Budget Spreadsheet Tracker

ECE508 - Python & Scripting Wksp

Final Project Report

Spring 2024 – Jean Paul Mugisha

May 27, 2024

Abram Fouts, Niko Nikolov

Abstract

The "***Budget Spreadsheet Tracker***" is an application that utilizes python and a set of user friend libraries. This application takes on a pre-defined *comma-separated value* (csv) with eight columns composed of credit card transactions containing card number, date, amount, reference_number, country, address, description and a category.

Once a csv is imported into the application four graphs are generated that contain top 10 most expensive purchases along with dates of purchases, the credit card balance over the course of the last 30 days, the total sum spent in each category, and the amount of categorical transactions taken place. The user can add another csv or exit if they would like.

Github Repository: https://github.com/abfouts/ece508_final_project

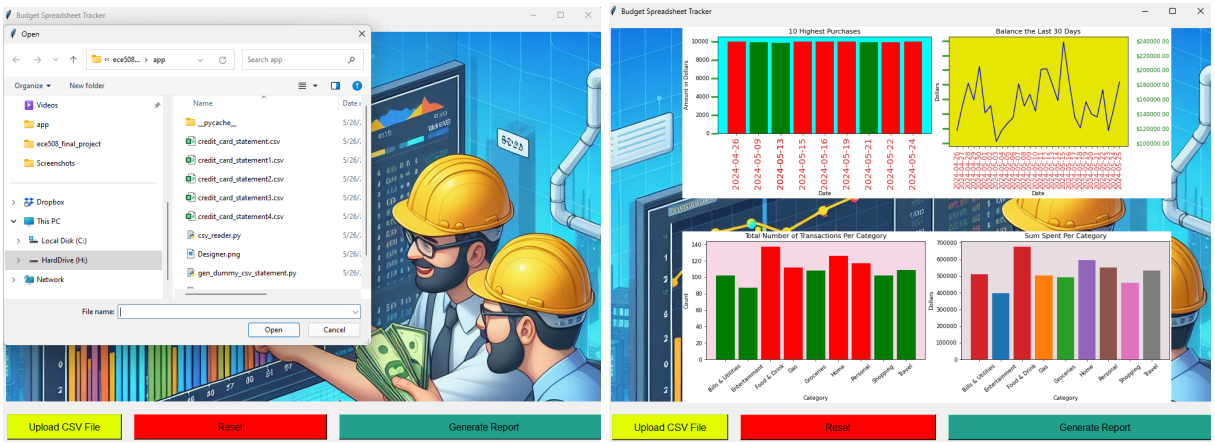

Figure 1 & 2. Opening a directory window and presenting the four charts

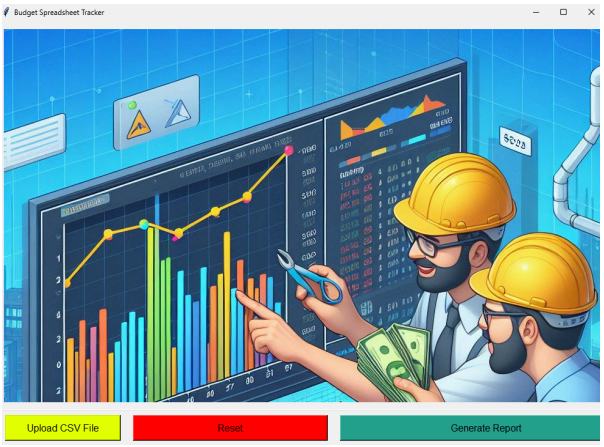

Figure 3. Main UI at the app's start and after issuing a Reset via button

Introduction

Background

A credit card statement reading application that takes on a csv that is typically provided by the credit card company. This application presents multiple forms of data analysis for the user once a valid csv has been identified.

Objective

The goal of this application is to be broken down into three categories.

1. Create a GUI that contains a local directory search for any Linux, OSX, and Windows, a reset button and generate button.
2. Create a class that takes on a csv and parses it into data types that flow into graphs easily.
3. Generate the plots inside of the GUI with the created class that generates graph data from the CSV.

Problem Statement

Create an application that takes on a credit card statement report and generate useful information to manage the users finances.

Methodology

Create an object oriented programming approach to the design to create classes that utilize powerful libraries. Using libraries and built in file IO functions combined with custom logic will generate a report from a provided csv.

Tools and Libraries

Tools:

- Python3 (tested with Python 3.12.3
- Github (version control)

Libraries:

- Tkinter (GUI)
- Matplotlib (Plots)
- OS
- Pandas (Data manipulation)
- Path
- Faker (Create fake data sets)
- numpy

Design and Approach

We kept our approach to the project simple but effective, splitting the app into two main pieces: one for grabbing data, and another for organizing and working with it. The GUI, handled by gui.py, provides the visual interface, while csv_reader.py takes care of the data extraction and manipulation behind the scenes. A separate script, gen_dummy_csv_statement.py, was also developed, which could potentially be integrated directly into the app in the future.

The app's structure is built around several classes, like "Backend", "PlotTop", etc., with the core "Application" class being the starting point for everything.

Within this class, a "MyCanvas" object is created to handle the background image setup. We grouped all the buttons together for easy adjustments, although this limited our ability to move them individually.

One challenge we faced was that some elements were on the same hierarchical level in the interface, meaning changes to padding in one area would affect another. This resulted in some fiddling to get the layout just right, especially after adding the bar charts and plots from the matplotlib library.

Since these graphs tended to overlap other elements, we divided them into top and bottom groups to manage their placement more easily. Positioning these plots correctly within the canvas required some trial and error, but we eventually got them to display as intended.

The bar charts themselves are a highlight, using a clear color scheme with green for values below the average and red for those above. Given more time, we would have loved to add features like tooltips, additional color coding, and more detailed annotations to the charts. Adding interactive elements, such as mouse hover responses, would have been another great enhancement.

Implementation Details

gui.py:

Application

```
class Application(tk.Tk):
      """Application class for the Budget Spreadsheet Tracker application"""

      def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)

      # Create the canvas, and the background image
      self.my_canvas = MyCanvas()
      self.my_canvas.grid(row=0, column=0, sticky="nsew")
      self.my_canvas.grid_columnconfigure(0, weight=1)
      self.my_canvas.grid_rowconfigure(0, weight=1)

      # Create the button structure
      self.my_buttons = Buttons()
      self.my_buttons.grid(sticky="nsew")
```

The Application class is the entry point of the application and is responsible for setting up the initial interface. It creates the canvas with its background image and arranges the buttons that will control the app's functionality. In essence, this class acts as a wrapper, initiating the application and providing the foundation for user interaction.

Canvas

```
class MyCanvas(Canvas):
      """Canvas class for the Budget Spreadsheet Tracker application"""

      def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)

      self.bg = tk.PhotoImage(file="Designer.png")
      self.my_canvas = Canvas(self, width=1000, height=600)

      self.my_canvas.create_image(0, 0, image=self.bg, anchor="nw")
      self.my_canvas.pack(
            fill="both",
            expand=True,
            padx=10,
            pady=10,
            ipadx=5,
            ipady=5,
            side="top",
      )
      self.my_canvas.pack_configure(fill="both", expand=True)
```

The MyCanvas class, derived from Tkinter's Canvas class, is primarily responsible for setting the background image and defining the area where plots will be displayed. The canvas size is intentionally smaller than its parent container to accommodate both the Buttons and MyCanvas elements. This class does not have any specific methods, as its primary function is to create and configure the Canvas for plot visualization.

Buttons

```
class Buttons(Button):
	"""Button class for the Budget Spreadsheet Tracker application"""

	def __init__(self, *args, **kwargs):
	super().__init__(*args, **kwargs)
	self.csv_path = ""
	self.csv_path = ""
	self.backend = Backend(self.csv_path)

	self.upload_csv_button = Button(
		self,
		text="Upload CSV File",
		command=self.backend.upload_csv_file,
		width=20,
		activebackground="#ffdf4f",
		bg="#e2ff00",
		fg="black",
		activeforeground="black",
		font=("Roboto", 12),
		borderwidth=2,
	)

	self.upload_csv_button.grid(
		row=3, column=0, sticky="nsew", padx=10, pady=5, ipady=5
	)

	self.reset_button = Button(
		self,
		text="Reset",
		command=self.backend.reset,
		width=30,
		bg="#ff0000",
		activebackground="#e23a08",
		fg="black",
		font=("Roboto", 12),
		borderwidth=2,
	)
	self.reset_button.grid(
		row=3, column=1, padx=10, sticky="nsew", pady=5, ipady=5, ipadx=20
	)

	self.generate_report_button = Button(
		self,
	  text="Generate Report",
```

```
            command=self.backend.generate_report,
            width=30,
            activeforeground="black",
            bg="#23a08e",
            activebackground="#a3ffb4",
            fg="black",
            font=("Roboto", 12),
            borderwidth=2,
        )
        self.generate_report_button.grid(
            row=3,
            column=2,
            padx=10,
            sticky="we",
            pady=5,
            ipadx=100,
            ipady=5,
          rowspan=3,
            columnspan=3,
        )

        self.generate_report_button.grid_columnconfigure(3, weight=1)
```

The Buttons class contains three buttons, intentionally varied in length to resemble a bar chart. This class instantiates three Tkinter Button objects, customizing their font, colors, text, width, and overall appearance. While each button can be individually modified, they are ultimately grouped into a single object rendered on the canvas using Tkinter's grid method. This grid method is employed twice: once for each button, and again for the entire structure of three buttons.

The three buttons, "Upload CSV," "Reset," and "Generate Report," each trigger corresponding methods within the "Backend" class. The Backend class is instantiated within the Buttons class, establishing a parent-child relationship that allows the graphs and plots to be displayed over the background. However, this approach has unintended consequences. The padding, width, and other properties of the buttons directly impact the appearance of the graphs. This limitation restricts our ability to easily adjust button placement, and in some cases, large graphs can cause the buttons to disappear below the visible canvas area, particularly on smaller screens.

Given more time, we would have reconsidered our object placement strategy. The grid() method offers numerous configuration options, requiring significant experimentation to achieve our desired button layout.

PlotTop

```
class PlotTop(Canvas):
	"""Plot class for the Budget Spreadsheet Tracker application"""

	def __init__(self, balance_per_day, ten_highest_transactions, *args, **kwargs):
	super().__init__(*args, **kwargs)

	if balance_per_day is None or ten_highest_transactions is None:
		return

	self.balance = balance_per_day
	self.transactions = ten_highest_transactions
	self.configure(bg="#020101")

	self.create_figure1()
	self.create_figure2()

	def create_figure1(self):
	"""Creates the first figure"""
	# Create figure 1
	transaction_df = pd.DataFrame(self.transactions)
	self.figure1 = Figure(figsize=(6, 4), dpi=65)
	self.ax1 = self.figure1.add_subplot()
	self.bar1 = FigureCanvasTkAgg(self.figure1, self)
	self.figure1.set_layout_engine("compressed")
	y_values = transaction_df["amount"]
	facecolors = [
		"red" if y > transaction_df["amount"].mean() else "green" for y in
y_values
	]
	edgecolors = facecolors
	self.ax1.bar(
		transaction_df["date"],
		transaction_df["amount"],
		color=facecolors,
		edgecolor=edgecolors,
	)

	self.ax1.tick_params(
		axis="x", labelcolor="tab:red", labelrotation=45, labelsize=16
	)
	self.ax1.tick_params(axis="y", color="tab:green", size=12, width=3)
	self.ax1.set_title("10 Highest Purchases")
	self.ax1.set_facecolor("#00ffff")
```

```
        self.ax1.tick_params(axis="both", which="minor", labelsize=10)
        self.ax1.set_xlabel("Date")
        self.ax1.set_ylabel("Amount in Dollars")
        self.ax1.set_xticks(transaction_df["date"])
        self.ax1.set_xticklabels(transaction_df["date"], rotation=90)

        self.bar1.get_tk_widget().pack(
            expand=True, side=tk.LEFT, anchor="w", fill="both"
        )

        def create_figure2(self):
        """Creates the second figure"""

        balance_df = pd.DataFrame(self.balance)
        self.figure2 = Figure(figsize=(6, 4), dpi=65)
        self.figure2.set_layout_engine("compressed")

        self.ax2 = self.figure2.add_subplot()
        self.line2 = FigureCanvasTkAgg(self.figure2, self)

        self.ax2.plot(balance_df["date"], balance_df["balance"], "b", label="Balance")
        self.ax2.set_title("Balance the Last 30 Days")

        self.ax2.set_facecolor("#e6e600")
        self.ax2.set_xlabel("Date")
        self.ax2.set_ylabel("Dollars")
        self.ax2.tick_params(axis="both", which="minor", labelsize=12)
        self.ax2.tick_params(
            axis="x", labelcolor="tab:red", labelrotation=45, labelsize=12
        )
        self.ax2.tick_params(axis="y", color="tab:green", size=12, width=3)
        self.ax2.yaxis.set_major_formatter("${x:1.2f}")

        self.ax2.yaxis.set_tick_params(
            which="major", labelcolor="green", labelleft=False, labelright=True
        )
        self.ax2.set_xticks(balance_df["date"])
        self.ax2.set_xticklabels(
            balance_df["date"],
            rotation=90,
        )

        self.line2.get_tk_widget().pack(
            expand=False, side=tk.LEFT, anchor="w", fill="both"
        )
```

The PlotBottom and PlotTop classes are largely identical. Initially, we planned for only two charts, but as the project's scope expanded, we incorporated two additional charts. Managing all four as a single object proved cumbersome, so we separated them into distinct "top" and "bottom" objects. Despite their names, these graphs were primarily displayed side-by-side rather than vertically until the final stages of development.

An initial conditional statement within these classes checks if any diagrams exist, preventing further initialization in the case of a reset or initial app launch. The actual graph instantiation occurs within the Backend class, triggered by the "Generate Report" button press. This action invokes the PlotTop and PlotBottom classes, creating the respective graphs. Importantly, the placement of PlotTop directly influences how PlotBottom can be positioned on the canvas.

A crucial element in achieving the desired layout was the use of the "sticky" property within Tkinter's grid() method. By selecting from options like "e" (east), "w" (west), "s" (south), and "n" (north), we could center the graphs on the screen. However, this required trial and error with various settings, consuming a considerable amount of time.

The general process for creating a plot is as follows:

1. Obtain the dataframe (provided by the Backend via csv_reader).
2. Create a new Figure object from Tkinter.
3. Set properties like size and dpi for the Figure.
4. Create the bar chart using FigureCanvasTkAgg, with the Figure as its parent.
5. Configure the bar chart with details such as x and y labels, title, ticks, and colors.
6. Display the graph using pack() and its options.
7. For the graphs, we employed the pack() method instead of grid(), utilizing the "anchor" option (similar to "sticky" in grid()) and the "side" option. The "side" option determines placement relative to sibling objects. For instance, if one object is on the left side, specifying side=LEFT for the next object will position it directly to the right.

Similar to the buttons, the graphs can be modified individually or as a combined object.

Backend

```
class Backend:
	"""Backend class for the Budget Spreadsheet Tracker application"""

	def __init__(self, csv_path=None):
	self.csv_path = csv_path
	self.csv_reader = None
	self.balance_per_day = None
	self.highest_transactions = None
	self.chart_top = None
	self.chart_bottom = None
	self.buttons = None
	self.chart = None
	self.category_sum = None
	self.category_count = None

	def upload_csv_file(self):
	"""Uploads a csv file"""
	self.csv_path = filedialog.askopenfilename(initialdir=os.getcwd())
	self.update_csv_path(self.csv_path)
	self.csv_reader = CSVReader(self.csv_path)
	return self.generate_report()

	def reset(self):
	"""Resets the application to its initial state"""
	if self.chart_top is None or self.chart_bottom is None:
		return

	return self.chart_top.destroy(), self.chart_bottom.destroy()

	def generate_report(self):
	"""Generates a report"""
	self.csv_reader = CSVReader(self.csv_path)
	self.balance_per_day = self.get_balance_per_day()
	self.highest_transactions = self.get_ten_highest_transactions()
	self.category_sum = self.get_category_sum()
	self.category_count = self.get_category_count()

	if self.chart_top is not None or self.chart_bottom is not None:
		self.chart_top.destroy()
		self.chart_bottom.destroy()

	self.chart_top = PlotTop(self.balance_per_day, self.highest_transactions)
	self.chart_top.grid(
		row=0, column=0, sticky="n", padx=10, pady=10, ipadx=10, ipady=10
```

```
        )

        self.chart_bottom = PlotBottom(self.category_sum, self.category_count)
        self.chart_bottom.grid(
                row=0, column=0, sticky="s", padx=10, pady=10, ipadx=10, ipady=10
        )

        return self.chart_top, self.chart_bottom

        def get_balance_per_day(self):
        """Get the balance per day."""

        self.csv_balance_per_day = self.csv_reader.get_balance_per_day()

        return self.csv_balance_per_day

        def get_ten_highest_transactions(self):
        """Get the ten highest transactions."""

        self.csv_ten_highest_transactions = (
                self.csv_reader.get_ten_highest_transactions()
        )

        return self.csv_ten_highest_transactions

        def update_csv_path(self, new_path):
            """Update the CSV path."""
        self.csv_path = new_path
        return self.csv_path

        def get_category_sum(self):
        self.category_sum = self.csv_reader.get_category_sum()
        return self.category_sum

        def get_category_count(self):
        self.category_count = self.csv_reader.get_category_count()
        return self.category_count
```

The Backend class serves two primary purposes: separating the main functionalities that require execution and establishing a connection to the csv_reader.py file. Essentially, it acts as a bridge, with the added responsibility of initiating graph creation when the "Generate Report" button is clicked.

While it consumes data, the Backend class doesn't directly interact with it extensively. We delegated this responsibility to the csv_reader module, which could perhaps have been more aptly named "data_handler."

The key functionality worth noting is the generate_report method. Within this method, we read data from the currently selected CSV file, gathering information like balance, category sums and counts, and the highest transactions. Subsequently, new PlotTop and PlotBottom objects are instantiated, each containing two graphs. In this structure, the Backend object, despite its name, is the parent of the graphs, while also being a child of the Buttons class. A potential improvement would have been to avoid this confusing hierarchy, as it makes it difficult to predict and control visual changes.

Within the generate_report method, we use the grid() method to display the graphs/charts, employing the "sticky" property to anchor the PlotTop graph to the "north" (top) and the PlotBottom graph to the "south" (bottom). The remaining methods in the Backend class primarily serve as helpers and could have been designated as "private" to indicate their limited external use.

csv_reader.py

```
#!python
# -*- coding: utf-8 -*-
"""
###############################################################################
# Class       : CSVReader
# Description: Creates an object that uses a method that takes in a CSV and
#             parses the CSVs first row into unique column entry lists
#
# Date  : 05/18/2024
# Author: Abram Fouts, Niko Nikolov
###############################################################################
"""

import os
from pathlib import Path

import pandas as pd


class CSVReader:
      """The proper way to access the data is to use the dictionary
      mycsv = CSVReader()
```

```
      mycsv.read_csv('credit_card_statement.csv'
      """

      def __init__(self, path_to_csv):
      self.csv_contents = pd.DataFrame()
      self.csv_dict = {}
      self.file_dir = Path(__file__).parent
      self.path_to_csv = path_to_csv
      self.balance_per_day = {}
      self.ten_highest_transactions = {}

      # Provide the path to the csv
      def read_csv(self, path_to_csv=None):
      """Reads the csv and stores it in a pandas dataframe"""
      if self.path_to_csv == "" or self.path_to_csv is None:
            # If no path is provided, use the default path
            self.path_to_csv = os.path.join(self.file_dir,
"credit_card_statement.csv")

      self.csv_contents = pd.read_csv(
            self.path_to_csv,
            header=0,
            names=[
            "card_number",
            "date",
            "amount",
            "reference_number",
            "country",
            "address",
            "description",
            "category",
            ],
      )
      return self.csv_contents

      def get_csv_dict(self):
      """Returns the csv dictionary"""
      self.read_csv()
      self.csv_dict = self.csv_contents.to_dict()
      return self.csv_dict

      def get_ten_highest_transactions(self):
      """Returns the ten highest transactions"""
```

```
        self.read_csv()
        # Sort the dataframe by date
        self.csv_contents = self.csv_contents.sort_values("date")

        # Get the top 10 transactions and sort
        top10_transactions = self.csv_contents.nlargest(10, "amount")[
                ["date", "amount"]
        ]
        # Sort the dataframe by date
        top10_transactions = top10_transactions.sort_values("date")
        # Convert the dataframe to a date list
        dates = top10_transactions["date"].tolist()
        # Convert the dataframe to a amount list
        amounts = top10_transactions["amount"].tolist()
        # Create a dictionary with the date and amount
        self.ten_highest_transactions = {"date": dates, "amount": amounts}

        # Return the dictionary
        return self.ten_highest_transactions

        def get_balance_per_day(self):
        """Returns the balance per day"""
        # Read the csv and group by date
        self.read_csv()
        # List to store the balance for each day
        balance = []
        # List to store the date of the month
        day_of_the_month = []
        # Attribute to hold the 2 lists which we will use
        self.balance_per_day = {}
        # Sort the dataframe by date
        self.csv_contents = self.csv_contents.sort_values("date")
        # Iterate through the unique dates
        for date in self.csv_contents["date"].unique():
                # Filter the dataframe by the date
                tempdf = self.csv_contents[self.csv_contents["date"] == date]

                # Sum the amount column and round to 2 decimal places
                # Append the balance to the list
                balance.append(round(tempdf["amount"].sum(), 2))
                # Append the date to the list
                day_of_the_month.append(date)
```

```
        # Create a dictionary with the date and balance
        self.balance_per_day = {"date": day_of_the_month, "balance": balance}
        # Return the dictionary
        return self.balance_per_day

        def get_category_sum(self):
        self.read_csv()
           total_by_category = self.csv_contents.groupby("category", as_index=False)[
               "amount"
        ].sum()
        return total_by_category

        def get_category_count(self):
        self.read_csv()
        category_count = self.csv_contents.groupby("category", as_index=False).size()
        category_count = category_count.rename(columns={"size": "count"})
        return category_count
```

The CSVReader class serves as the core data handler in the file. Its primary function is to read the CSV file and encapsulate the data into a DataFrame using the Pandas library. This library greatly simplifies data manipulation tasks that would typically require custom parsing solutions. However, the reliance on an external library introduces a dependency that might necessitate the development of a custom parser in a commercial product scenario.

The class provides the following methods:

- __init__(self, path_to_csv): Initializes the CSVReader object with the path to the CSV file.
- read_csv(self, path_to_csv=None): Reads the CSV file and stores it in a Pandas DataFrame.
- get_csv_dict(self): Returns the CSV data as a dictionary.
- get_ten_highest_transactions(self): Returns the ten highest transactions from the CSV.
- get_balance_per_day(self): Returns the balance per day from the CSV.
- get_category_sum(self): Returns the total sum of amounts by category.
- get_category_count(self): Returns the count of transactions by category.

These methods perform similar functions, but their results are structured as dictionaries containing two arrays, each representing either the x or y axis for the graphs.

gen_dummy_csv_statement.py

The script below is used to create the CSV using the faker library, numpy and pandas.

It creates eight columns: card number, date, amount, reference_number, country, address, description and a category. Category randomly selects one option from the category list: Food & Drink, Entertainment, Travel, Shopping, Personal, Bills & Utilities, Gas, Home, and Groceries.

The script is not directly part of the app functionality. However, one might say that we could have had

```
"""
Faker is used to generate random data. https://faker.readthedocs.io/en/master/
The script saves the generated data to a CSV file.
"""

import random

import pandas as pd
from faker import Faker
from numpy import int64

# Initialize Faker for generating random data
fake = Faker()

# Number of transactions
NUM_TRANSACTIONS = 1000

# Generate random data
data = {
      # Fake credit card number, transaction date, details, amount,
      # reference number, country, address, description
      "card_number": [int64(fake.credit_card_number()) for _ in
range(NUM_TRANSACTIONS)],
      "date": [
      fake.date_between(start_date="-30d", end_date="today")
      for _ in range(NUM_TRANSACTIONS)
      ],
      "amount": [
      round(float(random.uniform(-1000, 10000)), 2) for _ in
range(NUM_TRANSACTIONS)
```

```
      ],
      "reference_number": [int64(fake.aba()) for _ in
range(NUM_TRANSACTIONS)],
      "country": [fake.country().rstrip(",").strip() for _ in
range(NUM_TRANSACTIONS)],
      "address": [
      fake.street_address().rstrip(",").strip() for _ in
range(NUM_TRANSACTIONS)
      ],
      "description": [fake.bs().rstrip(",").strip() for _ in
range(NUM_TRANSACTIONS)],
      "category": [
      fake.random_element(
            elements=(
            "Food & Drink",
            "Entertainment",
            "Travel",
            "Shopping",
            "Personal",
            "Bills & Utilities",
            "Gas",
            "Home",
            "Groceries",
            )
      )
      .rstrip(",")
      .strip()
      for _ in range(NUM_TRANSACTIONS)
      ],
}

# Create a DataFrame
df = pd.DataFrame(data)

# Save to CSV
df.to_csv("credit_card_statement.csv", index=False)
print("Dummy credit card statement saved to 'credit_card_statement.csv'")
```

Results

Output and Findings

Main UI

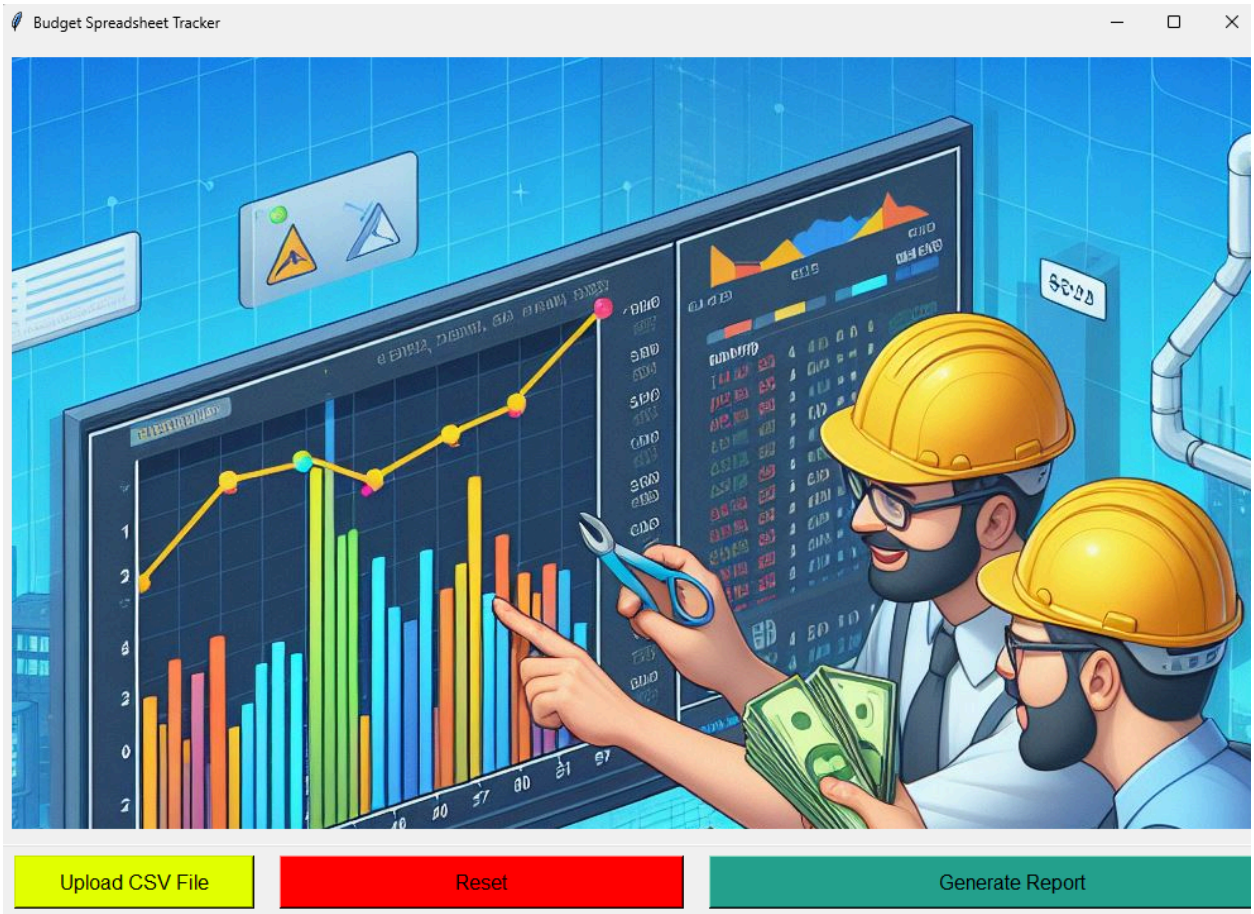

Figure 4. Main UI when the app starts or when Reset is clicked on

Upload CSV File UI

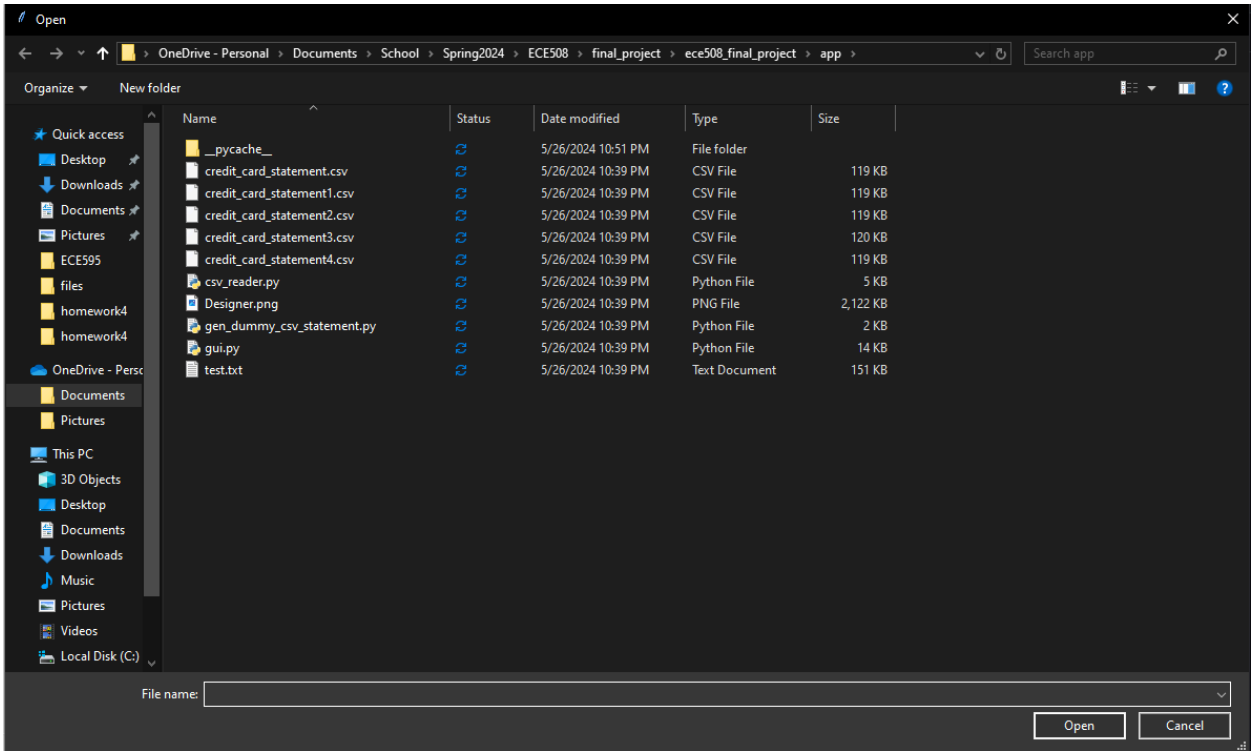

Figure 5. When clicking on Upload CSV File, it opens a window with the directory

After selecting a properly formatted csv the program will auto generate (or if the user named the file the default name, can select Generate Report). The user will see the four trend graphs that are described in the application.

Report UI

Figure 6. When Generate Report is clicked on, the four graphs appear

The user can select “Reset” if they would like to remove the graphs from the screen. The test shown during this example contains a csv with 1000 transactions. At reset the UI goes back to the Main UI layout.

Discussion

Challenges

The project presented several challenges, primarily stemming from the hierarchical structure of the Tkinter GUI framework. Grouping elements and managing their placement within the canvas proved to be a recurring issue, particularly with the addition of plots and charts. The interdependence between button properties and graph appearances also posed a constraint on design flexibility. Furthermore, the trial-and-error nature of configuring elements within the canvas, especially for achieving desired placement and visual outcomes, was time-consuming. Had more time been available, refining the object placement strategy and addressing these layout challenges would have been a priority.

Limitations

The project's primary limitations stem from the inherent constraints of the chosen GUI framework, Tkinter. Its hierarchical structure presented challenges in element placement and layout flexibility, particularly when accommodating dynamically generated plots and charts. Additionally, the reliance on an external library (Pandas) for data manipulation introduced a dependency that could potentially pose issues in a commercial setting. The lack of time for further development also prevented the implementation of more advanced features, such as interactive elements and additional data visualizations.

Conclusion

The project presented several challenges, primarily stemming from the hierarchical structure of the Tkinter GUI framework. Grouping elements and managing their placement within the canvas proved to be a recurring issue, particularly with the addition of plots and charts generated using the Matplotlib library. While Matplotlib's speed and flexibility allowed for the creation of visually appealing visualizations, time constraints limited our ability to fully explore its capabilities. The interdependence between button properties and graph appearances also posed a constraint on design flexibility. Furthermore, the trial-and-error nature of configuring elements within the canvas, especially for achieving desired placement and visual outcomes, was time-consuming. Had more time been available, refining the object placement strategy and addressing these layout challenges, along with a deeper exploration of Matplotlib's features, would have been a priority.

References

- Moore, Alan D. 2021. *Python GUI Programming with Tkinter - Second Edition: Design and Build Functional and User-Friendly GUI Applications*. 2nd ed. Birmingham: Packt Publishing
- J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007