Aim:

The aim of the Personal Finance Management System is to provide users with a comprehensive tool to manage their personal finances effectively. The application allows users to track income and expenses, set budgets, manage future bills, analyze financial data, and generate reports, all within a user-friendly graphical interface. The goal is to empower users to take control of their financial health by providing insights and tools for better decision-making.

Software: Pycharm

Theory

1. Architecture

The application follows an object-oriented programming (OOP) approach, encapsulated within a single class, PersonalFinanceManager. This class serves as the central hub for all functionalities, managing user interactions, data handling, and UI components.

2. Libraries Used

- Tkinter: This is the standard GUI toolkit for Python, which allows for the creation of desktop applications. It provides various widgets (buttons, labels, text fields, etc.) to build the user interface.
- Matplotlib: A plotting library used for creating static, animated, and interactive visualizations in Python. It is used here to generate graphs for income and expense analysis.
- JSON: This module is used to handle JSON data, allowing the application to read from and write to a JSON file for data persistence.
- OS: The os module is used to check for the existence of the data file and manage file paths.
- Datetime: This module is used for handling date and time operations, particularly for recording transaction dates.

3. Class Structure

The PersonalFinanceManager class contains several attributes and methods, organized as follows:

Attributes

- User Data:

- self.users: A dictionary to store user credentials (User ID and password).
- self.current_user: Stores the ID of the currently logged-in user.
- self.profile_details: A dictionary to store user profile information.
- Financial Data:
- self.balance: A float representing the current balance of the user.
- self.budget: A dictionary for storing budget categories and their allocated amounts.
- self.transactions: A list to keep track of all income and expense transactions.
- self.future_bills: A list to manage future bills.
- self.expense_limits: A dictionary to store limits on expenses by category.
- self.income_categories: A dictionary to track income by category.
- self.investments and self.assets: Dictionaries to track investments and assets, respectively.

Methods

- Data Management:
- load_data(): Reads user and financial data from a JSON file. It initializes attributes based on the data loaded.
- save_data(): Writes the current state of user and financial data back to the JSON file.
- User Interface Creation:
- create_login_widgets(): Sets up the login interface with input fields for User ID and password, along with buttons for login and registration.
- create_widgets(): Constructs the main application interface after a successful login, displaying the user's financial information and providing action buttons.
- User Interaction:
- login(): Validates user credentials and directs to the profile setup or main interface.
- register(): Allows new users to create an account by adding their credentials to the self.users dictionary.

- show_account_details(): Displays the user's profile information in a new window.
- Financial Operations:
- add_income(): Accepts income details (category and amount) and updates the balance and transactions list.
- add_expense(): Accepts expense details (category, amount, and date), checks for sufficient balance and expense limits, and updates the balance and transactions list.
- add_future_bill(): Allows users to add future bills with due dates and amounts.
- Expense Management:
- set_expense_limit(): Provides a UI to set limits on expenses for specific categories.
- get_total_expense_for_category(): Calculates total expenses for a given category.
- Data Visualization:
- show_income_analysis(): Generates a bar graph of income categorized by type using Matplotlib.
- show_expense_analysis(): Generates a line chart of expenses over time, categorized by daily, monthly, or yearly analysis.
- Reporting:
- generate_report(): Compiles a detailed financial report summarizing the user's financial status, including total income, expenses, future bills, and net worth.
- Application Control:
- reset_data(): Resets all user and financial data, prompting for confirmation to avoid accidental data loss.
- exit_application(): Saves data before closing the application.

4. Data Handling Logic

- Persistence:
- The application uses a JSON file (finance_data.json) to persist user and financial data. This allows users to close and reopen the application without losing their data.

- The load_data() method checks if the JSON file exists and loads the data into the respective attributes. The save_data() method is called after any changes to ensure the data is up-to-date.
- Input Validation:
- Throughout the application, input validation is implemented to ensure that users provide valid data. For example, when adding income or expenses, the program checks if the amount is a positive number and if the date is in the correct format (MM/DD/YYYY). This helps prevent errors and ensures data integrity.
- Error Handling:
- The application uses message boxes to inform users of errors, such as invalid input or insufficient funds. This enhances user experience by providing immediate feedback.

5. User Interface Design

- Layout:
- The UI is designed using Tkinter's grid and pack geometry managers, allowing for a responsive layout. The main window is divided into sections for navigation (side panel) and content display (main finance frame).
- Widgets:
- Various Tkinter widgets are used, including labels, buttons, entry fields, and frames. The use of colors and fonts enhances the visual appeal and usability of the application.
- Dynamic Updates:
- The UI updates dynamically based on user actions. For instance, when a user adds income or expenses, the current balance label and status bar are updated to reflect the changes immediately.

6. Data Visualization

- Graphs:
- The application utilizes Matplotlib to create visual representations of financial data. This includes bar graphs for income analysis and line charts for expense trends. These visualizations help users quickly grasp their financial situation and make informed decisions.
- Customization:

- The graphs are customized with titles, labels, and colors to improve readability and presentation. The use of annotations on the graphs provides additional context for significant data points.

Code:

```python
import tkinter as tk
from tkinter import messagebox
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import numpy as np
from datetime import datetime
import json
import os

class PersonalFinanceManager:  1 usage
    def __init__(self, master):
        self.master = master
        self.master.title("Personal Finance Management System")  # Set the title of the window
        self.master.geometry("800x600")  # Set the size of the window
        self.master.configure(bg="#f0f8ff")  # Set background color

        # User data
        self.users = {}  # Dictionary to store user credentials
        self.current_user = None  # Variable to store the currently logged-in user
        self.profile_details = {}  # Dictionary to store user profile details

        # Financial data
        self.balance = 0.0  # Current balance
        self.budget = {}  # Dictionary to store budget categories and amounts
        self.transactions = []  # List to store all transactions (income and expenses)
        self.future_bills = []  # List to store future bills
        self.expense_limits = {}  # Dictionary to store expense limits by category
        self.income_categories = {}  # Dictionary to store income by category
        self.investments = {}  # Dictionary to store investments
        self.assets = {}  # Dictionary to store assets

        # Load data if exists
        self.load_data()  # Load previously saved data from file

        # Create UI elements
        self.create_login_widgets()  # Create login interface

    def load_data(self):  1 usage
        """Load user and financial data from a JSON file."""
        if os.path.exists("finance_data.json"):  # Check if the data file exists
            with open("finance_data.json", "r") as file:
                data = json.load(file)  # Load data from the file
                # Populate the data attributes from the loaded data
                self.users = data.get("users", {})
                self.balance = data.get("balance", 0.0)
                self.budget = data.get("budget", {})
                self.transactions = data.get("transactions", [])
                self.future_bills = data.get("future_bills", [])

                self.expense_limits = data.get("expense_limits", {})
                self.income_categories = data.get("income_categories", {})
                self.investments = data.get("investments", {})
                self.assets = data.get("assets", {})
                self.profile_details = data.get("profile_details", {})  # Load profile details
```

```python
def save_data(self):  6 usages
    """Save user and financial data to a JSON file."""
    data = {
        "users": self.users,
        "balance": self.balance,
        "budget": self.budget,
        "transactions": self.transactions,
        "future_bills": self.future_bills,
        "expense_limits": self.expense_limits,
        "income_categories": self.income_categories,
        "investments": self.investments,
        "assets": self.assets,
        "profile_details": self.profile_details,  # Include profile details in saved data
    }
    with open("finance_data.json", "w") as file:
        json.dump(data, file)  # Save data to the file
```

```python
def create_login_widgets(self):  1 usage
    """Create login interface elements."""
    for widget in self.master.winfo_children():
        widget.destroy()  # Clear existing widgets

    # Create and pack title label
    title_label = tk.Label(self.master, text="Welcome to Finance Manager", font=("Helvetica", 20, "bold"), bg="#f0f8ff", fg="#4682b4")
    title_label.pack(pady=20)

    # Create and pack user ID label
    user_id_label = tk.Label(self.master, text="Enter USER ID:", font=("Helvetica", 14), bg="#f0f8ff", fg="blue")
    user_id_label.pack(pady=5)

    # Create user ID entry field
    self.id_entry = tk.Entry(self.master, width=30)
    self.id_entry.pack(pady=10)

    # Create and pack password label
    password_label = tk.Label(self.master, text="Password:", font=("Helvetica", 14), bg="#f0f8ff", fg="green")
    password_label.pack(pady=5)

    # Create password entry field
    self.password_entry = tk.Entry(self.master, width=30, show='*')
    self.password_entry.pack(pady=10)

    # Create login button
    login_button = tk.Button(self.master, text="Login", command=self.login, bg="#add8e6", borderwidth=0)
    login_button.pack(pady=10)
```

```python
    # Create register button
    register_button = tk.Button(self.master, text="Register", command=self.register, bg="#add8e6", borderwidth=0)
    register_button.pack(pady=10)

    footer_label = tk.Label(self.master, text="Developed by SUPREETH and TARUN", font=("Helvetica", 10),
                            bg="#f0f8ff", fg="#696969")
    footer_label.pack(side=tk.BOTTOM, pady=10)
```

```python
def login(self):  1 usage
    """Handle user login."""
    user_id = self.id_entry.get()  # Get user ID from entry
    password = self.password_entry.get()  # Get password from entry

    # Check if the user ID exists and the password matches
    if user_id in self.users and self.users[user_id] == password:
        self.current_user = user_id  # Set the current user

        # Check if the user has profile details
        if user_id not in self.profile_details:  # If no profile details exist
            self.open_profile_window()  # Open profile window for first-time users
        else:
            self.create_widgets()  # Create the main application interface for existing users
    else:
        messagebox.showerror( title: "Login Failed",  message: "Invalid User ID or Password.")  # Show error message
```

```python
def open_profile_window(self):  1 usage
    """Open a new window for the user to enter their profile details."""
    profile_window = tk.Toplevel(self.master)  # Create a new window
    profile_window.title("PROFILE")  # Set title
    profile_window.geometry("400x400")  # Set size
    profile_window.configure(bg="#e6f7ff")  # Set background color

    # Create a title label
    title_label = tk.Label(profile_window, text="Enter Your Profile Details", font=("Helvetica", 16, "bold"), bg="#e6f7ff", fg="#4682b4")
    title_label.pack(pady=10)

    # Create labels and entry fields for user details with colorful backgrounds
    fields = [("Name:", "Name"), ("Mobile No:", "Mobile"), ("Age:", "Age"), ("Gender:", "Gender"),
              ("Bank Account Number:", "Bank"), ("Email:", "Email")]

    entries = {}
    for label_text, entry_key in fields:
        frame = tk.Frame(profile_window, bg="#e6f7ff")  # Frame for each entry
        frame.pack(pady=5)

        label = tk.Label(frame, text=label_text, font=("Helvetica", 12), bg="#e6f7ff", fg="#333")
        label.pack(side=tk.LEFT)

        entry = tk.Entry(frame, width=30)
        entry.pack(side=tk.RIGHT)
        entries[entry_key] = entry  # Store entry in dictionary
```

```python
    # Create a button to submit the profile details with a colorful appearance
    submit_button = tk.Button(profile_window, text="Submit", command=lambda: self.save_profile_details(
        entries["Name"].get(), entries["Mobile"].get(), entries["Age"].get(), entries["Gender"].get(),
        entries["Bank"].get(), entries["Email"].get(), profile_window), bg="#98fb98", font=("Helvetica", 12))
    submit_button.pack(pady=20)

    # Create a footer label
    footer_label = tk.Label(profile_window, text="Please fill in your details.", font=("Helvetica", 10), bg="#e6f7ff", fg="#555")
    footer_label.pack(side=tk.BOTTOM, pady=10)
```

```python
def save_profile_details(self, name, mobile, age, gender, bank_account, email, window):  1 usage
    """Save the profile details and open the finance manager."""
    self.profile_details[self.current_user] = {
        "Name": name,
        "Mobile No": mobile,
        "Age": age,
        "Gender": gender,
        "Bank Account Number": bank_account,
        "Email": email
    }
    self.save_data()  # Save data to the file after updating profile details
    window.destroy()  # Close the profile window
    self.create_widgets()  # Open the finance manager interface
```

```python
def show_account_details(self):  1 usage
    """Display the current user's account details in a new window."""
    if self.current_user:
        # Create a new window for displaying account details
        profile_window = tk.Toplevel(self.master)  # Create a new window
        profile_window.title("Profile Details")  # Set title
        profile_window.geometry("400x400")  # Set size
        profile_window.configure(bg="#e6f7ff")  # Set background color

        # Fetch profile details for the current user
        details = self.profile_details.get(self.current_user, {})

        # Create labels to display user details
        tk.Label(profile_window, text="User  ID: " + self.current_user, font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Name: " + details.get("Name", "N/A"), font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Mobile No: " + details.get("Mobile No", "N/A"), font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Age: " + details.get("Age", "N/A"), font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Gender: " + details.get("Gender", "N/A"), font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Bank Account Number: " + details.get("Bank Account Number", "N/A"),
                 font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)
        tk.Label(profile_window, text="Email: " + details.get("Email", "N/A"), font=("Helvetica", 14), bg="#e6f7ff").pack(pady=5)

        # Create a button to close the profile window
        close_button = tk.Button(profile_window, text="Close", command=profile_window.destroy, bg="#ffcccb")
        close_button.pack(pady=20)
    else:
        messagebox.showerror( title: "Error",  message: "No user logged in.")  # Show error if no user is logged in
```

```python
def register(self):  1 usage
    """Handle user registration."""
    user_id = self.id_entry.get()  # Get user ID from entry
    password = self.password_entry.get()  # Get password from entry

    # Check if the user ID already exists
    if user_id in self.users:
        messagebox.showerror( title: "Registration Failed",  message: "User  ID already exists.")  # Show error message
    else:
        self.users[user_id] = password  # Add new user to the users dictionary
        messagebox.showinfo( title: "Registration Successful",  message: "You can now log in.")  # Show success message
```

```python
def create_widgets(self):  2 usages
    """Create the main application interface after login."""
    for widget in self.master.winfo_children():
        widget.destroy()  # Clear existing widgets

    # Create side panel for navigation
    side_panel = tk.Frame(self.master, bg="#4682b4", width=50)
    side_panel.pack(side=tk.LEFT, fill=tk.Y)

    # Create profile button
    profile_button = tk.Button(side_panel, text="Profile", command=self.show_account_details, bg="#4682b4",
                               borderwidth=0, font=("Helvetica", 14, "bold"), width=10, height=2)
    profile_button.pack(pady=20)

    # Create main finance frame
    finance_frame = tk.Frame(self.master, bg="#f0f8ff")
    finance_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

    # Create title label for finance manager
    title_label = tk.Label(finance_frame, text="Finance Manager", font=("Helvetica", 20, "bold"), bg="#f0f8ff",
                           fg="#4682b4")
    title_label.pack(pady=20)

    # Create label to display current balance
    self.balance_label = tk.Label(finance_frame, text=f"Current Balance: ${self.balance:.2f}",
                                  font=("Helvetica", 16), bg="#f0f8ff", fg="#228b22")
    self.balance_label.pack(pady=10)
```

```python
# Create button frame for action buttons
button_frame = tk.Frame(finance_frame, bg="#f0f8ff")
button_frame.pack(pady=20)

# Create rows of buttons for various actions
for i in range(0, 5, 5):  # 10 buttons total, 5 per row
    row_frame = tk.Frame(button_frame, bg="#f0f8ff")
    row_frame.pack(pady=10)

    # Create buttons for adding income, expenses, future bills, and savings goals
    income_button = tk.Button(row_frame, text="Add Income", command=self.open_income_categories, bg="#98fb98",
                              borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    income_button.pack(side=tk.LEFT, padx=10)


    expense_button = tk.Button(row_frame, text="Add Expense", command=self.open_expense_categories,
                               bg="#ffcccb", borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    expense_button.pack(side=tk.LEFT, padx=10)

    bill_button = tk.Button(row_frame, text="Add Future Bill", command=self.add_future_bill, bg="#ffebcd",
                            borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    bill_button.pack(side=tk.LEFT, padx=10)

    # Second row of buttons
    row_frame = tk.Frame(button_frame, bg="#f0f8ff")
    row_frame.pack(pady=10)
```

```python
    # Create buttons for setting expense limit and income analysis
    limit_button = tk.Button(row_frame, text="Set Expense Limit", command=self.set_expense_limit, bg="#f0e68c",
                             borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    limit_button.pack(side=tk.LEFT, padx=10)

    income_analysis_button = tk.Button(row_frame, text="Income Analysis", command=self.show_income_analysis,
                                       bg="#add8e6", borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    income_analysis_button.pack(side=tk.LEFT, padx=10)

    # Create a button for expense analysis
    expense_analysis_button = tk.Button(row_frame, text="Expense Analysis", command=self.open_expense_analysis_options,
                                        bg="#add8e6", borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    expense_analysis_button.pack(side=tk.LEFT, padx=10)

    # Create a frame for Reset and Exit buttons
    bottom_frame = tk.Frame(finance_frame, bg="#f0f8ff")
    bottom_frame.pack(side=tk.BOTTOM, anchor='se', padx=10, pady=10)  # Bottom right corner

    # Create Reset Data button
    reset_button = tk.Button(bottom_frame, text="Reset Data", command=self.reset_data, bg="#ffcc00",
                             borderwidth=0,
                             font=("Rockwell", 14, "bold"), width=10, height=1)
    reset_button.pack(side=tk.LEFT, padx=10)

    # Create buttons for generating reports and exiting the application
    report_button = tk.Button(row_frame, text="Generate Report", command=self.generate_report, bg="#add8e6",
                              borderwidth=0, font=("Rockwell", 14, "bold"), width=15, height=2)
    report_button.pack(side=tk.LEFT, padx=10)
```

```python
    exit_button = tk.Button(bottom_frame, text="❌ Exit", command=self.exit_application, bg="#ffb6c1",
                            fg="#000000",
                            font=("Rockwell", 14, "bold"), width=10, height=1)
    exit_button.pack(side=tk.LEFT, padx=10)

# Create status bar to display messages
self.status_bar = tk.Label(finance_frame, text="Welcome to the Personal Finance Manager!", bg="#f0f8ff",
                           fg="#4682b4", font=("Century Gothic", 18))
self.status_bar.pack(side=tk.BOTTOM, fill=tk.X)

footer_label = tk.Label(finance_frame, text="Developed by SUPREETH and TARUN", font=("Century Gothic", 14),
                        bg="#f0f8ff", fg="#696969")
footer_label.pack(side=tk.BOTTOM, pady=10)
```

```python
def open_expense_analysis_options(self):  1 usage
    """Open a new window to select the type of expense analysis."""
    analysis_window = tk.Toplevel(self.master)  # Create a new window
    analysis_window.title("Select Expense Analysis Type")  # Set title
    analysis_window.geometry("300x200")  # Set size

    # Create buttons for each analysis type
    daily_button = tk.Button(analysis_window, text="Daily Analysis",
                             command=lambda: self.show_expense_analysis("Daily Analysis"), bg="#98fb98")
    daily_button.pack(pady=10)

    monthly_button = tk.Button(analysis_window, text="Monthly Analysis",
                               command=lambda: self.show_expense_analysis("Monthly Analysis"), bg="#ffcccb")
    monthly_button.pack(pady=10)

    annual_button = tk.Button(analysis_window, text="Annual Analysis",
                              command=lambda: self.show_expense_analysis("Yearly Analysis"), bg="#add8e6")
    annual_button.pack(pady=10)

    # Create a close button
    close_button = tk.Button(analysis_window, text="Close", command=analysis_window.destroy, bg="#ffb6c1")
    close_button.pack(pady=10)
```

```python
def open_income_categories(self):  1 usage
    """Open a new window to select income category and amount."""
    income_window = tk.Toplevel(self.master)  # Create a new window
    income_window.title("Select Income Category")  # Set title
    income_window.geometry("500x500")  # Set size

    categories = ["SALARY", "BUSINESS INCOME", "INVESTING INCOME", "PENSION", "FREELANCING", "OTHERS"]  # Income categories

    # Create a label for category selection
    category_label = tk.Label(income_window, text="Select Income Category:", font=("Helvetica", 14))
    category_label.pack(pady=10)

    # Create a variable to store selected category
    selected_category = tk.StringVar(value=categories[0])  # Default to first category

    # Create radio buttons for each category
    for category in categories:
        radio_button = tk.Radiobutton(income_window, text=category, variable=selected_category, value=category)
        radio_button.pack(anchor=tk.W)

    # Create an entry for income amount
    amount_label = tk.Label(income_window, text="Enter Amount:", font=("Helvetica", 12))
    amount_label.pack(pady=10)

    amount_entry = tk.Entry(income_window)
    amount_entry.pack(pady=5)

    # Create a button to confirm income addition
    confirm_button = tk.Button(income_window, text="Add Income", command=lambda: self.add_income(selected_category.get(), amount_entry.get(), income_window)
    confirm_button.pack(pady=20)
```

```python
def open_expense_categories(self):  1 usage
    """Open a new window to select expense category and amount."""
    expense_window = tk.Toplevel(self.master)  # Create a new window
    expense_window.title("Select Expense Category")  # Set title
    expense_window.geometry("500x500")  # Set size

    categories = ["FOOD", "TRANSPORT", "UTILITIES", "ENTERTAINMENT", "HEALTH", "OTHERS"]  # Expense categories

    # Create a label for category selection
    category_label = tk.Label(expense_window, text="Select Expense Category:", font=("Helvetica", 14))
    category_label.pack(pady=10)

    # Create a variable to store selected category
    selected_category = tk.StringVar(value=categories[0])  # Default to first category

    # Create radio buttons for each category
    for category in categories:
        radio_button = tk.Radiobutton(expense_window, text=category, variable=selected_category, value=category)
        radio_button.pack(anchor=tk.W)

    # Create an entry for expense amount
    amount_label = tk.Label(expense_window, text="Enter Amount:", font=("Helvetica", 12))
    amount_label.pack(pady=10)

    amount_entry = tk.Entry(expense_window)
    amount_entry.pack(pady=5)

    # Create an entry for expense date
    date_label = tk.Label(expense_window, text="Enter Date (MM/DD/YYYY):", font=("Helvetica", 12))
    date_label.pack(pady=10)
```

```python
    date_entry = tk.Entry(expense_window)
    date_entry.pack(pady=5)

    # Create a button to confirm expense addition
    confirm_button = tk.Button(expense_window, text="Add Expense", command=lambda: self.add_expense(selected_category.get(), amount_entry.get(),
                                                                                                    date_entry.get(), expense_window), bg="#ffcccb")
    confirm_button.pack(pady=20)
```

```python
def add_income(self, category, amount_str, window):  1 usage
    """Add income to the user's account."""
    try:
        amount = float(amount_str)  # Convert amount to float
        if amount > 0:
            self.balance += amount  # Update balance
            self.transactions.append({"Type": "Income", "Amount": amount, "Category": category})  # Record transaction

            # Update income categories
            if category in self.income_categories:
                self.income_categories[category] += amount
            else:
                self.income_categories[category] = amount

            self.balance_label.config(text=f"Current Balance: ${self.balance:.2f}")  # Update balance label
            self.status_bar.config(text=f"Income of ${amount:.2f} under '{category}' added successfully!")  # Update status bar message
            self.save_data()  # Save data
            window.destroy()  # Close the income window
        else:
            messagebox.showerror( title: "Invalid Amount",  message: "Please enter a positive amount.")  # Show error for invalid amount
    except ValueError:
        messagebox.showerror( title: "Error",  message: "Invalid input for amount. Please enter a number.")  # Show error for invalid input
```

```python
def add_expense(self, category, amount_str, date_str, window):  2 usages
    """Add expense to the user's account."""
    try:
        amount = float(amount_str)  # Convert amount to float
        if amount > 0:
            date = datetime.strptime(date_str, format: "%m/%d/%Y")  # Convert string to date

            # Check if an expense limit is set for the category
            limit = self.expense_limits.get(category, None)
            if limit is not None and (self.get_total_expense_for_category(category) + amount) > limit:
                messagebox.showwarning( title: "Expense Limit Exceeded",
                                        message: f"The expense of ${amount:.2f} exceeds the limit of ${limit:.2f} for '{category}'.")
                return  # Do not proceed with the transaction

            if amount > self.balance:
                messagebox.showerror( title: "Insufficient Balance",
                                      message: "You do not have sufficient balance to make this expense.")  # Show error if insufficient balance
            else:
                self.balance -= amount  # Deduct amount from balance
                self.transactions.append({"Type": "Expense", "Amount": amount, "Category": category,
                                          "Date": date_str})  # Record transaction
                self.balance_label.config(text=f"Current Balance: ${self.balance:.2f}")  # Update balance label
                self.status_bar.config(
                    text=f"Expense of ${amount:.2f} on '{date_str}' under '{category}' added successfully!")  # Update status bar message
                self.save_data()  # Save data
                window.destroy()  # Close the expense window
        else:
            messagebox.showerror( title: "Invalid Amount",
                                  message: "Please enter a positive amount.")  # Show error for invalid amount
    except ValueError:
        messagebox.showerror( title: "Error",
                              message: "Invalid input for amount or date. Please enter valid values.")  # Show error for invalid input
```

```python
def get_total_expense_for_category(self, category):  1 usage
    """Calculate the total expenses for a specific category."""
    total_expense = sum(transaction['Amount'] for transaction in self.transactions
                        if transaction['Type'] == "Expense" and transaction['Category'] == category)
    return total_expense
```

```python
def reset_data(self):  1 usage
    """Reset all user and financial data."""
    confirm = messagebox.askyesno( title: "Confirm Reset",  message: "Are you sure you want to reset all data? This action cannot be undone.")
    if confirm:
        self.users = {}  # Clear user data
        self.current_user = None  # Reset current user
        self.balance = 0.0  # Reset balance
        self.budget = {}  # Clear budget
        self.transactions = []  # Clear transactions
        self.future_bills = []  # Clear future bills
        self.expense_limits = {}  # Clear expense limits
        self.income_categories = {}  # Clear income categories
        self.investments = {}  # Clear investments
        self.assets = {}  # Clear assets
        self.profile_details = {}  # Clear profile details
        self.save_data()  # Save the reset data
        self.balance_label.config(text=f"Current Balance: ${self.balance:.2f}")  # Update balance label
        self.status_bar.config(text="All data has been reset successfully!")  # Update status bar message
```

```python
def exit_application(self):  1 usage
    """Exit the application and save data."""
    self.save_data()  # Save data before exiting
    self.master.quit()  # Close the application
```

```python
def add_future_bill(self):  1 usage
    """Add a future bill to the user's account."""
    future_bill_window = tk.Toplevel(self.master)  # Create a new window
    future_bill_window.title("Select Bill Category")  # Set title
    future_bill_window.geometry("500x500")  # Set size

    categories = ["UTILITY", "GROCERY", "MEDICAL", "INSURANCE", "RENT", "CREDIT CARD", "TAX", "SUBSCRIPTION",
                  "OTHERS"]
    # Expense categories

    category_label = tk.Label(future_bill_window, text="Select Bill Category:", font=("Helvetica", 14))
    category_label.pack(pady=10)

    selected_category = tk.StringVar(value=categories[0])  # Default to first category

    for category in categories:
        radio_button = tk.Radiobutton(future_bill_window, text=category, variable=selected_category, value=category)
        radio_button.pack(anchor=tk.W)

    amount_label = tk.Label(future_bill_window, text="Enter Bill Amount:", font=("Helvetica", 12))
    amount_label.pack(pady=10)


    amount_entry = tk.Entry(future_bill_window)
    amount_entry.pack(pady=5)

    date_label = tk.Label(future_bill_window, text="Enter Due Date (MM/DD/YYYY):", font=("Helvetica", 12))
    date_label.pack(pady=10)

    date_entry = tk.Entry(future_bill_window)
    date_entry.pack(pady=5)

    # Create a button to confirm future bill addition
    confirm_button = tk.Button(future_bill_window, text="Add Future Bill",
                               command=lambda: self.add_expense(selected_category.get(), amount_entry.get(),
                                                                date_entry.get(), future_bill_window), bg="#ffcccb")
    confirm_button.pack(pady=20)
```

```python
def set_expense_limit(self):  1 usage
    """Set an expense limit for a specific category."""
    limit_window = tk.Toplevel(self.master)  # Create a new window for setting limits
    limit_window.title("Set Expense Limit")  # Set title
    limit_window.geometry("500x500")  # Set size

    categories = ["FOOD", "TRANSPORT", "UTILITIES", "ENTERTAINMENT", "HEALTH", "OTHERS"]  # Expense categories

    # Create a label for category selection
    category_label = tk.Label(limit_window, text="Select Expense Category:", font=("Helvetica", 14))
    category_label.pack(pady=10)

    # Create a variable to store selected category
    selected_category = tk.StringVar(value=categories[0])  # Default to first category

    # Create radio buttons for each category
    for category in categories:
        radio_button = tk.Radiobutton(limit_window, text=category, variable=selected_category, value=category)
        radio_button.pack(anchor=tk.W)

    # Create an entry for limit amount
    amount_label = tk.Label(limit_window, text="Enter Limit Amount:", font=("Helvetica", 12))
    amount_label.pack(pady=10)

    limit_entry = tk.Entry(limit_window)
    limit_entry.pack(pady=5)
```

```python
    # Create a button to confirm limit setting
    confirm_button = tk.Button(limit_window, text="Set Limit",
                               command=lambda: self.confirm_limit(selected_category.get(), limit_entry.get(),
                                                                  limit_window), bg="#f0e68c")
    confirm_button.pack(pady=20)
```

```python
def confirm_limit(self, category, limit_str, window):  1 usage
    """Confirm and set the limit for the selected category."""
    try:
        limit = float(limit_str)  # Convert limit to float
        if limit >= 0:  # Ensure limit is non-negative
            self.expense_limits[category] = limit  # Set the expense limit
            self.status_bar.config(
                text=f"Expense limit of ${limit:.2f} set for '{category}'.")  # Update status bar message
            self.save_data()  # Save data
            window.destroy()  # Close the limit setting window
        else:
            messagebox.showerror( title: "Invalid Amount",
                                  message: "Please enter a non-negative limit amount.")  # Show error for invalid limit
    except ValueError:
        messagebox.showerror( title: "Error",
                              message: "Invalid input for limit. Please enter a number.")  # Show error for invalid input
```

```python
def show_income_analysis(self):  1 usage
    """Display a bar graph of income by category."""
    if not self.income_categories:
        messagebox.showinfo( title: "Income Analysis",  message: "No income recorded.")  # Show message if no income
        return

    categories = list(self.income_categories.keys())  # Get income categories
    amounts = list(self.income_categories.values())  # Get income amounts

    plt.figure(figsize=(8, 6))  # Create a new figure for the bar graph
    plt.bar(categories, amounts, color='skyblue')  # Create bar graph
    plt.title("Income Analysis by Category")  # Set title
    plt.xlabel("Categories")  # Set x-axis label
    plt.ylabel("Income ($)")  # Set y-axis label
    plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility
    plt.tight_layout()  # Adjust layout to fit elements
    plt.show()  # Display the graph
```

```python
def show_expense_analysis(self, analysis_type):  3 usages
    """Display a colorful line chart of expenses over time based on the analysis type."""
    if not self.transactions:
        messagebox.showinfo( title: "Expense Analysis",  message: "No expenses recorded.")  # Show message if no expenses
        return

    # Initialize a dictionary to summarize expenses by date
    expense_summary = {}

    # Categorize expenses
    for transaction in self.transactions:
        if 'Type' in transaction and transaction['Type'] == "Expense":
            date_str = transaction.get('Date', '')
            if not date_str:
                continue

            try:
                date = datetime.strptime(date_str,  format: "%m/%d/%Y")  # Convert string to date
            except ValueError:
                messagebox.showerror( title: "Date Error",  message: f"Date '{date_str}' is not in the correct format (MM/DD/YYYY).")
                return

            # Determine the key for summarizing expenses based on analysis type
            if analysis_type == "Daily Analysis":
                key = date.date()
            elif analysis_type == "Monthly Analysis":
                key = date.replace(day=1)
            elif analysis_type == "Yearly Analysis":
                key = date.replace(month=1, day=1)
            else:
                continue
```

```python
            amount = transaction['Amount']
            if key in expense_summary:
                expense_summary[key] += amount  # Sum expenses for the key
            else:
                expense_summary[key] = amount  # Initialize the key

    if not expense_summary:
        messagebox.showinfo( title: "Expense Analysis",
                             message: "No expenses recorded for this period.")  # Show message if no expenses for the period
        return

    sorted_dates = sorted(expense_summary.keys())  # Sort dates
    sorted_amounts = [expense_summary[date] for date in sorted_dates]  # Get sorted amounts

    plt.figure(figsize=(12, 6))  # Create a new figure for the line chart

    # Generate a color map for the expenses
    colors = plt.cm.viridis(np.linspace( start: 0,  stop: 1, len(sorted_dates)))

    # Plot expenses over time with fill under the line
    plt.plot( *args: sorted_dates, sorted_amounts, marker='o', linestyle='-', color='b', linewidth=2, markersize=8)  # Line
    plt.fill_between(sorted_dates, sorted_amounts, color='skyblue', alpha=0.3)  # Fill under the line

    # Format the x-axis to avoid overlapping dates
    plt.gca().xaxis.set_major_locator(mdates.AutoDateLocator())
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
```

```python
    # Add annotations for significant points
    for i, amount in enumerate(sorted_amounts):
        plt.annotate( text: f'${amount:.2f}',  xy: (sorted_dates[i], sorted_amounts[i]), textcoords="offset points",
                     xytext=(0, 10), ha='center', fontsize=9, color='darkblue')

    plt.title( label: f"Expense Analysis - {analysis_type}", fontsize=16)  # Set title
    plt.xlabel( xlabel: "Date", fontsize=12)  # Set x-axis label
    plt.ylabel( ylabel: "Amount ($)", fontsize=12)  # Set y-axis label
    plt.grid( visible: True, linestyle='--', alpha=0.7)  # Show grid with dashed lines
    plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility
    plt.tight_layout()  # Adjust layout to fit elements
    plt.show()  # Display the chart
```

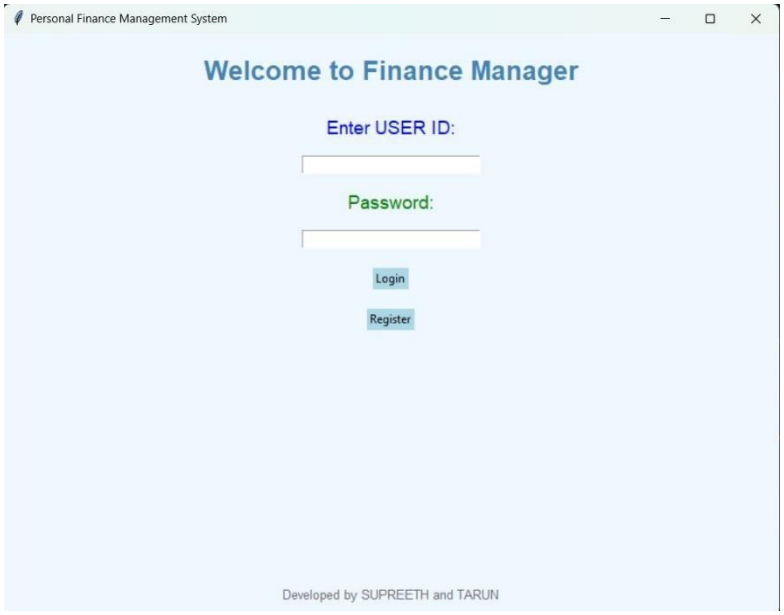
```python
def generate_report(self):  1 usage
    """Generate a financial report summarizing the user's financial status."""
    total_income = sum(amount for amount in self.income_categories.values())  # Calculate total income
    total_expenses = sum(transaction['Amount'] for transaction in self.transactions if
                         'Type' in transaction and transaction['Type'] == "Expense")  # Calculate total expenses

    total_assets = sum(self.assets.values())  # Calculate total assets
    total_investments = sum(self.investments.values())  # Calculate total investments

    net_worth = total_assets + total_investments + total_income - total_expenses  # Calculate net worth

    # Create a new window for the report
    report_window = tk.Toplevel(self.master)
    report_window.title("Financial Report")
    report_window.geometry("600x600")  # Set the size
    report_window.configure(bg="#f0f8ff")  # Set a light background color

    # Create a title label
    title_label = tk.Label(report_window, text="Financial Report", font=("Corbel", 24, "bold"), bg="#f0f8ff",
                           fg="#4682b4")
    title_label.pack(pady=20)

    # Create a text widget to display the report
    report_text = tk.Text(report_window, wrap=tk.WORD, bg="#ffffff", fg="#333333", font=("Corbel", 14))
    report_text.pack(expand=True, fill=tk.BOTH, padx=20, pady=10)
```

```python
        # Create a formatted report string
        report = ["==============================",
                  "         Financial Report     ",
                  "==============================",
                  f"Current Balance:  ${self.balance:.2f}",
                  f"Total Income:     ${total_income:.2f}",
                  "------------------------------",
                  "Income Details:"]

        for transaction in self.transactions:
            if transaction['Type'] == "Income":
                report.append(f"Category: {transaction['Category']}, Amount: ${transaction['Amount']:.2f}")

        report.append("------------------------------")
        report.append("Expense Details:")

        for transaction in self.transactions:
            if transaction['Type'] == "Expense":
                report.append(
                    f"Category: {transaction['Category']}, Amount: ${transaction['Amount']:.2f}, Date: {transaction['Date']}")

        report.append("------------------------------")
        report.append("Future Bills:")

        for bill in self.future_bills:
            report.append(f"Category: {bill['Category']}, Amount: ${bill['Amount']:.2f}, Due Date: {bill['Due Date']}")

        report.append("------------------------------")
        report.append("Expense Limits:")

        for category, limit in self.expense_limits.items():
            report.append(f"Category: {category}, Limit: ${limit:.2f}")

        report.append("==============================")
        report.append(f"Net Worth:       ${net_worth:.2f}")
        report.append("==============================")

        # Insert report text into the Text widget
        report_text.insert(tk.END, "\n".join(report))

        # Create a scrollbar for the text widget
        scrollbar = tk.Scrollbar(report_window, command=report_text.yview)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
        report_text.config(yscrollcommand=scrollbar.set)

        # Disable editing of the text widget
        report_text.config(state=tk.DISABLED)

        # Add some padding to the report text for a cleaner look
        report_text.config(padx=10, pady=10)

        # Add a footer label with a modern touch
        footer_label = tk.Label(report_window, text="Stay Financially Fit!", font=("Corbel", 12, "italic"),
                                bg="#f0f8ff", fg="#4682b4")
        footer_label.pack(side=tk.BOTTOM, pady=10)
```

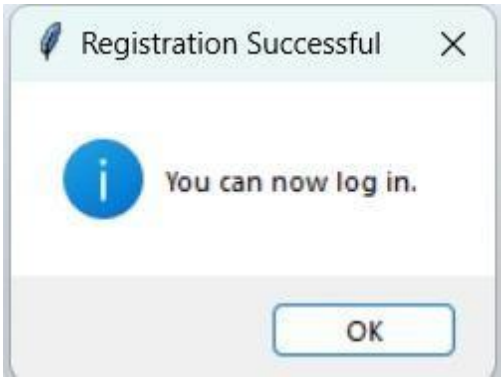
```python
if __name__ == "__main__":
    root = tk.Tk()  # Create the main application window
    app = PersonalFinanceManager(root)  # Create an instance of the PersonalFinanceManager
    root.mainloop()  # Start the application
```

OUTPUT:

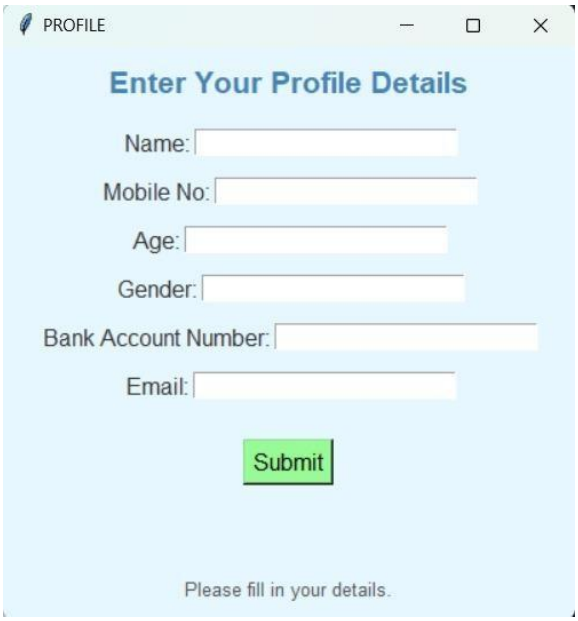
1)Login/Register Window



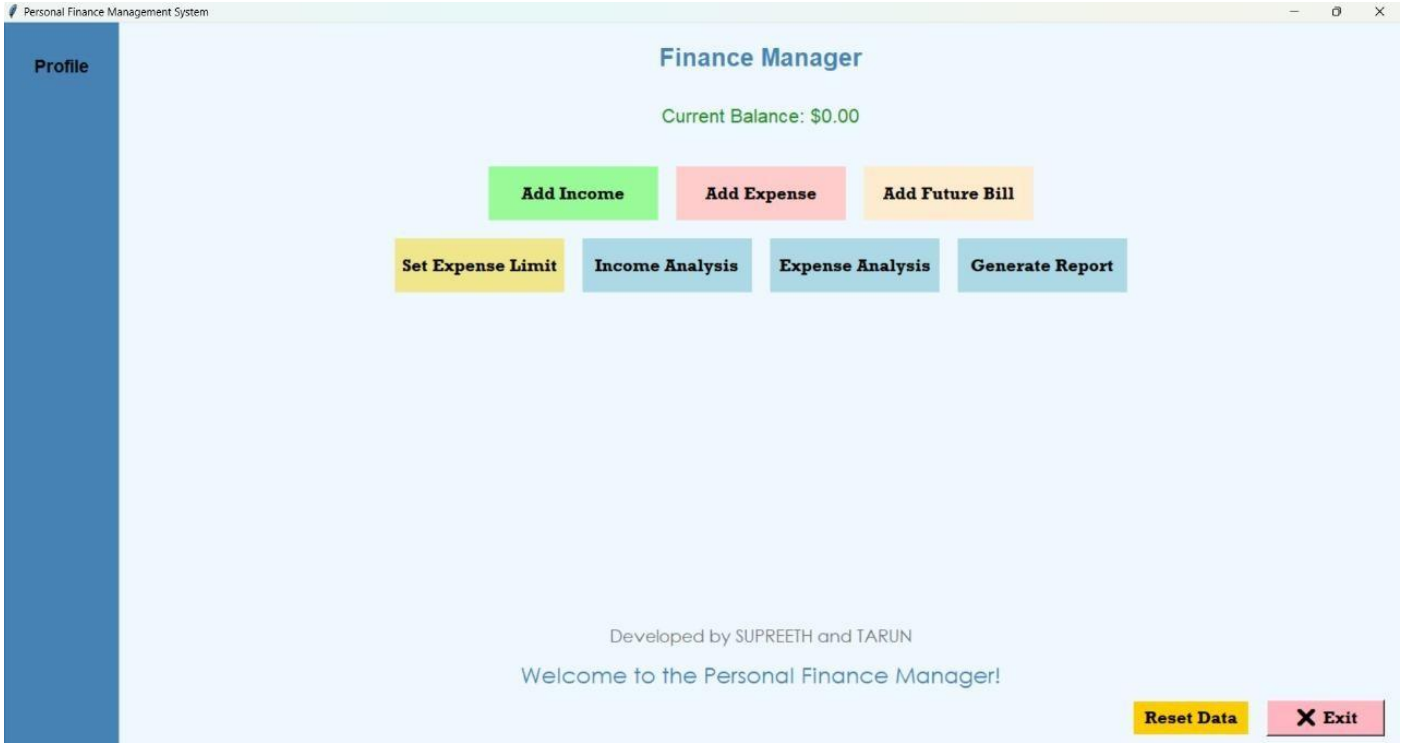
2)Registration successful



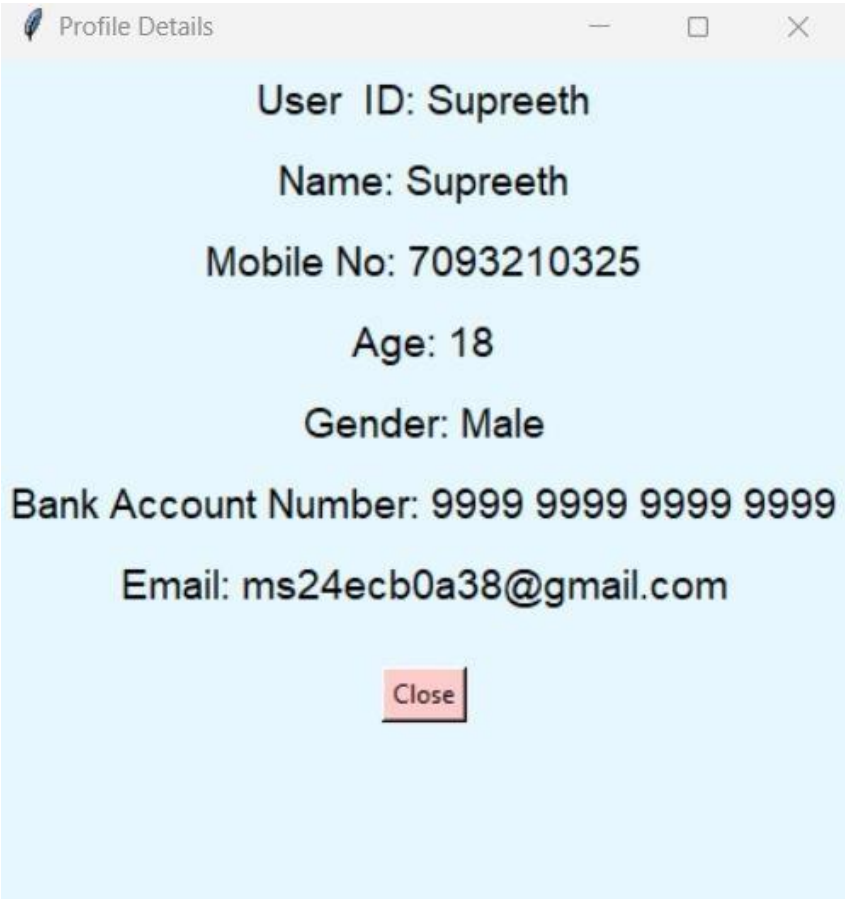
3) Entering Profile details



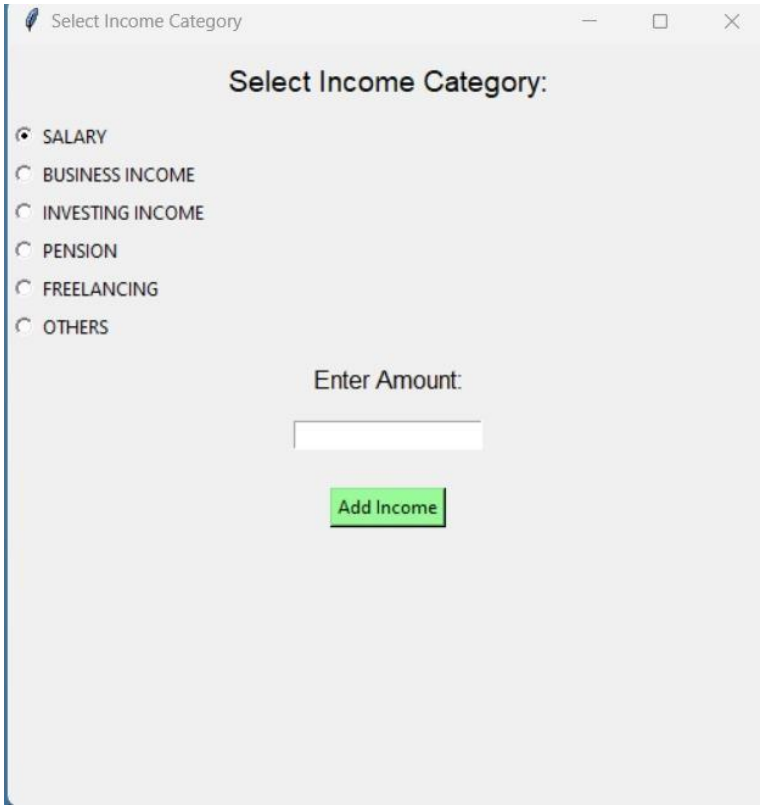
4) Main Interface



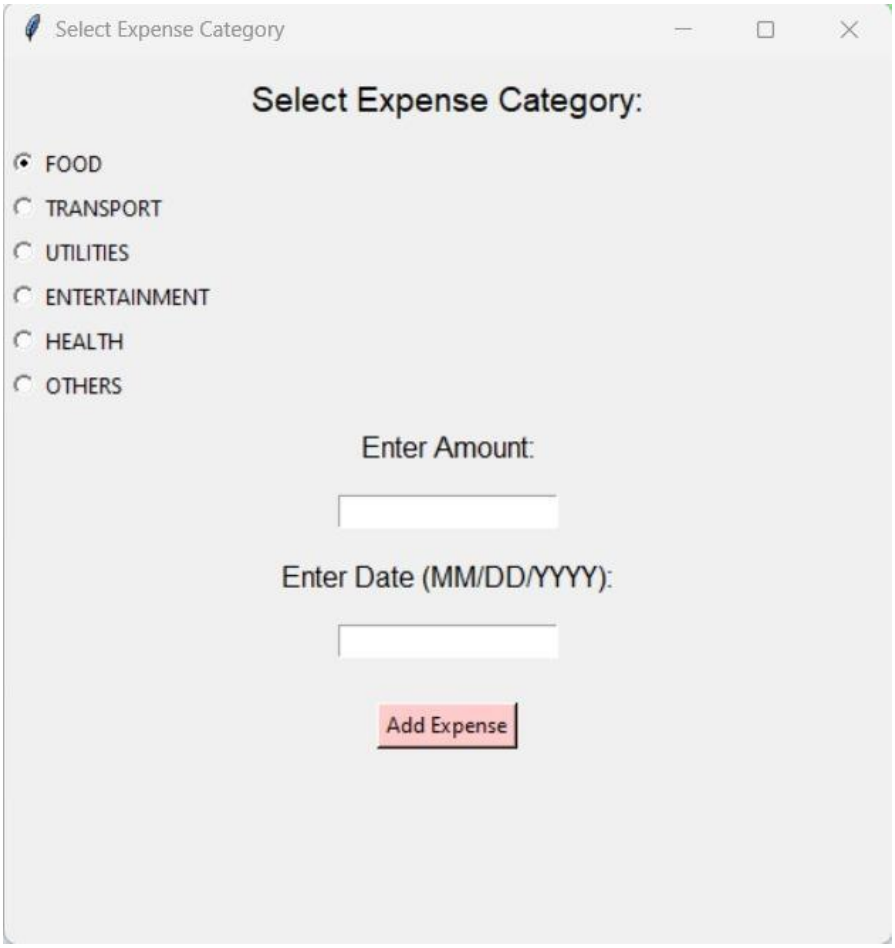
5) Profile details



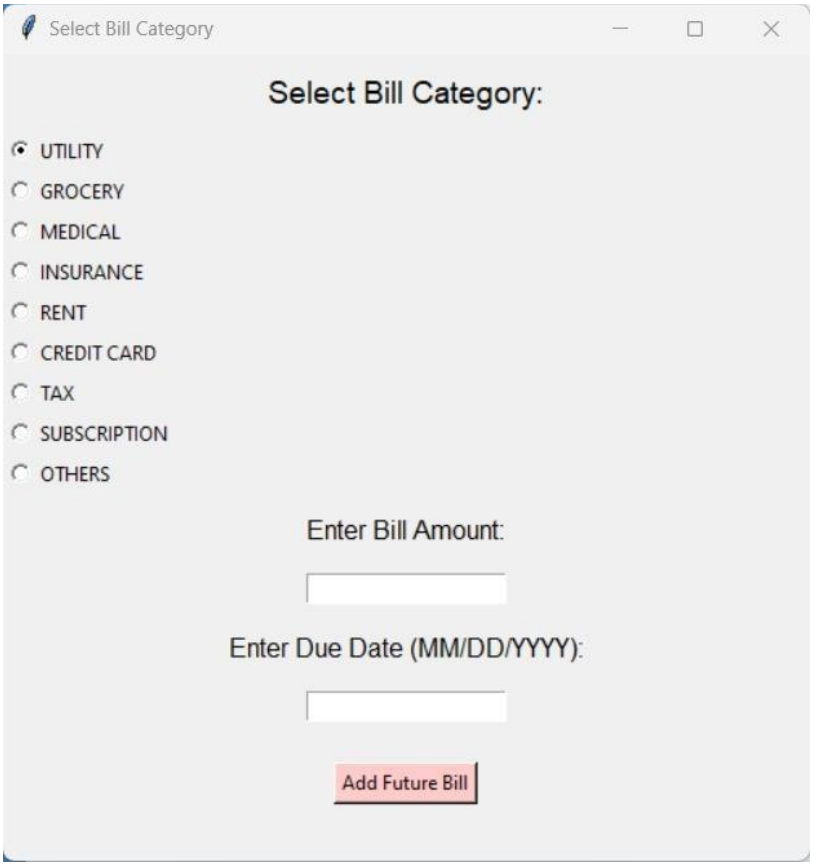
6) Entering Income



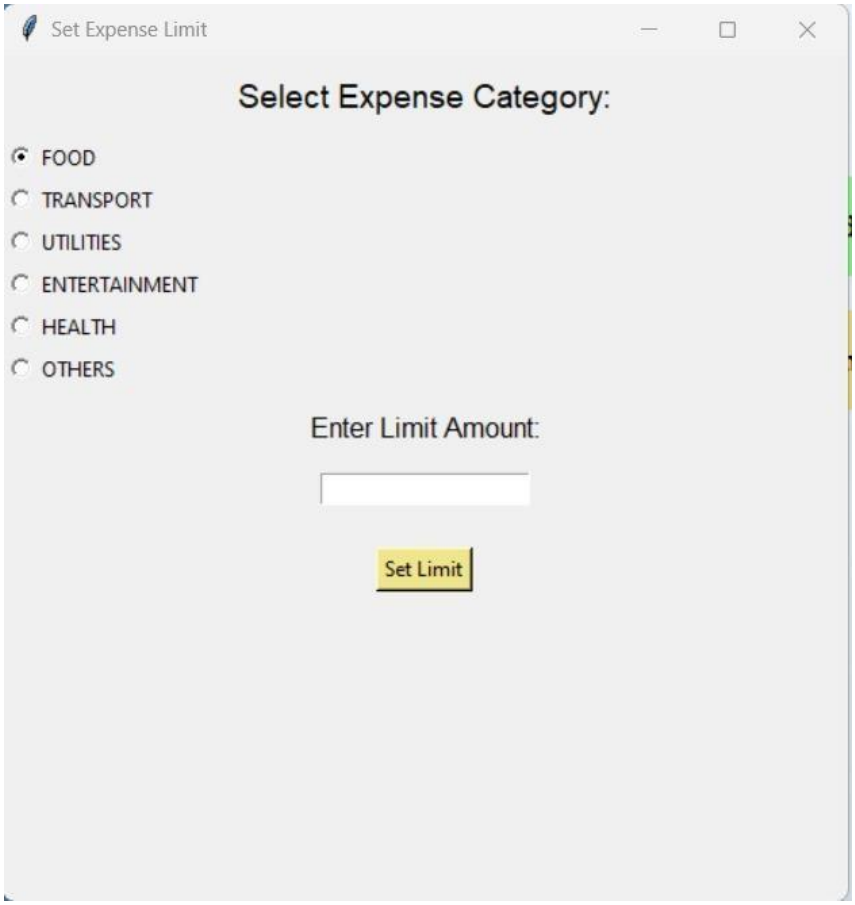
7) Entering Expense



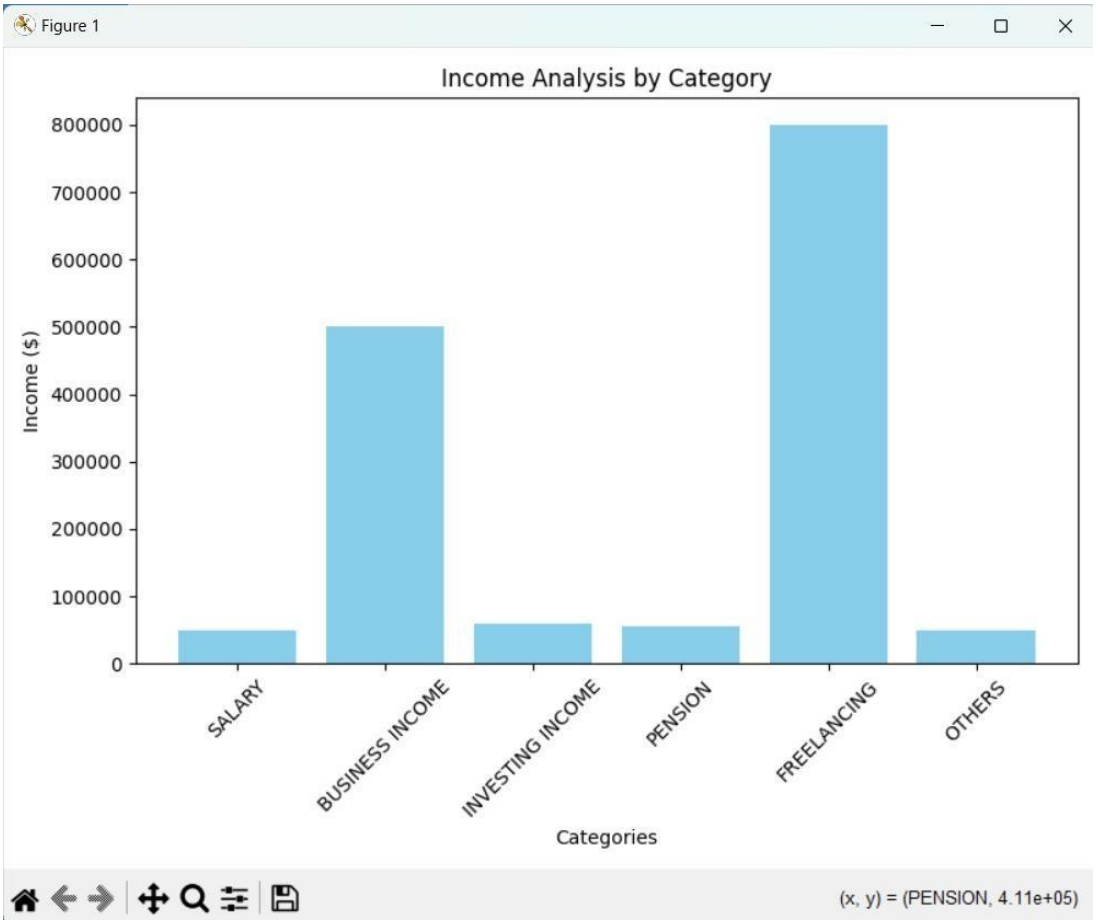
8) Entering Future Bill



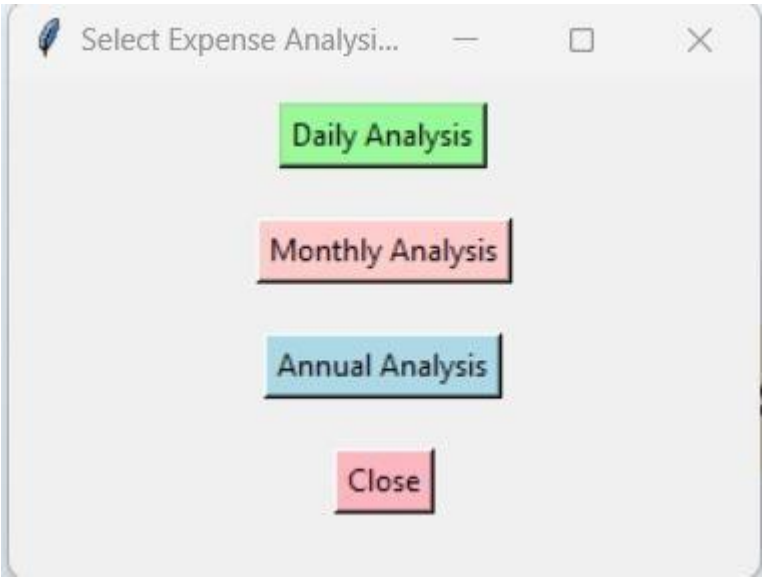
9) Setting Expense Limit



10) Income Analysis



11) Selectin Type Of Expense Analysis



11.1) Daily Expense Analysis

11.2) Monthly Expense Analysis

11.3) Yearly Expense Analysis

12) Financial Report

Conclusion:

The Personal Finance Management System successfully demonstrates how technology can facilitate better financial management for individuals. By integrating user-friendly features with essential financial concepts, the application provides a holistic approach to personal finance. Users can effectively track their financial activities, set goals, and analyze their financial health, leading to more informed financial decisions. The application not only serves as a practical tool for managing finances but also educates users on the importance of budgeting, tracking expenses, and planning for the future. Overall, the system promotes financial literacy and encourages users to take proactive steps toward achieving their financial goals.

THANK YOU

Python project made by:

- 24ECB0A38 Mudhigepalli Supreeth
- 24ECB0A59 Tarun Parashuramappanavara

NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL