INFORMATICS INSTITUTE OF TECHNOLOGY

UNIVERSITY OF WESTMINSTER

University Of Westminster

BSc.(Hons) Computer Science

Software Development I 4COSC006C

Coursework Part C

Test Plan

Module	: 4COSC006C.2 Software Development I
Module Leader	: Mr. Poravi Guhanathan
Type of Assignment	: Individual (Part C)
Submission Date	: 29th of April 2024
Group	: H
Name	: W G Pasindu Dilshan
IIT Student Number	: 20230294
UoW Student Number	: W20831848

I. Abstract

The purpose of this code is to improve the Personal Finance Tracker usefulness by creating an intricate graphical user interface (GUI) using Python Tkinter. To improve modularity and scalability, I aim to develop GUI components as objects by embracing the concepts of object-oriented programming (OOP). Financial transactions will be loaded and displayed by the application with seamless integration from JSON files. I created a comprehensive search function that will enable users to discover transactions quickly based on multiple parameters, so empowering them to manage their finances. In addition, my program will have a sorting function similar to a file explorer, which will let users effectively organize and evaluate financial data. My goal is to provide a user-friendly and effective tool for managing personal finances by integrating these components.

II. Acknowledgement

My sincere gratitude goes out to our module leader and professor, Mr. Guhanathan Poravi, as well as our tutorial instructor, Mr. Lakshan Costa, whose constant assistance and direction enabled me to finish my assignment to the highest standard. I also appreciate how quickly he responded to my questions, and how much I valued his advice and assistance. I also want to thank my colleagues for their unwavering support and encouragement in helping me finish this assignment.

Contents

List of Figures

Table of contents

Coursework 03 Specification: Enhanced Personal Finance Tracker (GUI Implementation with Tkinter and OOP)

Overview:

Building on your knowledge of Python, dictionaries, and file I/O, your next challenge is to enhance the Personal Finance Tracker by developing a graphical user interface (GUI) using Tkinter. This advanced version should not only display the information from a provided JSON file but also incorporate object-oriented programming (OOP) concepts for the GUI components. Additionally, your application will include a search function and a sorting feature, similar to a file explorer, to manage and analyze financial transactions more effectively.

Objectives:

1. Integrate a GUI using Tkinter and OOP concepts.
2. Load and display data from a JSON file upon GUI invocation.
3. Implement search and sorting functionalities within the GUI.

Design

3.1 Data Structure

Search Functionality: Allow users to search for transactions based on attributes such as date, amount, or type of expense. Implement filtering to display only the transactions that match the search criteria.

Sorting Feature: Implement a feature where clicking on a column heading in the transaction display table sorts the data based on that column. The sorting should toggle between ascending and descending order.

JSON Integration: Use the JSON file format for loading and saving transactions. Ensure your application correctly parses the JSON structure as specified in the original assignment.

Functions

- **Create widgets:** This method organizes the GUI elements into sections for adding transactions, searching transactions, and displaying transactions. It ensures that the user interface is functional for managing personal finance data.
- **Load Transactions:** This function ensures that transaction data is loaded from a JSON file if it exists, and returns an empty dictionary otherwise, providing a clean way to handle file loading errors.
- **Save Transactions:** This function writes the transactions stored in the self.transactions dictionary to the specified JSON file. It overwrites the content of the file if it already exists.

- **Display Transactions:** This function updates the display of transactions in the GUI by removing existing entries from the Treeview and adding new entries based on the provided transaction data.
- **Search Transactions:** Enables users to search for transactions based on a query and a selected attribute, and then updates the display to show the search results.
- **Sort Column:** This function sorts the items in a Treeview widget based on the values of a specified column and toggles the sorting order each time it's called.
- **Launch CLI Menu:** This function for saves any transactions made in the GUI application, closes the GUI window, and returns control to a main menu in a command-line interface. It's essentially transitioning from a graphical interface to a command-line interface.

Python Code

```
import tkinter as tk

from tkinter import ttk

import json

from datetime import datetime


# Global variable to store transactions

transactions = {}


# File handling functions

filename = "transactions.json"



# Function to load transactions from JSON

def load_transactions():

    global transactions

    try:

        with open(filename, "r") as file:

            transactions = json.load(file)  # Open a JSON file and load the transactions.

    except FileNotFoundError:

        print("No transactions found.")



# Function to save transactions to JSON

def save_transactions():

    with open(filename, "w") as file:
```

```
    json.dump(transactions, file, indent=4, default=str)  # Store transactions with indentation in a JSON file.
    print("Transaction Saved Successfully.!")



# Function to read bulk transactions from file
def read_bulk_transactions_from_file(filename):
  global transactions
  try:
    with open(filename, 'r') as file:
      data = json.load(file)  # Load information from the given file
      for transaction in data:
        try:
          expense_type = transaction.get("type")
          if expense_type is None:
            print("Skipping transaction without type information:", transaction)
            continue
          transactions.setdefault(expense_type, []).append(transaction)
        except AttributeError:
          print("Error parsing transaction:", transaction)
  except FileNotFoundError:
    print("File can not found.")  # Take action if the file is missing.



# Function to add a transaction
def add_transaction():
  while True:
```

```
        expense_type = input("Enter expense type: ")
        try:
            amount = float(input("Enter amount: "))
            break  # Exit the loop if the amount input is successfully converted to float
        except ValueError:
            print("Please Enter a Valid Amount.!")
            continue
    while True:
        date_str = input("Enter date (YYYY-MM-DD): ")
        try:
            date = datetime.strptime(date_str, "%Y-%m-%d").date()
            break
        except ValueError:
            print("Please Enter a Valid Date in YYYY-MM-DD format.!")
            continue
    transaction = {"amount": amount, "date": date}
    transactions.setdefault(expense_type, []).append(transaction)  # Append a new transaction to the
dictionary
    print("Transaction added successfully.")



# Function to view all transactions
def view_transactions():
    global transactions  # Call the global variable using the global function
    for expense_type, expense_list in transactions.items():
        print(expense_type)
        for expense in expense_list:
```

```
        print(f"  Amount: {expense['amount']}, Date: {expense['date']}")  # Print each transaction



# Function to update a transaction
def update_transaction():
    view_transactions()
    global transactions
    # Display the current transactions using a user-defined function
    expense_type = input("Enter expense type to update: ")
    if expense_type in transactions:
        index = int(
            input("Enter index of transaction to update: ")) - 1  # Python uses zero indexing & we use -1 for user
input
        if 0 <= index < len(transactions[expense_type]):
            amount = float(input("Enter new amount: "))
            while True:
                date_str = input("Enter new date (YYYY-MM-DD): ")
                try:
                    date = datetime.strptime(date_str, "%Y-%m-%d").date()
                    break
                except ValueError:
                    print("Please Enter a Valid Date in YYYY-MM-DD format.!")
                    continue
            transactions[expense_type][index] = {"amount": amount, "date": date}
            print("Transaction updated successfully.")
        else:
            print("Invalid index.")
```

```
    else:
        print("Expense type not found.")



# Function to delete a transaction
def delete_transaction():
    # Display the current dictionary of the transaction
    view_transactions()  # Display current transactions


    # Check if the transaction dictionary is empty
    global transactions
    expense_type = input("Enter expense type to delete: ")
    if expense_type in transactions:
        index = int(
            input("Enter index of transaction to delete: ")) - 1  # Python uses zero indexing & we use -1 for user
input
        if 0 <= index < len(transactions[expense_type]):
            del transactions[expense_type][index]
            print("Transaction deleted successfully.")
        else:
            print("Invalid index.")
    else:
        print("Expense type not found.")



# Function to calculate and display the summary
def display_summary():
```

```
    total = 0
    for expense_list in transactions.values():
        for expense in expense_list:
            total += expense["amount"]
    print(f"Total expenses: LKR.{total}")



# Main menu from Part B
def main_menu():
    global filename  # Declare filename as global
    load_transactions()  # Load transactions initially
    while True:
        # Main menu options
        print("\n|| Financial Tracker || ")
        print("------------------------")

        print("1. Add Transaction")
        print("2. View Transactions")
        print("3. Update Transaction")
        print("4. Delete Transaction")
        print("5. Display Summary")
        print("6. Bulk Read Transactions from File")
        print("7. Save Transactions")
        print("8. Switch to GUI")
        print("9. Exit")
        choice = input("Enter your choice: ")
```

```
    if choice == '1':
        add_transaction()
    elif choice == '2':
        view_transactions()
    elif choice == '3':
        update_transaction()
    elif choice == '4':
        delete_transaction()
    elif choice == '5':
        display_summary()
    elif choice == '6':
        filename = input("Enter filename to read transactions from: ")
        read_bulk_transactions_from_file(filename)
    elif choice == '7':
        save_transactions()
    elif choice == '8':
        start_gui()
    elif choice == '9':
        save_transactions()
        print("Exiting the program..!!")
        break
    else:
        print("Invalid choice. Please try again.")
```

```
# GUI part (Part C)
class FinanceTrackerGUI:
```

```
def __init__(self, root):
    self.root = root
    self.root.title("Personal Finance Tracker")
    self.create_widgets()
    self.transactions = self.load_transactions(filename)
    self.display_transactions(self.transactions)

def create_widgets(self):
    menu_bar = tk.Menu(self.root)
    self.root.config(menu=menu_bar)

    file_menu = tk.Menu(menu_bar, tearoff=0)
    file_menu.add_command(label="Launch CLI Menu", command=self.launch_cli_menu)
    menu_bar.add_cascade(label="File", menu=file_menu)

    # Frame for searching transactions
    search_frame = tk.LabelFrame(self.root, text="Search Transactions", )
    search_frame.pack(padx=10, pady=10, fill=tk.BOTH, expand=True)

    # Search entry
    ttk.Label(search_frame, text="Search:").grid(row=0, column=0, padx=5, pady=5)
    self.search_var = tk.StringVar()  # Variable to hold search query
    self.search_entry = ttk.Entry(search_frame, textvariable=self.search_var)
    self.search_entry.grid(row=0, column=1, padx=5, pady=5)

    # Search criteria dropdown
    ttk.Label(search_frame, text="Search by:").grid(row=0, column=2, padx=5, pady=5)
```

```
    self.search_criteria = ttk.Combobox(search_frame,
                                        values=["Date",
                                                "Amount"])  # Dropdown for search criteria
    self.search_criteria.current(0)  # Set default selection to first option
    self.search_criteria.grid(row=0, column=3, padx=5, pady=5)  # Placing dropdown in the search
frame

    # Search button
    search_button = ttk.Button(search_frame, text="Search",
                               command=self.search_transactions)  # Button to trigger search
    search_button.grid(row=0, column=4, padx=5, pady=5)

    # Frame for displaying transactions
    display_frame = ttk.LabelFrame(self.root, text="Transactions")
    display_frame.pack(padx=10, pady=10, fill=tk.BOTH,
                       expand=True)  # Packing the display_frame with some padding, filling both horizontally and
vertically, and expanding it

    # Treeview for displaying transactions
    self.tree = ttk.Treeview(display_frame, columns=("Date", "Description", "Amount"),
show='headings')
    # Setting headings for other columns and linking them to sorting functions
    self.tree.heading("Date", text="Date", command=lambda: self.sort_column("Date", False))
    self.tree.heading("Description", text="Description", command=lambda:
self.sort_column("Description", False))
    self.tree.heading("Amount", text="Amount", command=lambda: self.sort_column("Amount",
False))
    self.tree.pack(side=tk.LEFT, fill=tk.BOTH,
```

```
                expand=True)  # Packing the Treeview to the left, filling both horizontally and vertically


    # Scrollbar for the Treeview
    scrollbar = ttk.Scrollbar(display_frame, orient="vertical",
                              command=self.tree.yview)  # Creating a vertical scrollbar
    scrollbar.pack(side=tk.RIGHT, fill=tk.Y)  # Packing the scrollbar to the right, filling vertically
    self.tree.configure(
        yscrollcommand=scrollbar.set)  # Configuring the Treeview to use the scrollbar for vertical
scrolling


    #  Variable to keep track of sorting order for different columns
    self.sort_order = {"Date": False, "Description": False, "Amount": False}


  def load_transactions(self, filename):
    try:
        with open(filename, "r") as file:
            transactions = json.load(file)
        return transactions
    except FileNotFoundError:
        return {}


  def display_transactions(self, transactions):
    # Remove existing entries
    for row in self.tree.get_children():
        self.tree.delete(row)


    # Add transactions to the treeview
```

```
    for category, entries in transactions.items():
        for i, entry in enumerate(entries, 1):
            self.tree.insert("", "end", text=f"{category}-{i}", values=(
                entry["date"],
                entry.get("description", category),  # Fetch description if available, else empty string
                entry["amount"]))

def search_transactions(self):
    query = self.search_var.get().lower()
    criteria = self.search_criteria.get()
    results = {}
    for category, transaction_list in self.transactions.items():
        for i, transaction in enumerate(transaction_list, 1):
            # Convert all fields to lowercase for case-insensitive search
            date_str = str(transaction["date"]).lower()
            description = transaction.get("description", "").lower()
            amount_str = str(transaction["amount"]).lower()

            if criteria == "Date" and query in date_str:
                results.setdefault(category, []).append(transaction)
            elif criteria == "Description" and query in description:
                results.setdefault(category, []).append(transaction)
            elif criteria == "Amount" and query in amount_str:
                results.setdefault(category, []).append(transaction)
    self.display_transactions(results)

def sort_column(self, column, reverse):
```

```
        data = [(self.tree.set(child, column), child) for child in self.tree.get_children('')]
        data.sort(reverse=reverse)
        for index, (val, child) in enumerate(data):
            self.tree.move(child, '', index)
        # Toggle the sort order
        self.sort_order[column] = not self.sort_order[column]

    def save_transactions(self):
        with open(filename, "w") as file:
            json.dump(self.transactions, file, indent=4, default=str)
        print("Transactions saved successfully!")

    def launch_cli_menu(self):
        self.save_transactions()
        self.root.destroy()  # Close the GUI window
        main_menu()  # Switch back to the CLI


def start_gui():
    root = tk.Tk()
    app = FinanceTrackerGUI(root)
    root.mainloop()


def main():
    main_menu()
```

```
if __name__ == "__main__":
    main()
```

Test Plan

Test Cases for Main Program

Test No.	Test Case	Input	Expected Output	Actual Output	Pass or fail
1	Run in IDLE	Check the program run in IDLE correctly	Must display the Personal Finance Tracker GUI	Displayed The program and ran correctly.	Pass
	Exit the program	Close window button clicked	Must Exit the program	Exited the program correctly.	Pass

Table 1-Test Cases for the main program

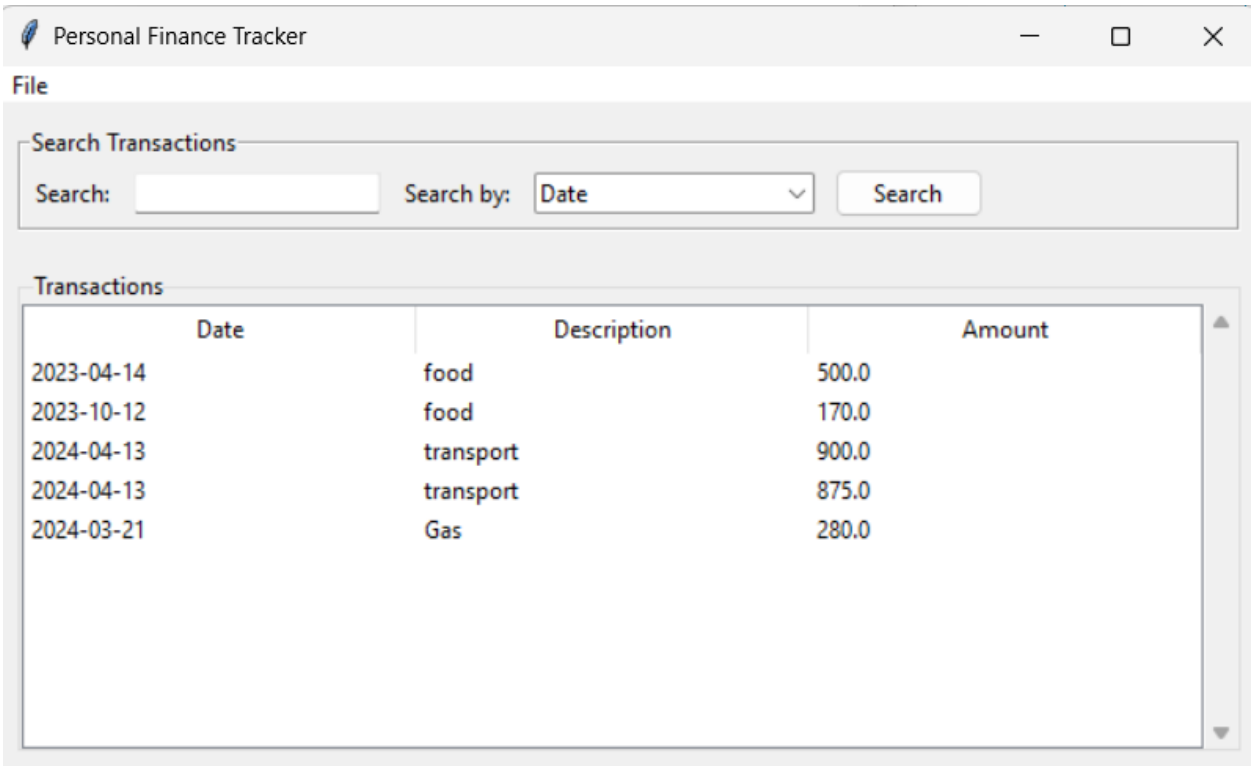

Figure 1- Main program test case

Test Case for Searching Transactions:

Test No.	Test Case	Input	Expected Output	Actual Output	Pass or fail
2	Searching Transactions by Type	Enter a search query in the search field and select search criteria from the dropdown and select “Type”. Then, click the "Search" button.	Transactions matching the search criteria should be displayed in the Treeview widget.	Displayed matching transactions related to Type	Pass
3	Searching Transactions by Date	Enter a search query in the search field and select search criteria from the dropdown and select “Date”. Then, click the "Search" button.	Transactions matching the search criteria should be displayed in the Treeview widget.	Displayed matching transactions related to date	Pass
4	Searching Transactions by Amount	Enter a search query in the search field and select search criteria from the dropdown and select “Amount”. Then, click the "Search" button	Transactions matching the search criteria should be displayed in the Treeview widget.	Displayed matching transactions related to amount	Pass

Table 2-Test Cases for Searching Transactions

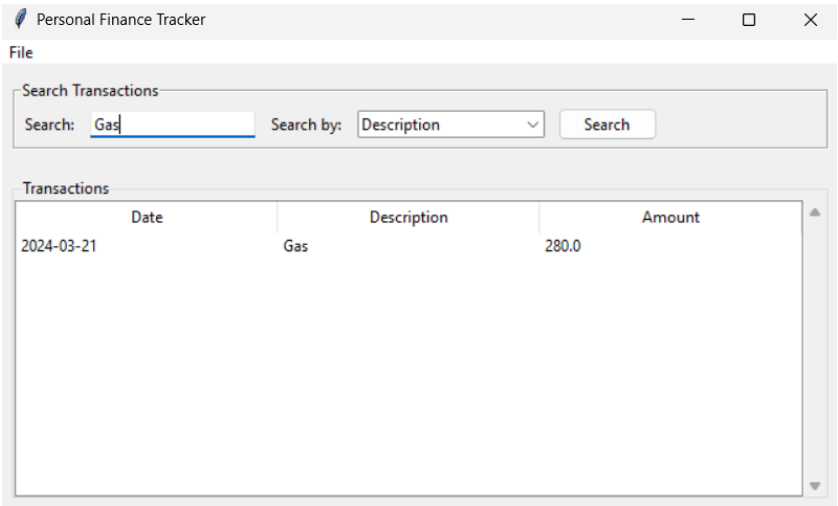

Figure 2-Searching Transactions by Type

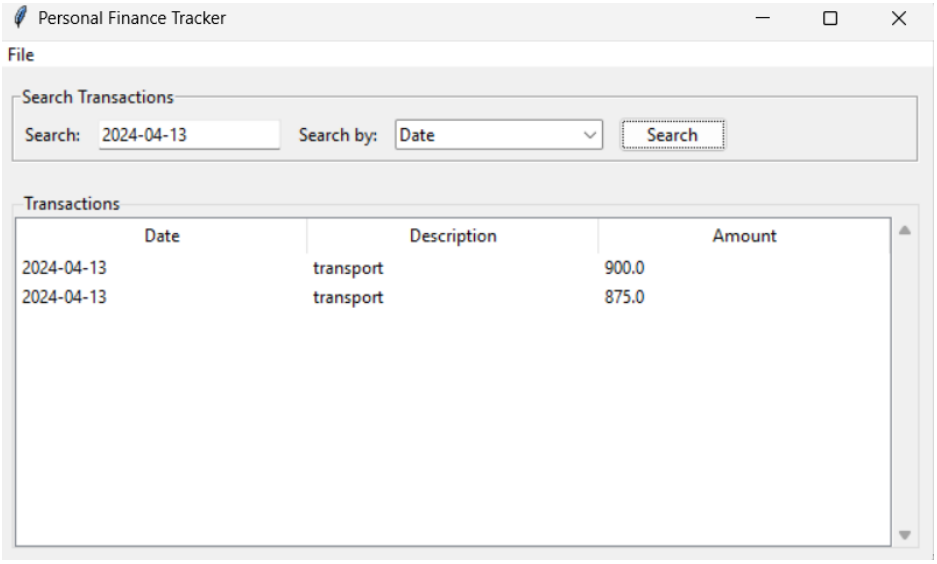

Figure 3-Searching Transactions by Date

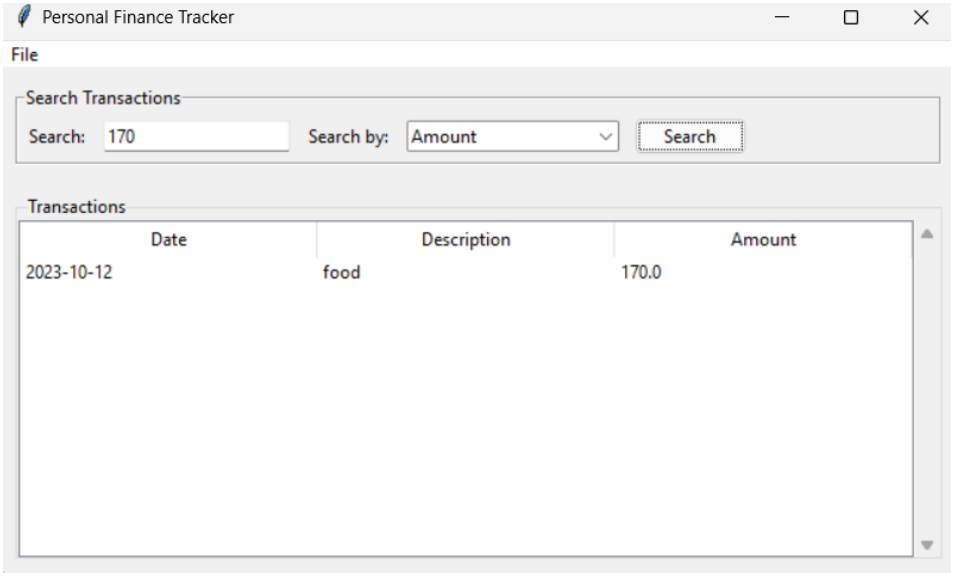

Figure 4-Searching Transactions by Amount

Test Cases for Sorting Transactions:

Test No.	Test Case	Input	Expected Output	Actual Output	Pass or fail
5	Sorting Transactions by Date	Click on the "Date" column header in the treeview.	Transactions should be sorted in ascending order based on the date.	Transactions are successfully sorted in ascending order based on the date.	Pass
6	Sorting Transactions by Description	Click on the headers of the “Description” in the treeview	Transactions should be sorted in ascending order based on the description.	Transactions are sorted in successfully ascending order based on the description.	Pass
7	Sorting Transactions by Amount	Click on the "Amount" column header in the treeview.	Transactions should be sorted in ascending order based on the amount.	Transactions are successfully sorted from Lowest price to the Highest price order based on the amount.	Pass

Table 3- Test Cases for Sorting Transactions

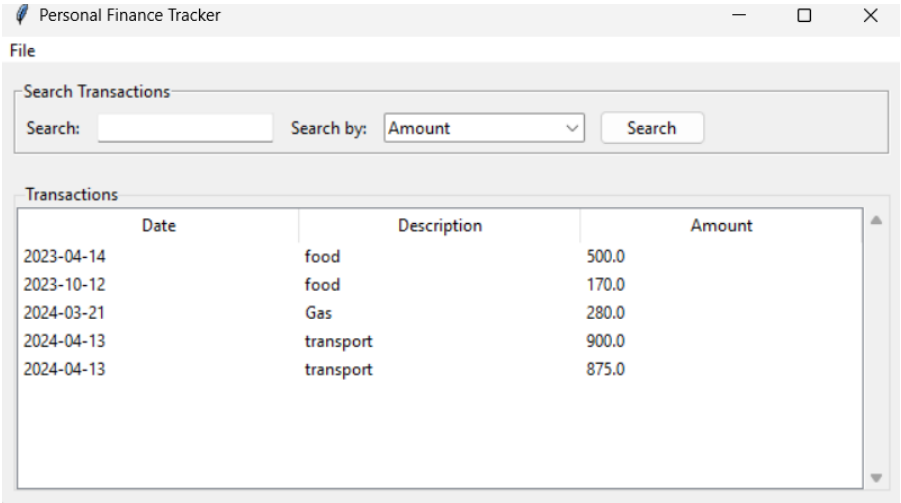

Figure 5-Sorting Transactions by Date

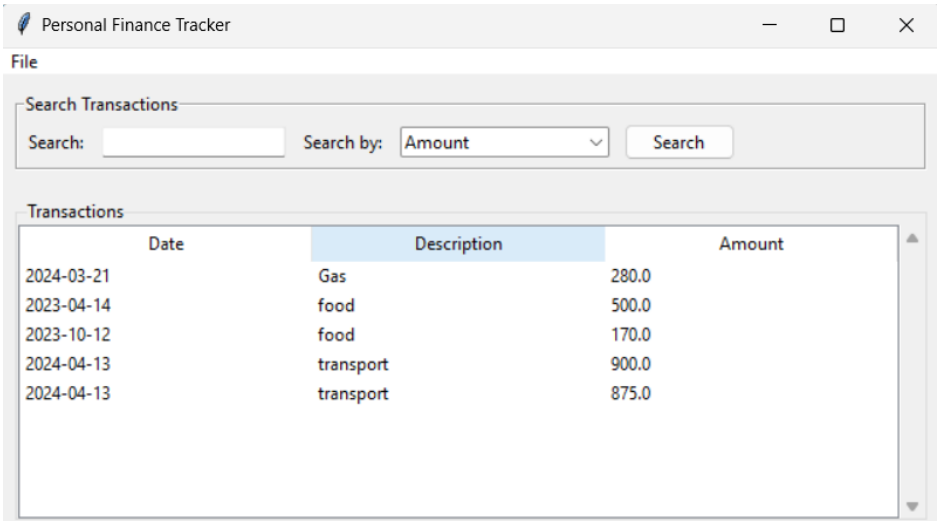

Figure 6-Sorting Transactions by Description

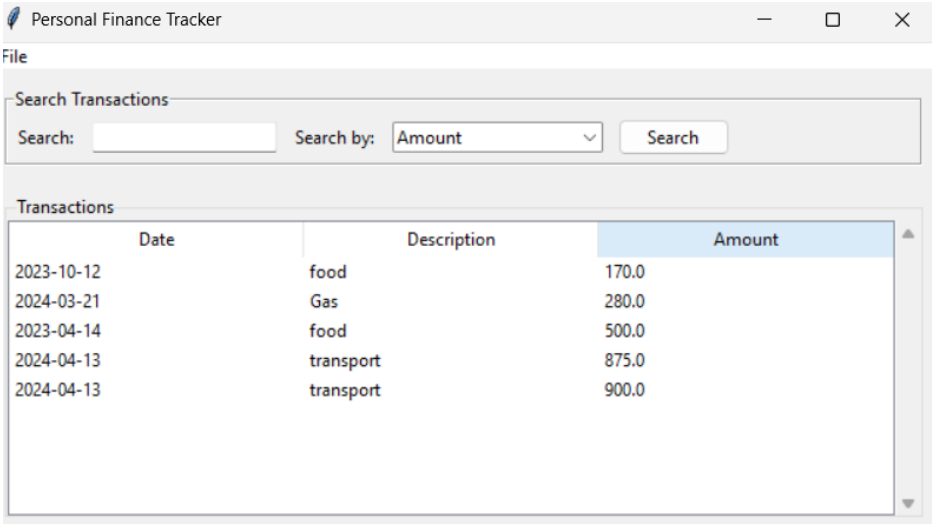

Figure 7-Sorting Transactions by Amount

transactions.json > ...

```
{
    "food": [
        {
            "amount": 500.0,
            "date": "2023-04-14"
        },
        {
            "amount": 170.0,
            "date": "2023-10-12"
        }
    ],
    "transport": [
        {
            "amount": 900.0,
            "date": "2024-04-13"
        },
        {
            "amount": 875.0,
            "date": "2024-04-13"
        }
    ],
    "Gas": [
        {
            "amount": 280.0,
            "date": "2024-03-21"
        }
    ]

}
```

Figure 8- Test Case 9 - Successful creation of JSON file & save correct transaction data.

Test Summary

Test Objectives

• Ensure all features function correctly according to requirements.

• Validate accurate recording, searching, and sorting of transactions.

• Confirm successful saving and loading of transaction data.

Test Results

1. **Searching Transactions:** Validate the functionality of searching transactions by type, date, and amount.
2. **Sorting Transactions**: Confirm the correct sorting of transactions by date, type , and amount.
3. **Loading Transactions from File**: Verify the loading of transactions from a file and their accurate display.
4. **Handling Nonexistent File:** Ensure graceful handling of scenarios where the file does not exist.

Conclusion

The personal finance tracker developed using Tkinter offers a robust solution for individuals seeking to manage their finances effectively. With its user-friendly interface, comprehensive features, and seamless transition between GUI and CLI, this application provides users with a convenient tool for tracking, analyzing, and organizing their financial transactions.

****