KharchaBook

Using Python 

This code is a Python program designed to help you track your income and expenses, store them in a CSV file, and visualize them using a line plot. Below, I'll explain each part of the code step by step, focusing on the logic, syntax, and purpose.

1. **Importing Libraries**

|  |
| --- |
| *Python* |
| import pandas as pd  import csv  from datetime import datetime  from Data\_Entry import get\_Amount, get\_Category, get\_Date, get\_Description  import matplotlib.pyplot as plt |

**import pandas as pd**: Imports the pandas library, which is used for data manipulation and analysis.

**import csv**: Imports the csv module to handle CSV (Comma-Separated Values) files, which are used to store tabular data.

**from datetime import datetime**: Imports the datetime module to work with dates and times.

**from Data\_Entry import ...**: Imports functions get\_Amount, get\_Category, get\_Date, and get\_Description from another file/module named Data\_Entry. These functions prompt the user for input related to a transaction.

**import matplotlib.pyplot as plt**: Imports matplotlib, a library for creating visualizations, specifically its pyplot module, which provides functions to create plots, graphs and etc.

1. **Defining the CSV Class**

The CSV class is responsible for managing the transactions to be stored in the CSV file.

*Class Attributes:*

|  |
| --- |
| *Python* |
| class CSV:  CSV\_FILE = "KharchaBook.csv"  COLUMNS = ["Date", "Amount", "Category", "Description"]  FORMAT = "%d-%m-%Y" |

**CSV\_FILE**: Specifies the name of the CSV file where transactions will be stored.

**COLUMNS**: A list of column names for the CSV file.

**FORMAT**: The format for representing dates (e.g., 01-01-2024).

*Class Methods*:

What is Class Method?

A class method is a method that is bound to a class rather than its object instance. It receives the class itself as the first argument, conventionally named “ cls “.

Why Use a Class?

While not mandatory, using a class often improves code organization, reusability, and maintainability. It provides a structured way to encapsulate CSV-related logic and data.

Can You Use a Different Name?

Absolutely! The name CSV is just a convention. You can use any valid Python identifier for your class or function names. For instance, you could use *Transaction, DataHandler*, or any other descriptive name that suits your project.

* 1. **initialize\_csv**: This method checks if the CSV file exists. If it doesn't, it creates an empty file with the appropriate columns.

|  |
| --- |
| *Python* |
| @classmethod  def initialize\_csv(cls):  try:  pd.read\_csv(cls.CSV\_FILE)  except FileNotFoundError:  df = pd.DataFrame(columns=cls.COLUMNS)  df.to\_csv(cls.CSV\_FILE, index=False) |

**try/except**: If reading the file fails (because it doesn't exist), the code creates a new CSV file with the specified columns.

**pd.DataFrame(columns=cls.COLUMNS)**: Creates an empty DataFrame with the given columns.

**to\_csv**: Saves the DataFrame as a CSV file.

* 1. **add\_entry**: Adds a new transaction (or entry) to the CSV file.

|  |
| --- |
| *Python* |
| @classmethod  def add\_entry(cls, Date, Amount, Category, Description):  new\_entry = {  "Date": Date,  "Amount": Amount,  "Category": Category,  "Description": Description,  }  with open(cls.CSV\_FILE, "a", newline="") as csvfile:  writer = csv.DictWriter(csvfile, fieldnames=cls.COLUMNS)  writer.writerow(new\_entry)  print("Transaction added successfully") |

**new\_entry**: A dictionary containing the transaction details (Date, Amount, Category, Description).

**csv.DictWriter**: Writes a dictionary to the CSV file.

**with open(...)**: Opens the CSV file in append mode ("a") so that new entries are added at the end.

* *This method takes transaction details (Date, Amount, Category, Description) as arguments.*
* *It creates a dictionary (new\_entry) with these details.*
* *It opens the CSV file in append mode ("a") to add new entries without overwriting existing data.*
* *It uses the csv.DictWriter class to write the dictionary (new\_entry) as a new row in the CSV file.*
* *It prints a confirmation message "Transaction added successfully".*
  1. **get\_transactions**: Retrieves transactions within a specified date range and calculates income, expenses, and net savings.

|  |
| --- |
| *Python* |
| @classmethod  def get\_transactions(cls, start\_date, end\_date):  df = pd.read\_csv(cls.CSV\_FILE)  df["Date"] = pd.to\_datetime(df["Date"], format=CSV.FORMAT)  start\_date = datetime.strptime(start\_date, CSV.FORMAT)  end\_date = datetime.strptime(end\_date, CSV.FORMAT) |

**pd.read\_csv**: Reads the CSV file into a DataFrame.

**pd.to\_datetime**: Converts the "Date" column from strings to actual datetime objects, making it easier to filter by date.

* 1. **mask**: A filter that selects rows within the date range specified.

|  |
| --- |
| *Python* |
| *mask = (df["Date"] >= start\_date) & (df["Date"] <= end\_date)*  *filtered\_df = df.loc[mask]* |

**filtered\_df**: A DataFrame containing only the transactions within the specified date range.

*The mask Expression*

**df["Date"]**: This selects the "Date" column from the DataFrame df. The result is a pandas Series containing all the dates in the DataFrame.

**df["Date"] >= start\_date**: This is a comparison operation. It checks each date in the "Date" column *to see if it is greater than or equal to start\_date*.

The result is a Boolean Series where each value is True if the condition is met and False otherwise.

**df["Date"] <= end\_date**: Similarly, this checks each date in the "Date" column *to see if it is less than or equal to end\_date.*

The result is another Boolean Series.

**& (Logical AND):** The ‘&’ operator combines the two Boolean Series using a logical AND operation.

The result is a single Boolean Series (mask) where each value is True only if both conditions are True (i.e., the date is within the specified range).

Example of mask

Suppose df["Date"] contains these values:

Date

01-01-2024

15-01-2024

10-02-2024

25-02-2024

And you set:

start\_date = datetime(2024, 1, 1)

end\_date = datetime(2024, 2, 15)

Then mask would be:

**Date Mask**

01-01-2024 True

15-01-2024 True

10-02-2024 True

25-02-2024 False

**2. Filtering the DataFrame with .loc**

***filtered\_df = df.loc[mask]***

***df.loc[mask]:***

**.loc[ ]** is a pandas method used for accessing a group of rows and columns by labels or a Boolean array.

In this case, mask is a Boolean array that tells .loc[ ] which rows to keep:

*If mask is True for a row, that row is included in the filtered\_df.*

*If mask is False for a row, that row is excluded.*

Example of filtered\_df;

Given the mask from the example above, filtered\_df would contain:

Date

01-01-2024

15-01-2024

10-02-2024

The row with *25-02-2024 is excluded* because it is outside the date range.

The method then prints the transactions and calculates totals:

|  |
| --- |
| *Python* |
| total\_income = filtered\_df[filtered\_df["Category"] == "Income"]["Amount"].sum()  total\_expense = filtered\_df[filtered\_df["Category"] == "Expense"]["Amount"].sum()  print(f"Total Income: ${total\_income:.2f}")  print(f"Total Expense: ${total\_expense:.2f}")  print(f"Net Savings: ${(total\_income - total\_expense):.2f}") |

1. **Defining Helper Functions**

**add Function**: Prompts the user to input a new transaction and adds it to the CSV file.

|  |
| --- |
| *Python* |
| def add():  CSV.initialize\_csv()  Date = get\_Date(...)  Amount = get\_Amount()  Category = get\_Category()  Description = get\_Description()  CSV.add\_entry(Date, Amount, Category, Description) |

**plot\_transactions Function**: Plots income and expenses over time using matplotlib.

|  |
| --- |
| *Python* |
| def plot\_transactions(df):  df.set\_index("Date", inplace=True)  ...  plt.plot(...)  plt.show() |

1. **The FinTrack Function**

This function is the main menu loop that allows the user to interact with the program:

|  |
| --- |
| *Python* |
| def FinTrack():  while True:  ...  if choice == "1":  add()  elif choice == "2":  start\_date = get\_Date("Enter the start date (dd-mm-yyyy): ")  end\_date = get\_Date("Enter the end date (dd-mm-yyyy): ")  df = CSV.get\_transactions(start\_date, end\_date)  if input("Do you want to see a plot? (y/n) ").lower() == "y":  plot\_transactions(df)  elif choice == "3":  print("Exiting...")  break  else:  print("Invalid choice. Enter 1, 2 or 3.") |

1. **Running the Program**

|  |
| --- |
| *Python* |
| if \_\_name\_\_ == "\_\_FinTrack\_\_":  FinTrack() |

This line ensures that the program only runs when the script is executed directly, not when it's imported as a module.

Summary

This code helps manage and analyze personal finances by recording transactions in a CSV file and providing a summary of income, expenses, and savings. It also offers a visual plot of income and expenses over time. The code uses Python libraries like pandas for data handling, csv for file operations, and matplotlib for plotting data.