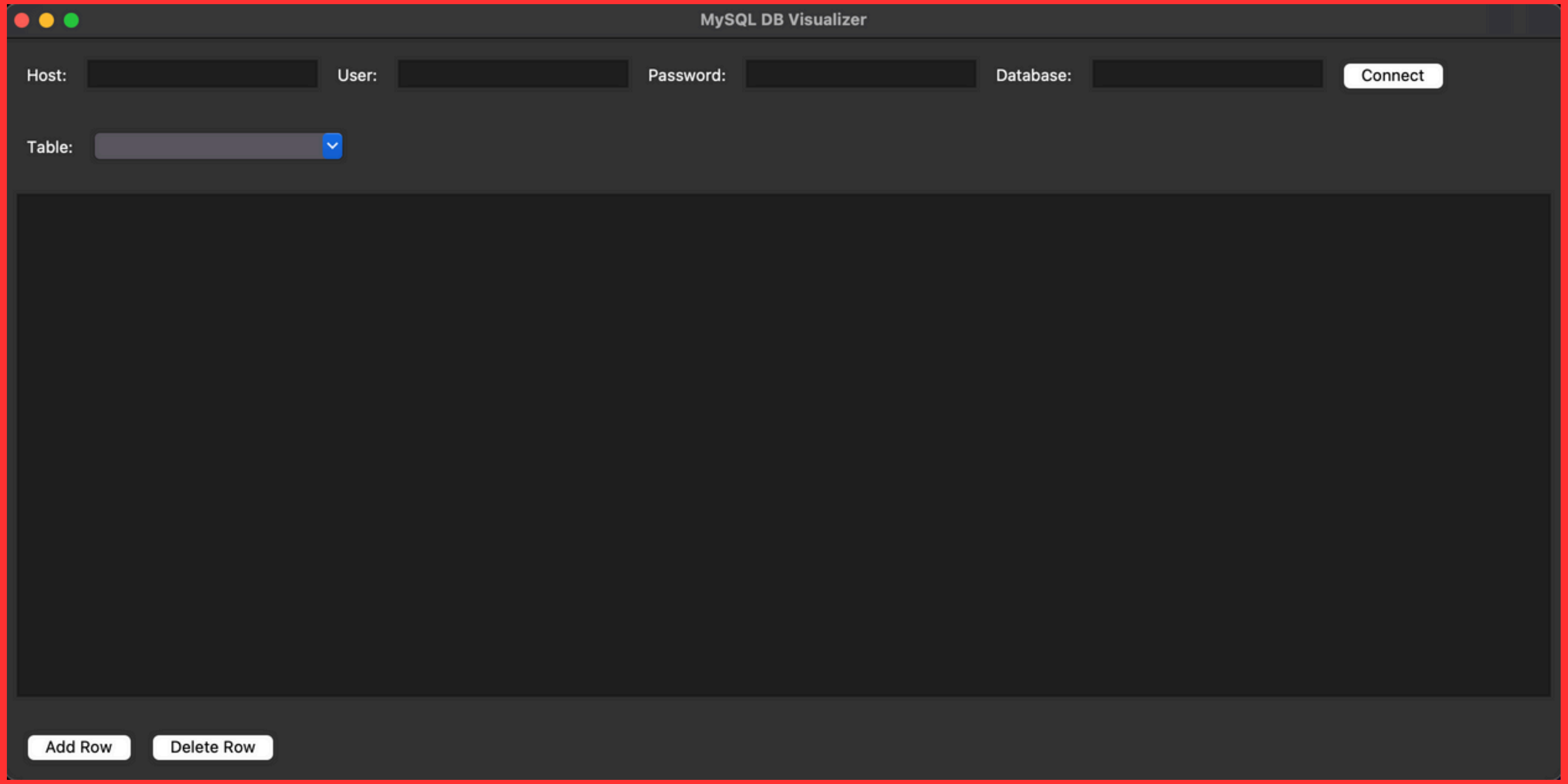
Main APP Window



Main APP Window

```
def __init__(self, root):
        self.root = root
        self.root.title("MySQL DB Visualizer")
        self.root.geometry("800x600")
        self.conn = None
        self.cursor = None
        self.create_widgets()

 def create_widgets(self):
 # Database Connection Frame
 connection_frame = tk.Frame(self.root, padx=10, pady=10)
 connection_frame.pack(fill=tk.X)

 tk.Label(connection_frame, text="Host:").grid(
 row=0, column=0, padx=5, pady=5)
 self.host_entry = tk.Entry(connection_frame)
 self.host_entry.grid(row=0, column=1, padx=5, pady=5)

 tk.Label(connection_frame, text="User:").grid(
 row=0, column=2, padx=5, pady=5)
 self.user_entry = tk.Entry(connection_frame)
 self.user_entry.grid(row=0, column=3, padx=5, pady=5)

 tk.Label(connection_frame, text="Password:").grid(
 row=0, column=4, padx=5, pady=5)
 self.password_entry = tk.Entry(connection_frame, show="*")
 self.password_entry.grid(row=0, column=5, padx=5, pady=5)

 tk.Label(connection_frame, text="Database:").grid(
 row=0, column=6, padx=5, pady=5)
 self.database_entry = tk.Entry(connection_frame)
 self.database_entry.grid(row=0, column=7, padx=5, pady=5)

 self.connect_button = tk.Button(
 connection_frame, text="Connect", command=self.connect_db)
 self.connect_button.grid(row=0, column=8, padx=5, pady=5)

 # Table Selection Frame
 table_frame = tk.Frame(self.root, padx=10, pady=10)
 table_frame.pack(fill=tk.X)

 tk.Label(table_frame, text="Table:").grid(
 row=0, column=0, padx=5, pady=5)
 self.table_dropdown = ttk.Combobox(table_frame, state="readonly")
 self.table_dropdown.grid(row=0, column=1, padx=5, pady=5)
 self.table_dropdown.bind("<<ComboboxSelected>>", self.load_table)

 # Data Table Frame
 self.tree = ttk.Treeview(self.root, show="headings")
 self.tree.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

 self.tree.bind("<Double-1>", self.edit_row)

 # Button Frame
 button_frame = tk.Frame(self.root, padx=10, pady=10)
 button_frame.pack(fill=tk.X)

 self.add_button = tk.Button(
 button_frame, text="Add Row", command=self.add_row)
 self.add_button.pack(side=tk.LEFT, padx=5, pady=5)

 self.delete_button = tk.Button(
 button_frame, text="Delete Row", command=self.delete_row)
 self.delete_button.pack(side=tk.LEFT, padx=5, pady=5)
```

Implementation



Algorithm

The create_widgets method sets up the graphical interface for a MySQL database visualizer app. It includes:

Database Connection: A section where the user inputs database credentials (host, user, password, and database name). There is a Connect button to establish the connection.

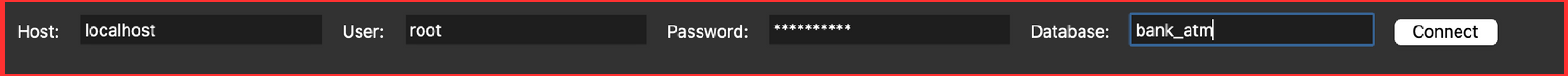
Table Selection: A dropdown list where the user can choose a table from the connected database. Once selected, it displays the table's contents.

Data Table: A table view (using Treeview) to show the data of the selected table, where each row can be double-clicked to edit.

Action Buttons: Two buttons at the bottom to add or delete rows in the selected table.

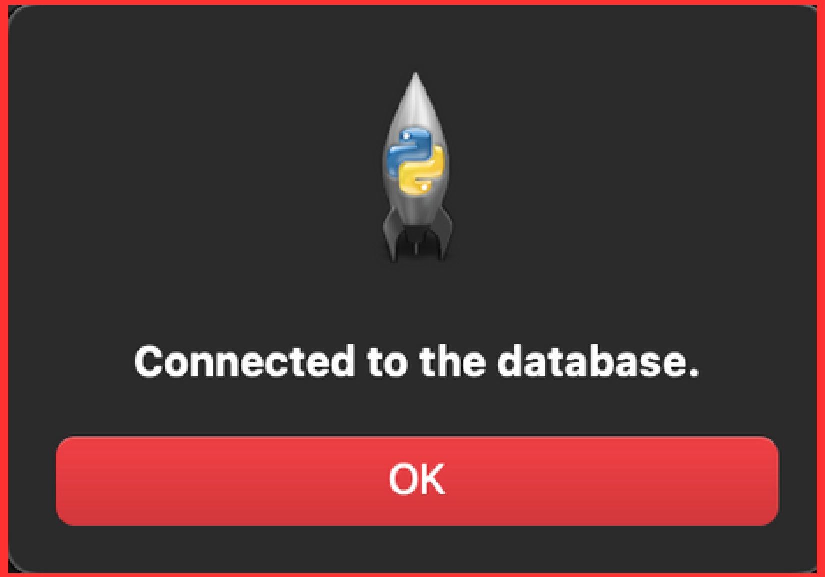
Login Connection to DB Module

User Input:



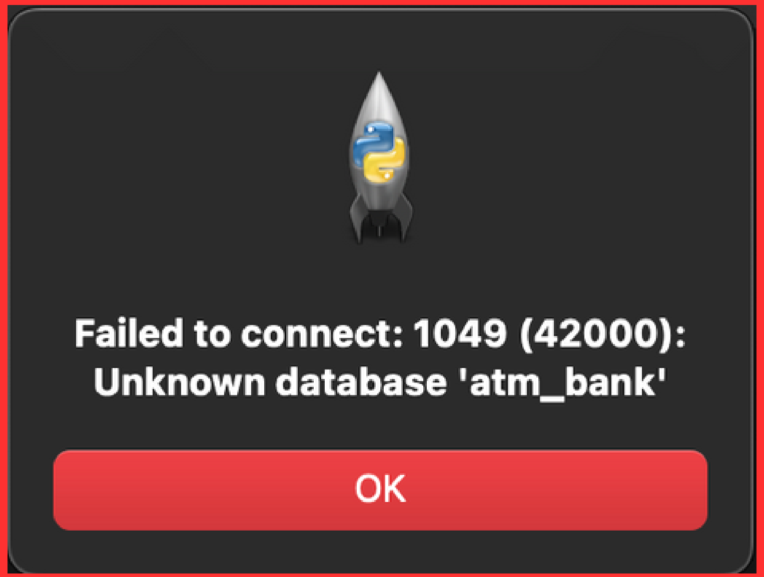
Host: localhost
User: root
Password: ***********
Database: bank_atm

Successful Case:



Connected to the database.

Error Case:



Failed to connect: 1049 (42000): Unknown database 'atm_bank'

Implementation

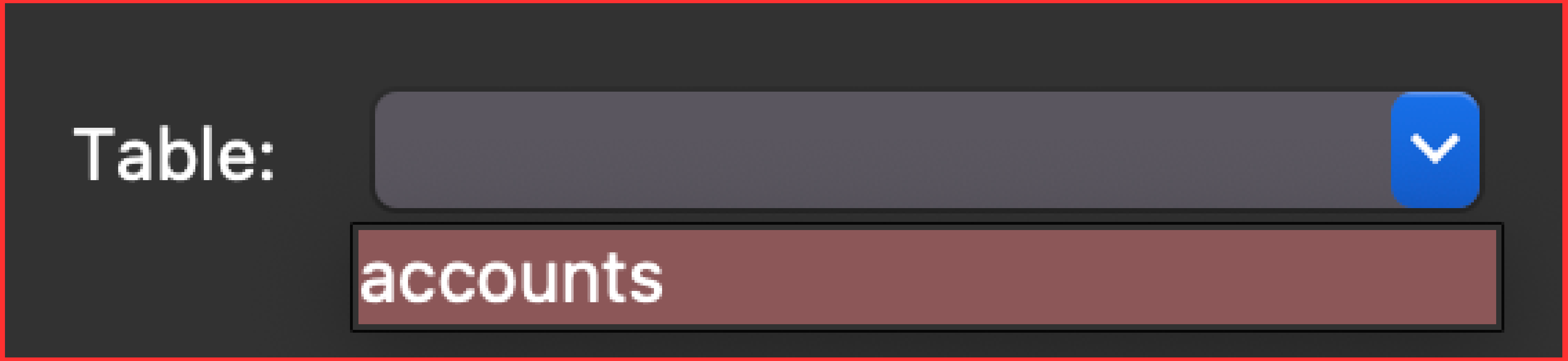
```
def connect_db(self):
 try:
     self.conn = mysql.connector.connect(
     host=self.host_entry.get(),
     user=self.user_entry.get(),
     password=self.password_entry.get(),
     database=self.database_entry.get()
     )
    self.cursor = self.conn.cursor()
    self.load_tables()
    messagebox.showinfo("Success","Connected to the database.")
 except mysql.connector.Error as err:
    messagebox.showerror("Error", f"Failed to connect: {err}")
```

Algorithm

The connect_db method connects to a MySQL database using the credentials provided by the user.

1. It gets the connection details (host, user, password, and database) from the input fields.
2. It tries to establish the connection and create a cursor to interact with the database.
3. If successful, it loads the available tables and shows a success message.
4. If there's an error, it shows an error message with the specific issue.

Table Dropdown Module



Implementation

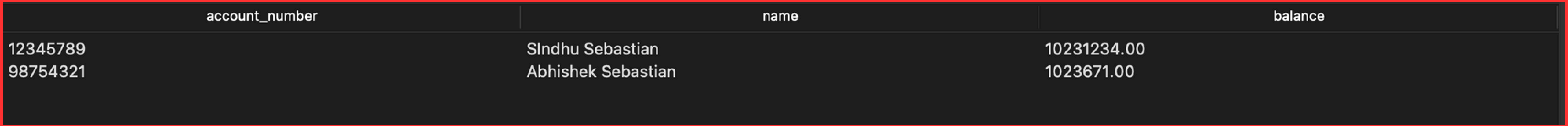
```
def load_tables(self):
    try:
       self.cursor.execute("SHOW TABLES")
       tables = [row[0] for row in self.cursor.fetchall()]
       if not tables:
           messagebox.showwarning(
                   "No Tables", "No tables found in the database.")
        self.table_dropdown["values"] = tables
     except mysql.connector.Error as err:
        messagebox.showerror("Error", f"Failed to load tables:{err}")
```

Algorithm

The load_tables method retrieves and displays the list of tables from the connected MySQL database.

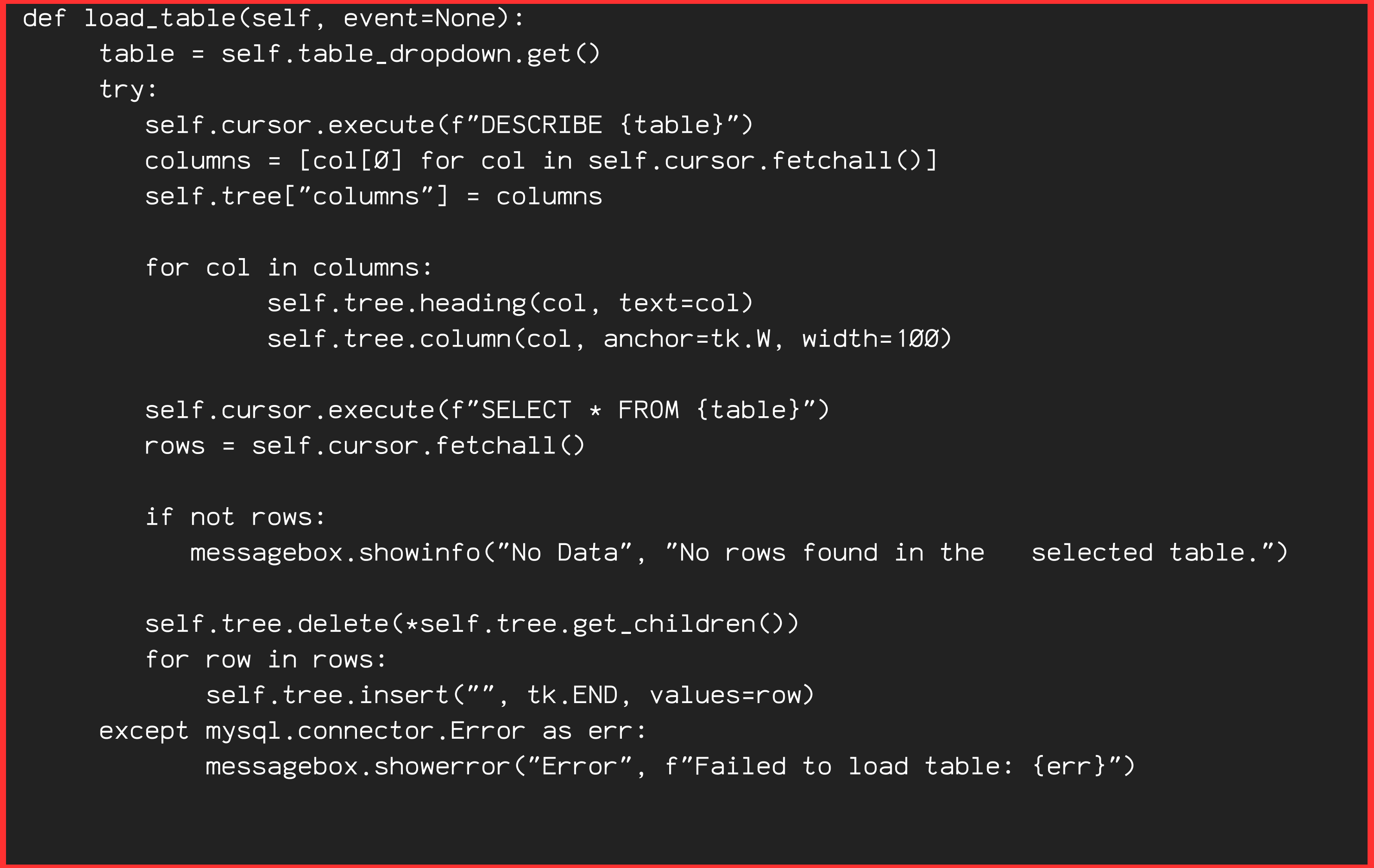
1. Fetch Tables: It executes the SQL query SHOW TABLES to get the list of tables in the database using the cursor.
2. Store Tables: It retrieves the table names from the query result and stores them in the tables list.
3. Check for Tables: If no tables are found, a warning message is shown to the user.
4. Update Dropdown: If tables are found, the list of tables is set as the options in the table_dropdown (Combobox) for the user to select.
5. Error Handling: If there's any issue fetching the tables, an error message is shown with the specific error.

Table Visualise Module



account_number	name	balance
12345789	SIndhu Sebastian	10231234.00
98754321	Abhishek Sebastian	1023671.00

Implementation



```
def load_table(self, event=None):
     table = self.table_dropdown.get()
     try:
        self.cursor.execute(f"DESCRIBE {table}")
        columns = [col[0] for col in self.cursor.fetchall()]
        self.tree["columns"] = columns

        for col in columns:
                self.tree.heading(col, text=col)
                self.tree.column(col, anchor=tk.W, width=100)

        self.cursor.execute(f"SELECT * FROM {table}")
        rows = self.cursor.fetchall()

        if not rows:
           messagebox.showinfo("No Data", "No rows found in the   selected table.")

        self.tree.delete(*self.tree.get_children())
        for row in rows:
            self.tree.insert("", tk.END, values=row)
     except mysql.connector.Error as err:
            messagebox.showerror("Error", f"Failed to load table: {err}")
```

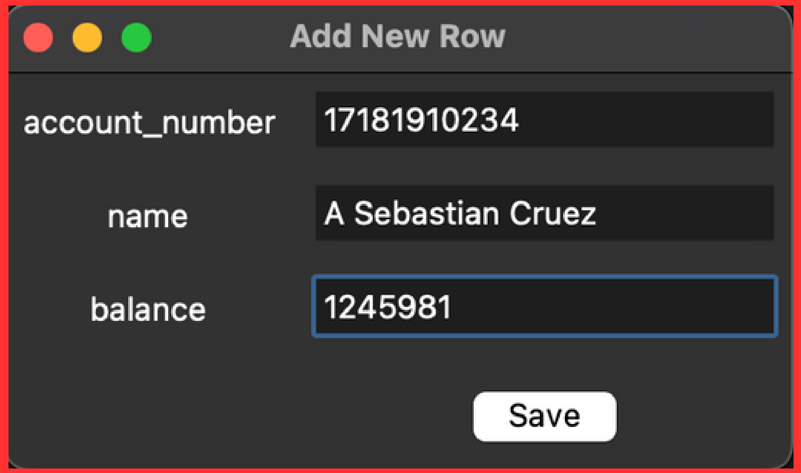
Algorithm

The load_table method loads and displays the contents of a selected table in the Treeview widget.

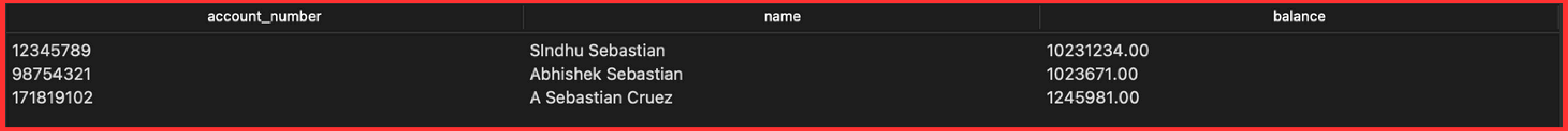
1. Get Selected Table: It retrieves the table name from the dropdown.
2. Get Column Names: It runs the DESCRIBE SQL query to get the column names of the selected table. These columns are set as the headings in the Treeview.
3. Fetch Data: It runs a SELECT * FROM table query to fetch all rows from the selected table.
4. Check for Data: If no rows are found, it shows a message indicating that the table is empty.
5. Display Data: It clears the existing data in the Treeview and inserts the fetched rows.
6. Error Handling: If any errors occur (e.g., invalid table name), an error message is shown.

Table Edit Module

Adding New Row

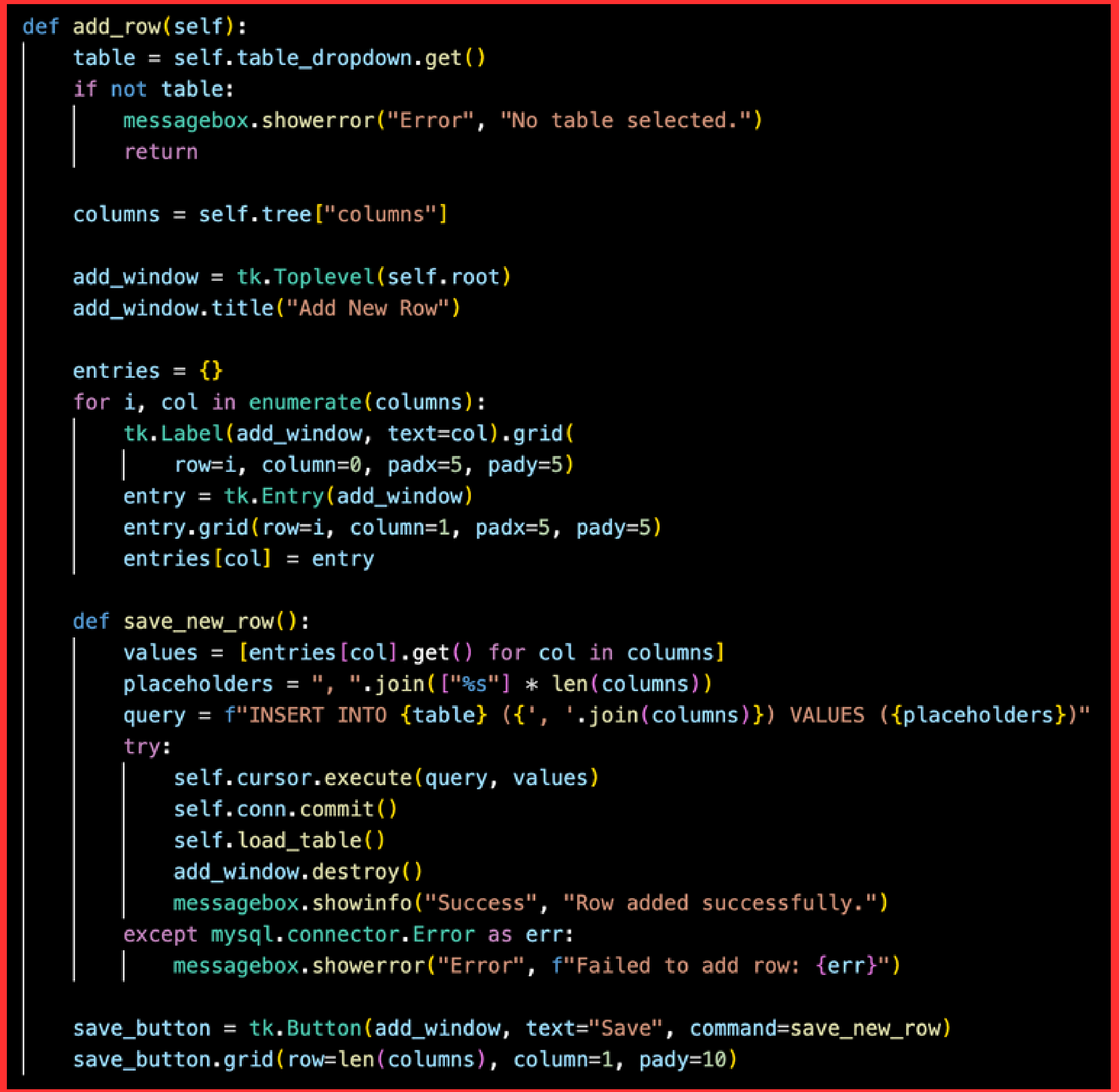


account_number	17181910234
name	A Sebastian Cruez
balance	1245981



account_number	name	balance
12345789	SIndhu Sebastian	10231234.00
98754321	Abhishek Sebastian	1023671.00
171819102	A Sebastian Cruez	1245981.00

Implementation



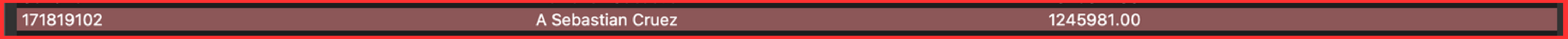
```python
def add_row(self):
    table = self.table_dropdown.get()
    if not table:
        messagebox.showerror("Error", "No table selected.")
        return

    columns = self.tree["columns"]

    add_window = tk.Toplevel(self.root)
    add_window.title("Add New Row")

    entries = {}
    for i, col in enumerate(columns):
        tk.Label(add_window, text=col).grid(
            row=i, column=0, padx=5, pady=5)
        entry = tk.Entry(add_window)
        entry.grid(row=i, column=1, padx=5, pady=5)
        entries[col] = entry

    def save_new_row():
        values = [entries[col].get() for col in columns]
        placeholders = ", ".join(["%s"] * len(columns))
        query = f"INSERT INTO {table} ({', '.join(columns)}) VALUES ({placeholders})"
        try:
            self.cursor.execute(query, values)
            self.conn.commit()
            self.load_table()
            add_window.destroy()
            messagebox.showinfo("Success", "Row added successfully.")
        except mysql.connector.Error as err:
            messagebox.showerror("Error", f"Failed to add row: {err}")

    save_button = tk.Button(add_window, text="Save", command=save_new_row)
    save_button.grid(row=len(columns), column=1, pady=10)
```

Algorithm

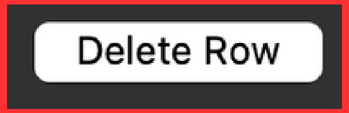
The add_row method allows the user to add a new row to the selected table.

1. Check Table Selection: It checks if a table is selected from the dropdown. If not, it shows an error message.
2. Get Columns: It retrieves the column names from the Treeview widget.
3. Create Add Row Window: It opens a new window (using Toplevel) where the user can input values for each column in the selected table.
4. Input Fields for Columns: For each column, it creates a label and an entry field where the user can input the data for that column.
5. Save New Row: The save_new_row function collects the values entered by the user, constructs an SQL INSERT query, and attempts to insert the new row into the selected table.
6. Commit Changes: If the insertion is successful, the changes are committed to the database, the table is reloaded to reflect the new row, and a success message is shown. If there’s an error, an error message is displayed.
7. Close Window: Once the row is added or an error occurs, the input window is closed.

Table Edit Module Deleting Existing Row



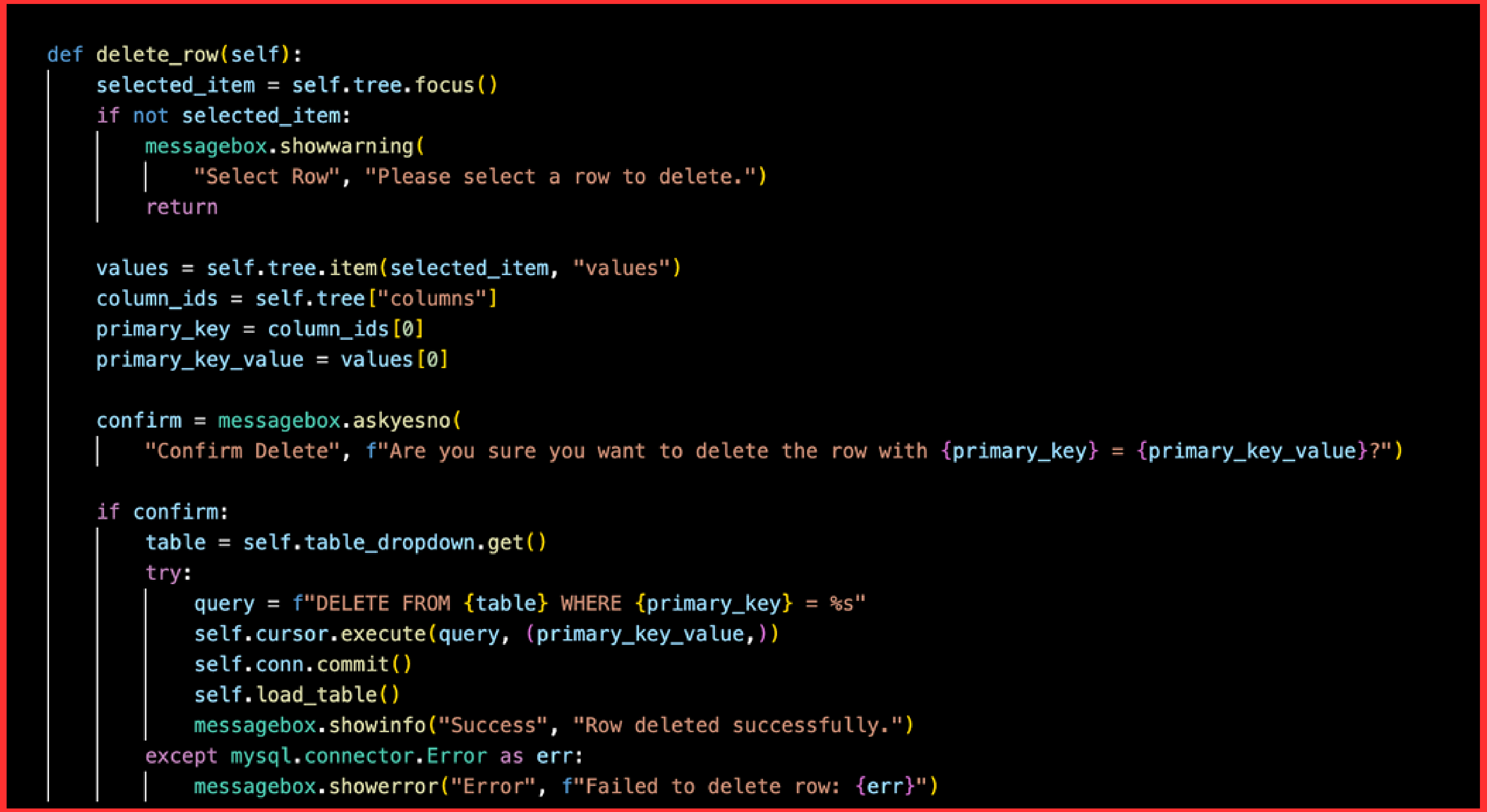
```
171819102                    A Sebastian Cruez                    1245981.00
```



Click the row to be deleted

Press Delete

Implementation



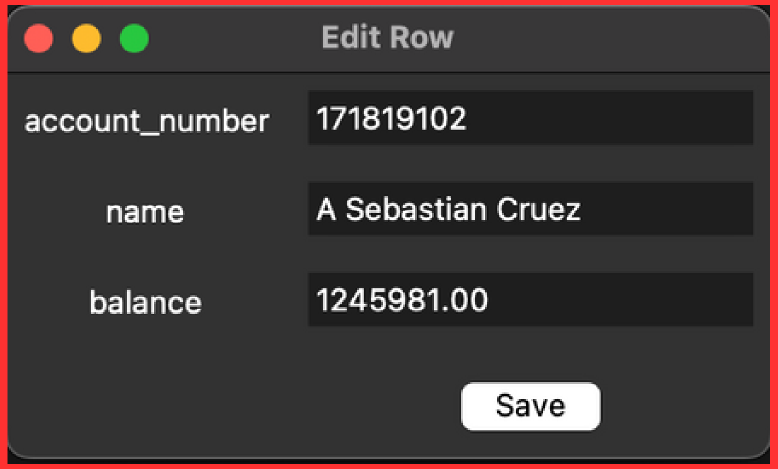
```python
def delete_row(self):
    selected_item = self.tree.focus()
    if not selected_item:
        messagebox.showwarning(
            "Select Row", "Please select a row to delete.")
        return

    values = self.tree.item(selected_item, "values")
    column_ids = self.tree["columns"]
    primary_key = column_ids[0]
    primary_key_value = values[0]

    confirm = messagebox.askyesno(
        "Confirm Delete", f"Are you sure you want to delete the row with {primary_key} = {primary_key_value}?")

    if confirm:
        table = self.table_dropdown.get()
        try:
            query = f"DELETE FROM {table} WHERE {primary_key} = %s"
            self.cursor.execute(query, (primary_key_value,))
            self.conn.commit()
            self.load_table()
            messagebox.showinfo("Success", "Row deleted successfully.")
        except mysql.connector.Error as err:
            messagebox.showerror("Error", f"Failed to delete row: {err}")
```

Algorithm

The delete_row method allows the user to delete a selected row from the database table.

1. Check Row Selection: It checks if a row is selected in the Treeview. If no row is selected, it shows a warning message.
2. Get Row Data: It retrieves the values of the selected row and identifies the primary key (assumed to be the first column) and its corresponding value.
3. Confirmation: It asks the user for confirmation before proceeding with the deletion. The user is prompted with a message showing the primary key and its value to confirm the deletion.
4. Delete Row: If the user confirms, it constructs a SQL DELETE query using the primary key and its value to delete the selected row from the table.
5. Execute Query: It executes the query, commits the change to the database, and reloads the table to reflect the updated data.
6. Error Handling: If there's any issue during the deletion process, an error message is displayed.
7. Success Message: Once the row is deleted, a success message is shown to the user.

Table Edit Module   Editing Exisiting Row

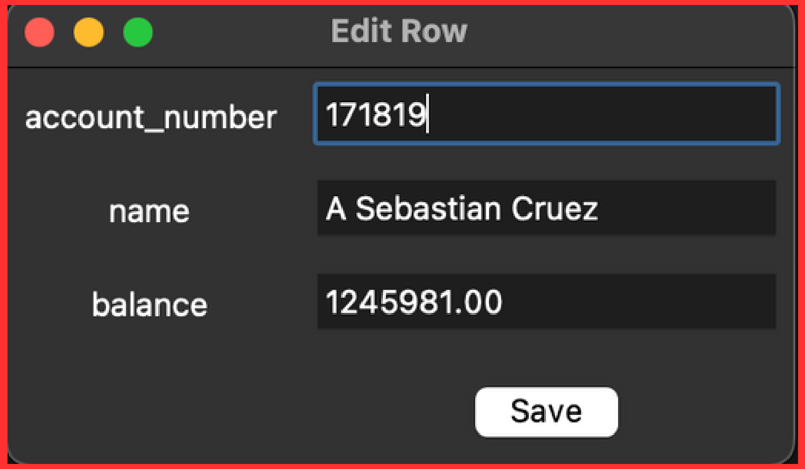

Edit Row

account_number 171819102

name A Sebastian Cruez

balance 1245981.00

Save

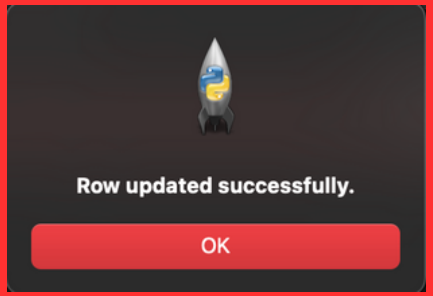

Edit Row

account_number 171819

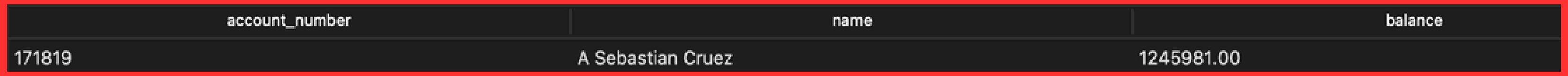
name A Sebastian Cruez

balance 1245981.00

Save

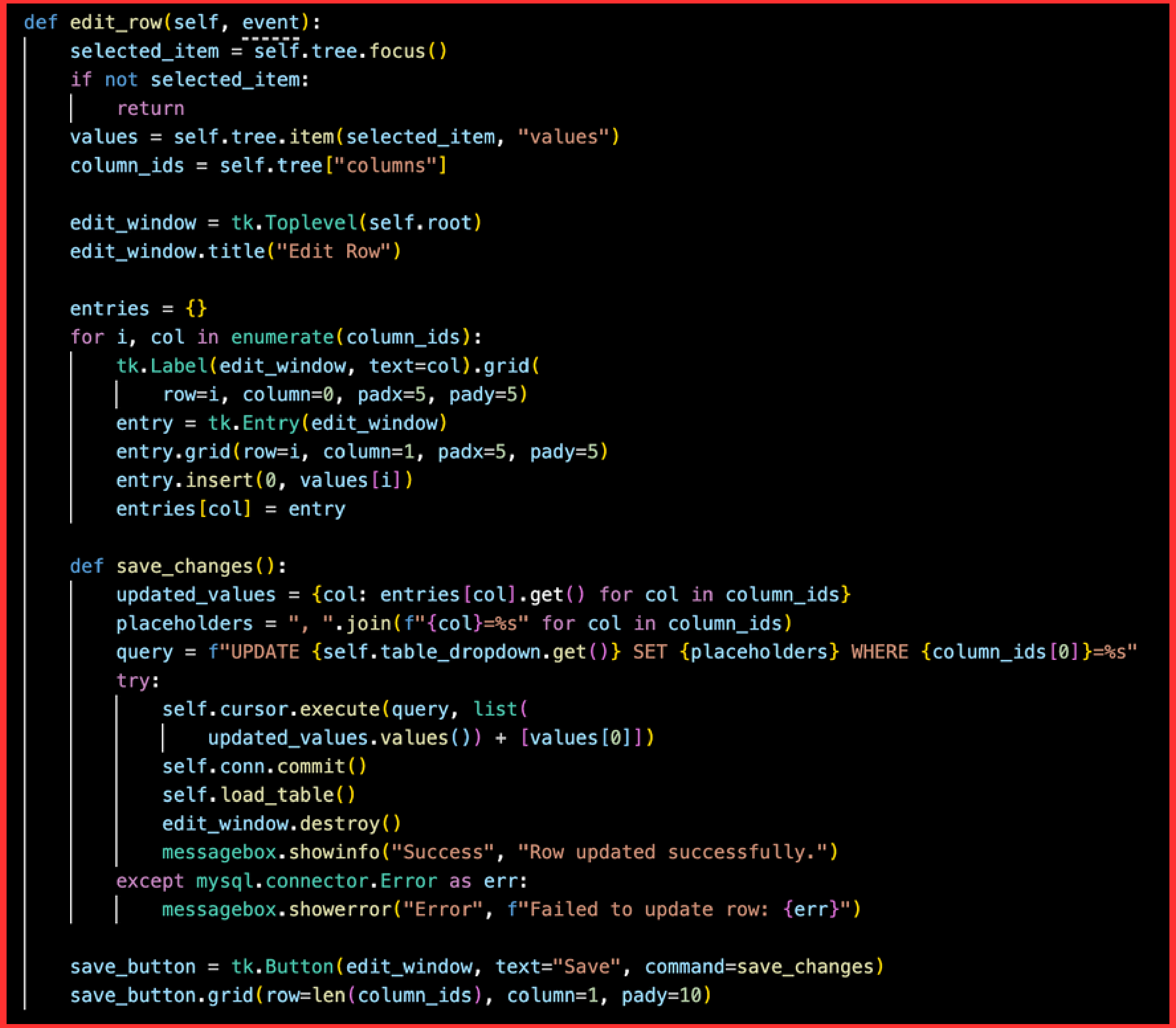

Row updated successfully.

OK



account_number	name	balance
171819	A Sebastian Cruez	1245981.00

Implementation



```
def edit_row(self, event):
    selected_item = self.tree.focus()
    if not selected_item:
        return
    values = self.tree.item(selected_item, "values")
    column_ids = self.tree["columns"]

    edit_window = tk.Toplevel(self.root)
    edit_window.title("Edit Row")

    entries = {}
    for i, col in enumerate(column_ids):
        tk.Label(edit_window, text=col).grid(
            row=i, column=0, padx=5, pady=5)
        entry = tk.Entry(edit_window)
        entry.grid(row=i, column=1, padx=5, pady=5)
        entry.insert(0, values[i])
        entries[col] = entry

    def save_changes():
        updated_values = {col: entries[col].get() for col in column_ids}
        placeholders = ", ".join(f"{col}=%s" for col in column_ids)
        query = f"UPDATE {self.table_dropdown.get()} SET {placeholders} WHERE {column_ids[0]}=%s"
        try:
            self.cursor.execute(query, list(
                updated_values.values()) + [values[0]])
            self.conn.commit()
            self.load_table()
            edit_window.destroy()
            messagebox.showinfo("Success", "Row updated successfully.")
        except mysql.connector.Error as err:
            messagebox.showerror("Error", f"Failed to update row: {err}")

    save_button = tk.Button(edit_window, text="Save", command=save_changes)
    save_button.grid(row=len(column_ids), column=1, pady=10)
```

Algorithm

The edit_row method allows the user to edit the data of a selected row in the table.

1. Check Row Selection: It checks if a row is selected in the Treeview. If no row is selected, the function exits.
2. Get Row Data: It retrieves the values of the selected row and the column names from the Treeview.
3. Create Edit Window: It opens a new window (using Toplevel) where the user can edit the values for each column in the selected row. Each column gets a label and an entry field, with the current value of the row pre-filled in the entry fields.
4. Save Changes: The save_changes function collects the updated values from the input fields, constructs an SQL UPDATE query to modify the row in the database, and executes it.
5. Commit Changes: If the update is successful, it commits the changes to the database, reloads the table to show the updated data, and closes the edit window. If an error occurs, it shows an error message.
6. Success Message: After a successful update, a success message is displayed.