

By :

INDEX

1. Project Overview and Introduction…….(3 to 4)

2. CODE EXPLANATION……………………….(4 to 24)

2.1 Import Statements……………………(4 to 5)

2.2 **Global Data Structures & Initial Load………(5 to 6)**

**2.3** Utility Functions……………..(7 to 24)

1. clear\_screen()……………………………………(7)

2. is\_valid\_date(date\_str)……………….(7 to 8)

3. (save\_list() & write\_expenses\_to\_file())……..(8 to 10)

4. write\_expenses\_to\_file……………(10 to 11)

5. auto\_category(description)…..…(11 to 12)

6. **add\_expense(expenses), is\_valid\_date())…………….(13 to 15)**

**7.** summary\_by\_category(expenses)**…………………(15 to 17)**

**8.** option()…………………………….………(17 to 19)

**9.** search\_expenses()…………………...(19 to 20)

**10.** currency\_converter()…………………….….(21)

**11. main\_menu()…………………………………………..(**21 to 23)

12. Program Execution Flow.………(23 to 24)

3. Output……….………………………………..(24 to 27)

1. Project Overview and Introduction ✨

**🧠💰 Automated Expense Categorizer Project Overview 🚀📈**

**In today's digital financial landscape, managing personal and organizational finances has become easier in execution yet more complex in analysis. This project addresses the challenge of automatically classifying expenses into logical categories from unstructured textual descriptions (e.g., "Coffee at cafe," "Online book purchase"). By automating this process, it aims to provide clear spending insights, reduce manual effort, and enhance financial record-keeping efficiency.**

**Key Challenges:**

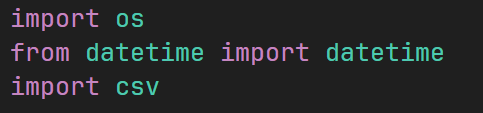
* **Lack of Granular Insight: Raw transaction lists without categories limit strategic financial planning. 🧐**
* **Inconsistent Data Quality: Manual categorization leads to unreliable datasets due to subjective labeling. 📉**
* **Time Intensive: Manual review and categorization are tedious for high transaction volumes. ⏱️**
* **Reduced Analytical Capabilities: Lack of structured data hinders advanced financial analysis. 📊**
* **Human Error: Manual processes increase the risk of mislabeling and errors. 🚨**
* **Disincentive for Tracking: Complexity discourages consistent expense tracking. 🚫**

**Objectives:**

1. **Automated Classification: Develop an algorithm for accurate and consistent expense categorization. ✅**
2. **Comprehensive Schema: Establish a detailed category knowledge base for high accuracy. 📚**
3. **User Flexibility: Allow manual category overrides to align with user preferences. 🤝**
4. **Data Integrity: Ensure reliable storage and retrieval of categorized data. 💾**
5. **Insightful Visualization: Generate clear expenditure summaries for quick insights. 📊**
6. **Scalability: Design for future expansion and integration with advanced tools. 📈**
7. **User-Friendly Interface: Maintain an intuitive command line interface (CLI). 🧑‍💻**

2. 🧩CODE EXPLANATION:

2.1 Import Statements 📥



**import os**:

**🔧 Purpose:** The os module provides a way of using operating system dependent functionality. In this script, it's specifically used for clearing the terminal screen.

**from datetime import datetime**:

**📅Purpose:** The datetime module supplies classes for working with dates and times. Specifically, we import the datetime *class* from this module.

**import csv**:

**📄Purpose:** The csv module implements classes to read and write tabular data in CSV (Comma Separated Values) format.

**2.2 Global Data Structures & Initial Load 🌐**

**🧠 CATEGORIES Dictionary (Global Constant) 📚**



**🔍 Purpose:  
This dictionary serves as the *core knowledge base* for automatic expense categorization. It maps predefined categories to relevant keywords. When a user enters an expense description, the system scans for these keywords to intelligently assign a category.**

**Structure:** It's a Python dictionary.

* **Keys:** These are string labels representing the main expense categories (e.g., "food", "transport", "subscriptions"). These are the final classifications applied to expenses.
* **Values:** Each value is a list of strings. These strings are the keywords or phrases that, if found within an expense's description, suggest that the expense belongs to that particular category.

**Content & Breadth:** The lists of keywords are intentionally extensive and comprehensive. For instance, the "food" category includes specific dishes, types of establishments (restaurant, bakery), common food-related actions (takeout, delivery), and even common items (coffee, groceries). This broad coverage significantly enhances the accuracy of automatic categorization across varied user descriptions. 🥕🍔

**The "Other" Category:**

Acts as a **safety net**. If no match is found in any category, the system defaults the expense to "other", ensuring every transaction is classified and no data is left uncategorized. 🤷‍♀️

**🛠️ Extensibility:** Easily updatable and expandable. New categories or keywords can be added anytime, allowing the system to **evolve without rewriting core logic**. 🌱✨

**expenses List (Global Data Store) 📋**



**🎯Purpose:** This list acts as the **main in-memory storage** for all expense records while the application is running. Each item is a **dictionary** representing a single, categorized expense (e.g., amount, description, date, category

**Initialization:** 📂 This attempts to **load previously saved expenses** from save.csv.🧼 If the file doesn’t exist, save\_list() handles it smoothly, and expenses simply starts as an **empty list**—ensuring the app always starts cleanly. 🔄

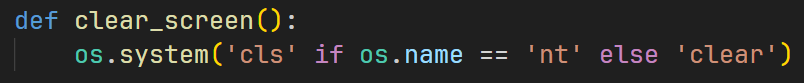
spending\_limits Dictionary (Global Data Store for Budgets) 🔒



**🎯Purpose:** This dictionary stores **user-defined spending limits (budgets)** for different expense categories. The keys are category names (e.g., "food"), and the values are the numerical limits (e.g., 5000.0).

2.3 Utility Functions ⚙️

1 . clear\_screen() 🧹



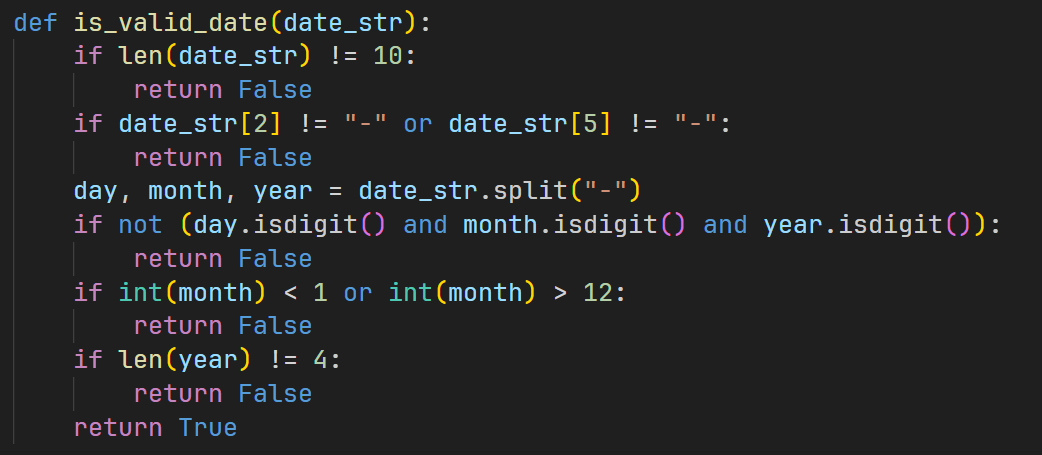
**Purpose:** To clear the terminal or command prompt screen. This provides a cleaner user interface by removing previous output before displaying new menus or information, improving readability. ✨

**os.system():**

* **🧠 os.name** returns a string that identifies the OS.
* **🪟 If 'nt'**, the OS is Windows → runs the cls command.
* **🐧 Else**, it's a Unix-like system (Linux/macOS) → runs the clear command.

This ensures **platform-independent** screen clearing for a smooth user experience across devices. 🌐✅

2 . is\_valid\_date(date\_str) 📅



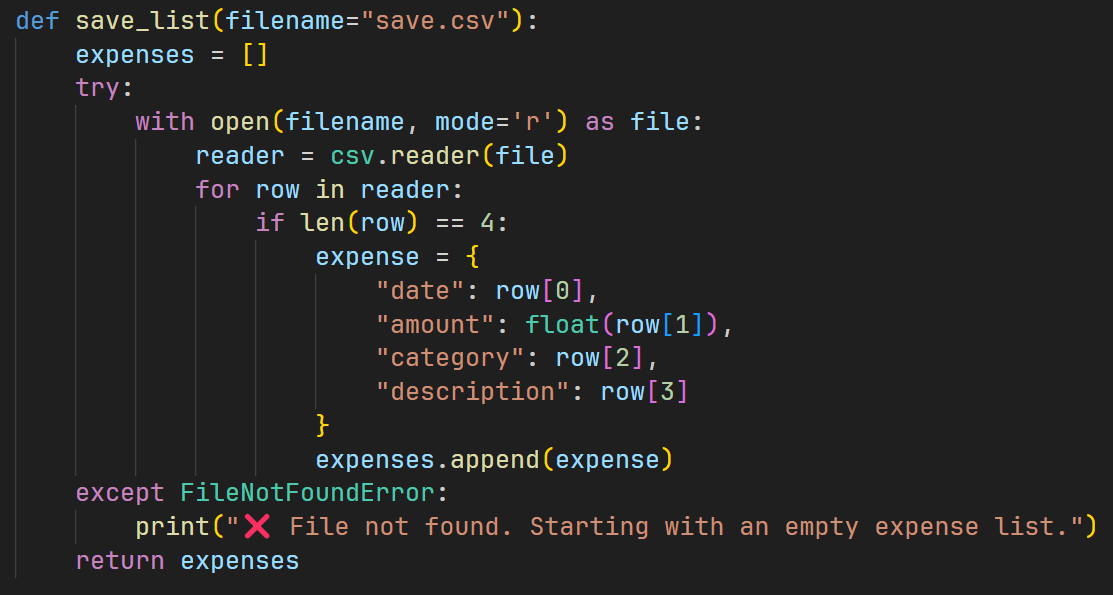
**🛡️Purpose:** Validates whether a given date string (date\_str) follows the **DD-MM-YYYY** format. This is essential to maintain **data consistency** and avoid errors in date-based features like filtering or summaries.

**🧠 Step-by-Step Logic:**

1. **🔢 Length Check:** Verifies that the string is **exactly 10 characters** (e.g., "01-01-2023"). If not, it's invalid.
2. **➖ Hyphen Placement:** Ensures that hyphens are correctly placed at **index 2 and index 5** ("DD-MM-YYYY" structure).
3. **🔍 Component Extraction:** Uses date\_str.split("-") to split the input into day, month, and year.
4. **🔢 Digit Check:** Confirms that all parts (day, month, year) contain **only digits** using isdigit().
5. **📆 Month Range:** Converts month to an integer and checks that it’s between **1 and 12** (inclusive).
6. **📅 Year Length:** Checks that the year part contains **exactly 4 digits**.

**✅ Return Value:** Returns True if **all conditions are satisfied**, otherwise False. This boolean result is perfect for use in conditional checks within functions like add\_expense().

3 . Data Persistence Modules (save\_list() & write\_expenses\_to\_file()) 💾



**🔄Purpose:** This function loads all **previously saved expense records** from a CSV file (default: save.csv) into memory when the program starts. This allows the application to **resume seamlessly** from past sessions.

**expenses = []**: Initializes an empty list. This list will be populated with expense dictionaries read from the file.

**🛡️ try...except FileNotFoundError: Robust File Handling**

* **try: Attempts to open and read from the CSV file.**
* **except FileNotFoundError: If save.csv doesn’t exist (e.g., on first run), a friendly message is shown, and an empty list is returned. This prevents crashes and ensures a smooth first-time experience. ✨**

**📖 with open(filename, mode='r') as file:**

* **Opens the file in read mode ('r')**
* **The with block ensures the file is automatically closed afterward—avoiding leaks or locking issues. 🔐**

**🔄 reader = csv.reader(file)  
Creates a CSV reader to iterate over rows in the file.  
Each row is a list of strings (e.g., ['01-01-2023', '100.0', 'food', 'Lunch']).**

**🔍 for row in reader:  
Loops through each row in the file.**

* **if len(row) == 4:Ensures each row has exactly 4 fields (date, amount, category, description).This is a basic integrity check to skip corrupted or malformed lines. 🧹**

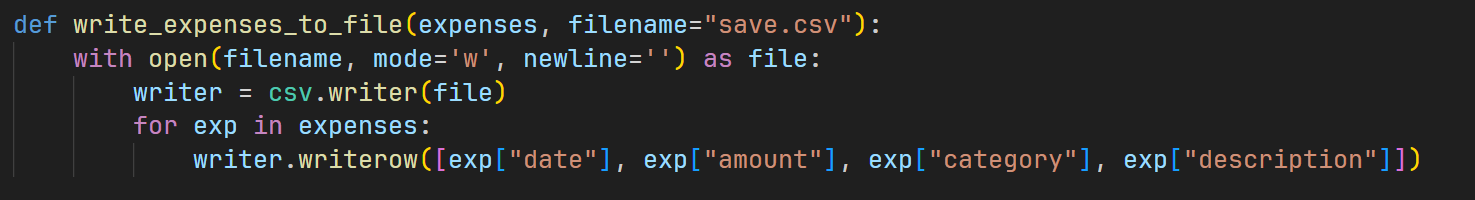
**expense = {...}**: Each valid row (which is a list of strings) is converted into a Python dictionary. This dictionary uses descriptive keys ("date", "amount", "category", "description") to make the expense data easy to access and manipulate within the rest of the Python script.

**"amount": 💵 amount is converted to float to support decimal values for calculations.**

**➕expenses.append(expense)**: Each dictionary is added to the expenses list—building up the **in-memory record** of transactions.

**📤 return expenses**: Returns the full list of loaded expenses, ready for use by the rest of the app.

4 . write\_expenses\_to\_file(expenses, filename="save.csv")



**✍️Purpose:** This function ensures that all expense records currently stored in memory (expenses list) are **permanently saved** to a CSV file (save.csv by default). It captures all new entries and updates made during the session, preserving data across program runs.

**with open(filename, mode='w', newline='') as file:**

* open(filename, mode='w'): Opens the file named filename in **write mode ('w')**. **This is a crucial point:** using 'w' mode means that if save.csv already exists, its **entire content will be overwritten** by the new data. This ensures that the file always precisely reflects the expenses list's current state. If the file doesn't exist, it will be created.
* newline=''\*\*: This argument is **absolutely vital** when writing CSV files in Python using the csv module. It prevents the csv.writer from adding extra blank rows between each actual data row in the CSV file, which can lead to malformed CSVs and issues when reading them back.

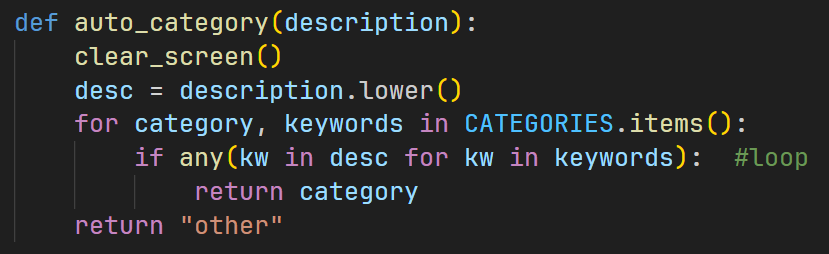
**writer = csv.writer(file)**: Creates a writer object from the csv module. This object provides methods to write rows to the CSV file.

**for exp in expenses:** Loops through each individual expense dictionary present in the expenses list (which holds all the categorized expense data in memory).

**writer.writerow([...])**: For each exp dictionary, this method writes a single row to the CSV file.

* [exp["date"], exp["amount"], exp["category"], exp["description"]]: The values from the exp dictionary are extracted and put into a list in a specific order. This order **must match** the order expected when reading the CSV file (by save\_list()) to maintain data consistency and integrity. ➡️

5 . The Core Categorization Logic (auto\_category(description)) 🧠



**Purpose:** Automatically assigns the **most relevant category** to an expense based on its **natural language description**. This saves users from manually selecting categories, making the experience faster and more intuitive.

**Input Parameter:** It accepts a single string argument, description, which is the user-provided textual detail of the expense (e.g., "Lunch at a restaurant," "Bus ticket," "Netflix subscription").

**🔍 Line-by-Line Logic:**

1. **desc = description.lower()**
   * Converts the input to **lowercase** for **case-insensitive matching**.
   * Ensures "Coffee", "coffee", and "COFFEE" are treated the same. 🔡
2. **for category, keywords in CATEGORIES.items():**
   * Iterates over each **category** (like "food", "transport") and its **associated keywords** from the CATEGORIES dictionary. 🔄
3. **if any(kw in desc for kw in keywords):**
   * The **core logic**:
     + Checks if **any keyword** from the current category appears in the desc.
     + Example: if "coffee" is in the "food" category, and the input is "Grabbed morning coffee", it'll match. ☕
     + any(...) returns True **as soon as one match is found**, optimizing performance.
4. **return category**
   * If a match is found, the function immediately **returns the matched category**.
   * Efficient exit once a valid category is detected. ✅🏁
5. **return "other"**
   * If **no keywords match**, returns "other" as a **fallback**.
   * Guarantees every expense has a classification, preserving data consistency. 📦🤷‍♂️

**6 . Expense Management (add\_expense(expenses), is\_valid\_date()) ➕**

add\_expense(expenses)



**🎯Purpose: Guides the user through adding a new expense, uses auto-categorization for smart classification, allows manual override, and immediately saves the data for persistence.**

**🧹clear\_screen() :** Clears the console for a fresh, clean input experience.

**📅 Date Input:**

* **Prompts user: “Manually enter date? (Y/N)”**
* **If ‘Y’, calls is\_valid\_date() to check format (DD-MM-YYYY). If invalid, shows error and exits early. ⚠️**
* **If ‘N’, automatically uses the current date (datetime.now().strftime("%d-%m-%Y")).**
* **Invalid choice also exits early.**

**💵 Amount Input:**

* **Uses try-except to convert input to float for monetary values.**
* **On invalid number input (e.g., letters), prints error and returns early. 💸**

**Description Input:** Collects the textual description of the expense, which is crucial for categorization. 📝

**🧠 Auto Categorization:**

* **Calls auto\_category(description) to assign the most relevant category automatically.**

**🔄 User Confirmation / Override:**

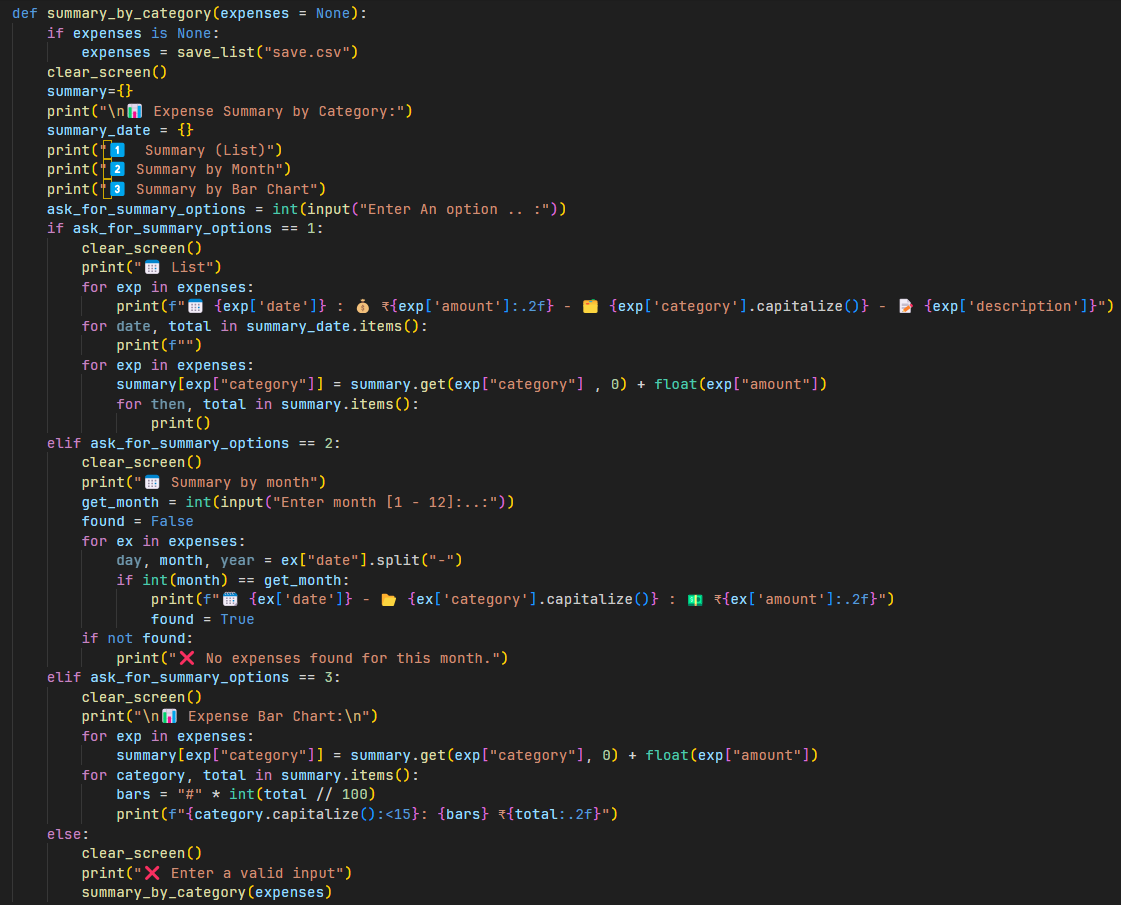
* **Displays auto-suggested category: "📂 Auto categorized as: {category.capitalize()}"**
* **Prompts user: “Confirm or Change Category? (Y/N)”**
* **If user wants to change, lists all categories and validates new input.**
* **Updates category if valid, ensuring user control over classification.**

**📦 Expense Assembly & Storage:**

* **Creates a dictionary with date, amount, description, and category.**
* **Appends this dictionary to the global expenses list in memory.**
* **Calls write\_expenses\_to\_file(expenses) to save all expenses to save.csv immediately, guaranteeing no data loss. 💾**

**Confirmation:** A ✅ Expense added successfully! message is printed to confirm the operation.

7 . Data Reporting and Summaries (summary\_by\_category(expenses)) 📊



**🎯 Purpose:  
Provides multiple ways for users to explore their categorized expenses, turning raw data into insightful summaries and visualizations.**

**🛡️ Safeguard: if expenses is None: Ensures up-to-date data by loading from "save.csv" if no expenses list is provided.**

**summary = {}**: creates an empty dictionary to accumulate category totals for bar chart visualization.

**🔢 User Options:  
1️⃣ Detailed List Summary**

* **Iterates over every expense in the list.**
* **Prints formatted details: date, amount, category (capitalized), description.**
* **Provides a full chronological log of all categorized expenses. 📜**

**2️⃣ Monthly Summary**

* **Prompts user to enter a month number [1-12].**
* **Validates input with try-except for robustness.**
* **Filters expenses by matching the month extracted from expense["date"].**
* **Prints details of expenses from that month. 🗓️**
* **If no expenses found for the month, informs the user.**

**3️⃣ Bar Chart Visualization**

* **Aggregates total spending per category by summing amounts in summary dictionary.**
* **For each category, generates a horizontal bar made of # characters proportional to spending (1 # = ₹100).**
* **Prints a neat line showing category name (capitalized), bar graph, and total amount formatted to two decimals.**
* **Enables quick visual comparison of spending across categories. 📊**

**⚠️ Invalid Option Handling:**

* If input is not 1, 2, or 3, prints an error message.
* Recursively calls summary\_by\_category(expenses) to let the user try again.

8 . Spending Limits and Monitoring (option()) 🔒

**Purpose:** To enable users to set **spending limits for specific expense categories** and to view their current spending against these limits. This feature directly leverages the power of categorization to provide actionable budgeting insights. 🎯

**spending\_limits = {}:** A global dictionary where keys are category names and values are user-defined numerical spending limits.

**User Menu:** Presents options to Add or update a spending limit (Option 1), View current spending and limits (Option 2), or return to the Main Menu (Option 3).

**Looping & Return:** After completing an action, the user can choose to continue in this menu or quit, which returns to main\_menu(), maintaining smooth application navigation.



**Option 1: Add / Update Limit ✏️**

* **Prompts user to enter a category name and a limit amount.**
* **Validates inputs:**
  + **Category name cannot be empty.**
  + **Limit must be a non-negative float (with error handling for invalid inputs).**
* **Updates spending\_limits dictionary with the new or modified limit:  
  spending\_limits[cate] = limit**

**Option 2: View Spending & Limits 📊**

* **Reloads latest expenses from "save.csv" to ensure accuracy:  
  expenses = save\_list("save.csv")**
* **Aggregates total spending per category into category\_totals. This relies on expenses already being categorized.**
* **For each category in category\_totals:**
  + **Retrieves the user-defined limit from spending\_limits (or None if unset).**
  + **Compares spent amount vs. limit and determines status:**
    - **❌ Over — spending exceeded the limit**
    - **✅ OK — spending within or equal to limit**
    - **— No limit set**
* **Displays results in a neatly formatted table showing:  
  Category | Amount Spent | Limit | Status**
* **Provides instant, actionable feedback on budget adherence based on categorization.**

9 . Search Expenses (search\_expenses()) 🔍



**🕵️‍♀️ Purpose: Helps users quickly find specific expenses by searching for keywords in both the description and category, making data retrieval intuitive and efficient.**

**🔍 Keyword Input: keyword = input(...).strip().lower()**

* **Prompts user to enter a search term.**
* **.strip() removes extra spaces.**
* **.lower() ensures the search is case-insensitive, so "Food" = "food" = "FOOD".**

**try...except FileNotFoundError:** Handles the case where save.csv might not exist yet, preventing a crash. **with open('save.csv', mode='r', newline='') as file::** Opens the CSV file in read mode.

**reader = csv.DictReader(file, fieldnames=["date", "amount", "category", "description"])**: This is a powerful feature of the csv module.

* **Reads each row as a dictionary instead of a list.**
* **Since the file lacks headers, column names are manually assigned.**
* **Enables readable, named access like row["description"].**

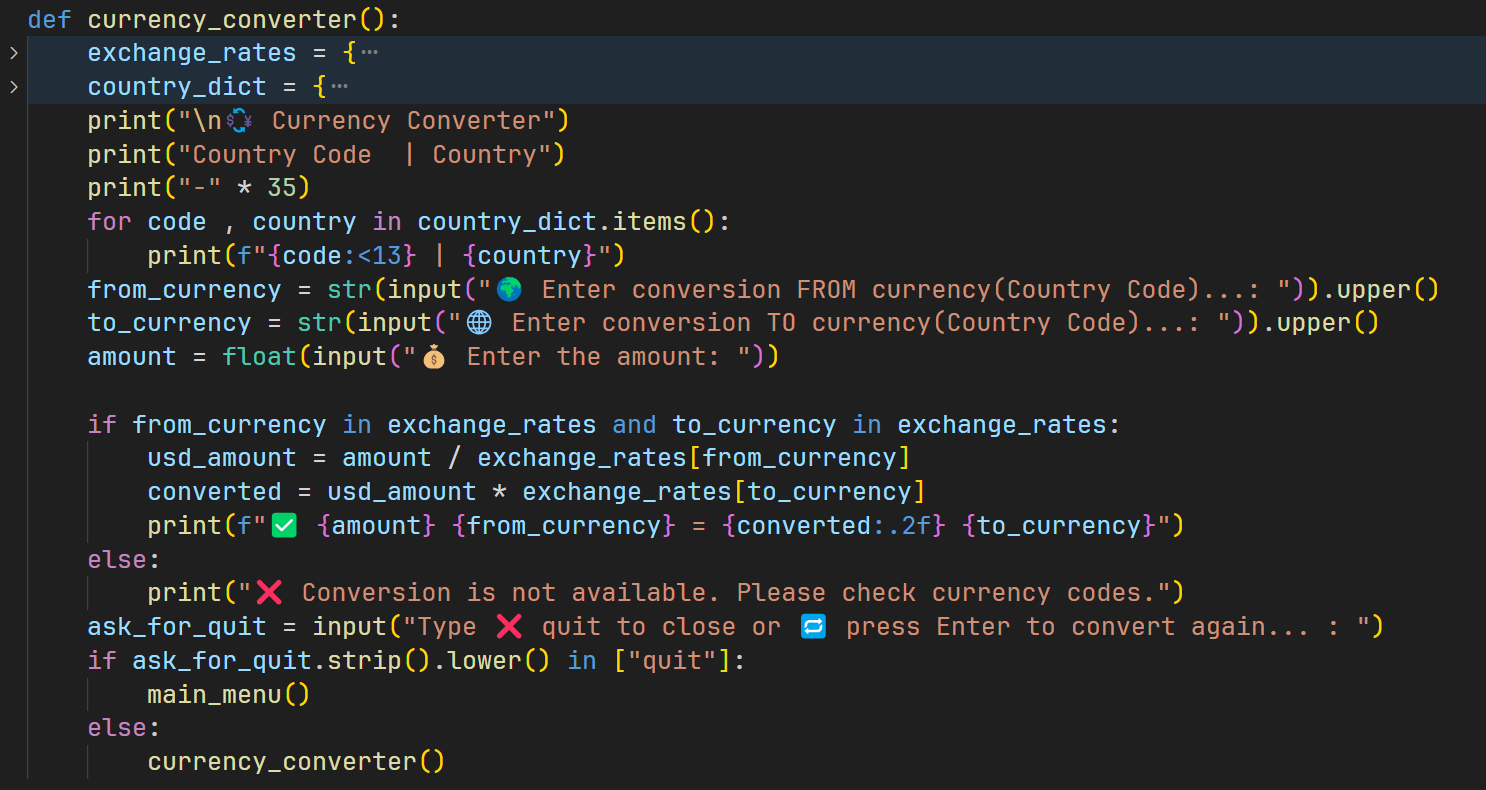
**Core Search Logic (if (keyword in row["description"].lower() or keyword in row["category"].lower()):)**:

Converts both fields to lowercase to ensure **case-insensitive matching**. Matches **any substring** of the keyword in either field.

**found = False flag:** This flag is used to determine if any matching expenses were found during the loop. If found remains False after checking all rows, a "No matching expenses found" message is displayed.

**Looping:** After a search, the user is prompted to perform another search or quit (which returns to the main\_menu()).

10 . Currency Converter (currency\_converter()) 💱



🌍 **Purpose**: Converts a specified amount between two currencies — perfect for budgeting, travel, or international use. 💸

💱 **exchange\_rates**: Hardcoded rates with USD as the base (e.g., "INR": 86.76 ➡️ 1 USD = ₹86.76). For projects, static values are fine. ✅

🗺️ **country\_dict**: Maps currency codes to country names for easier understanding (e.g., "EUR": "European Union"). 🇪🇺

🔤 **User Input**:

* FROM & TO currency codes (auto-uppercase).
* Amount to convert (validated via try-except for float).

11 . Main Application Loop (main\_menu()) 🏡

**🖥️ Purpose: Acts as the central hub for all user interactions, looping through options and directing the user to specific functions based on their choices.**

**while True:** This initiates an **infinite loop**. The menu and input prompt will continuously reappear until a break statement is encountered (specifically, when the user chooses Exit). This provides the persistent interactive environment for the application. ♾️

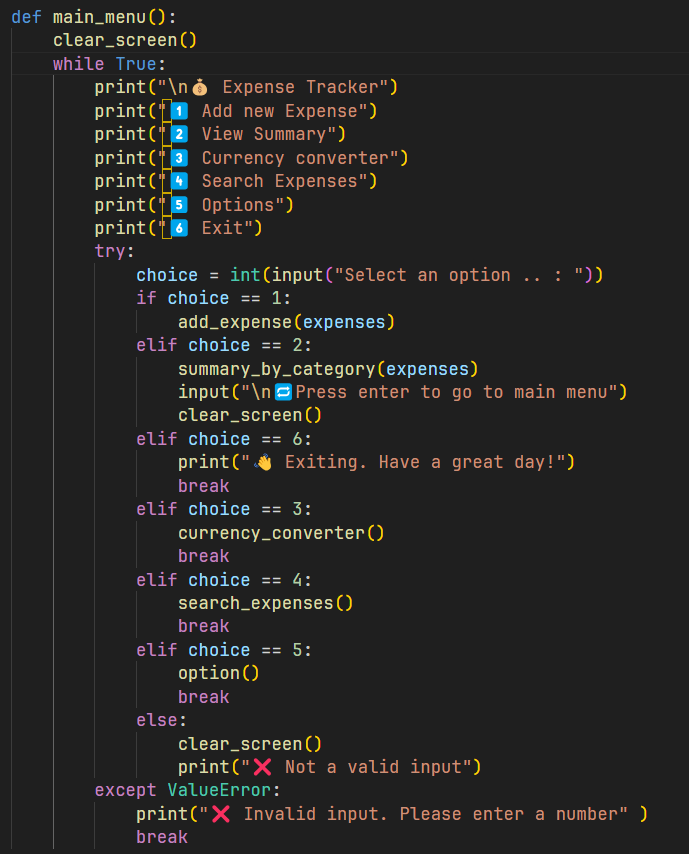
**Menu Display:** Prints a clear, numbered list of options that represent the various functionalities available in the application.

**Exit Mechanism (break):** When choice == 6 (Exit), the break statement is executed. This immediately terminates the while True loop, causing the main\_menu() function to finish its execution, which in turn ends the entire Python program gracefully. 👋

**try...except ValueError:** This block is used for robust error handling.

* choice = int(input("Select an option .. : ")): It attempts to convert the user's input to an integer.
* prompting the user for input again. This prevents the program from crashing due to bad input. 🚫
* If the user types something that *cannot* be converted to an integer (e.g., "abc"), a ValueError occurs.
* The except ValueError: block catches this error, prints an "Invalid input" message, and then (because the break was removed from here) the while True loop continues,
  + summary\_by\_category(expenses) for viewing summaries (includes a pause input() and clear\_screen() for readability after a summary).
  + currency\_converter(), search\_expenses(), option() for their respective features.

🔀 **Conditional Dispatch**: Uses if/elif/else to run functions based on user input (e.g., add\_expense(expenses) for adding). If input doesn't match any option, it shows "Not a valid input" and re-prompts. 🚫

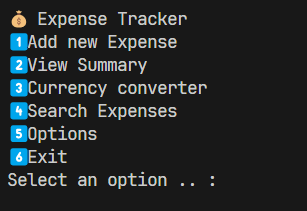


12 . Program Execution Flow (End of Script) 🎬

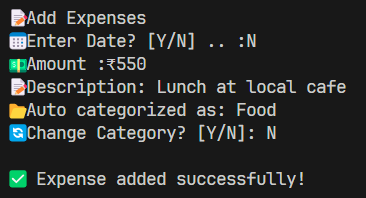
* 📂 expenses = save\_list("save.csv"): Loads all saved expenses from the CSV file into memory at startup. If the file doesn’t exist, it starts with an empty list.
* 🖥️ main\_menu(): Launches the main interactive loop, displaying the menu and waiting for user input until they choose to exit. 🚪

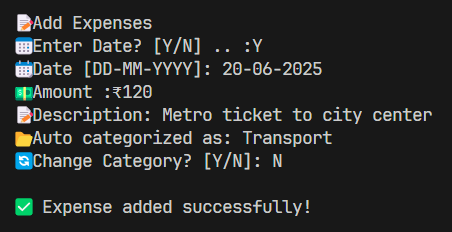
3 . OUTPUT

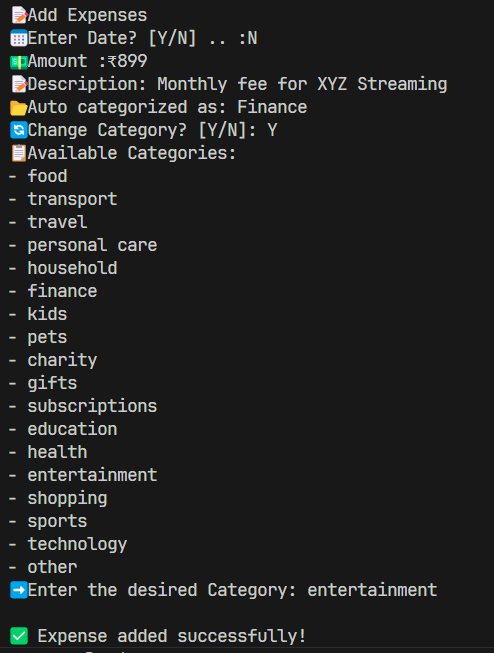
1 . MAIN MENU:



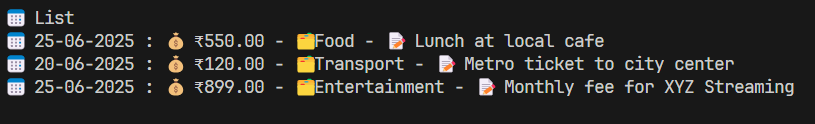
2 . ADDING THE EXPENSE:



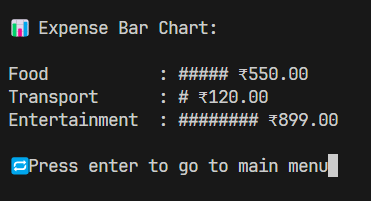




3 . Viewing Summary (List) 📜



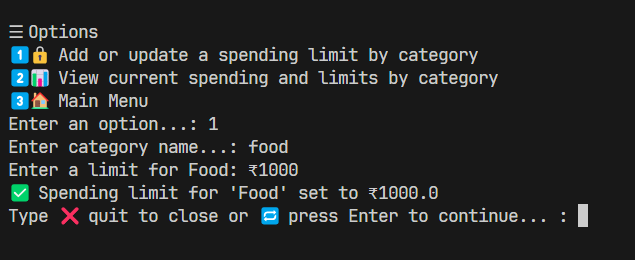
4 . Viewing Summary (Bar Chart) 📊



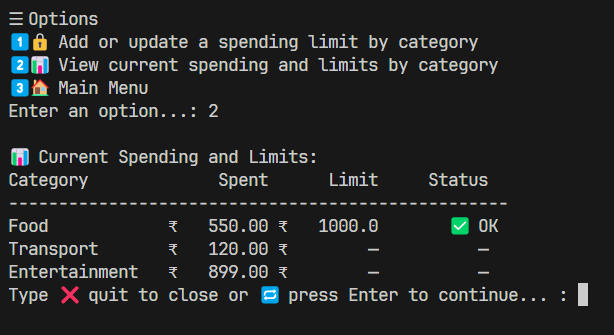
5 . Viewing Summary (Month List)



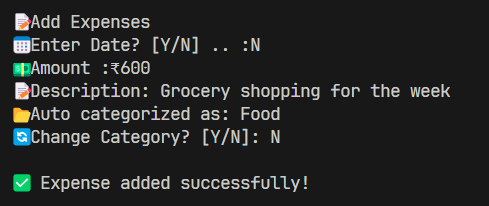
6 . Setting a Spending Limit (Food) 🍔🔒



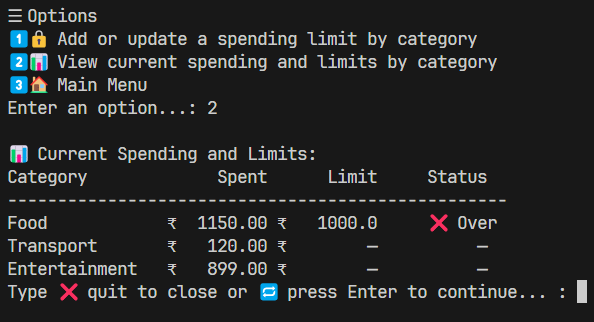
7 . Viewing Spending Limits 📊✅



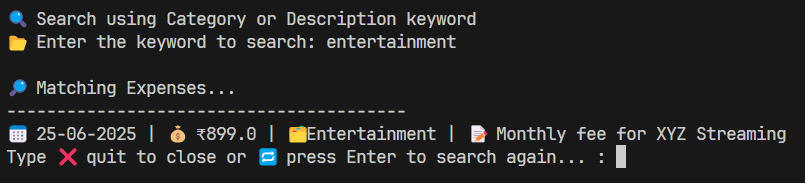
8 . **Adding Another Food Expense (Exceeding Limit) 🍔❌**

****

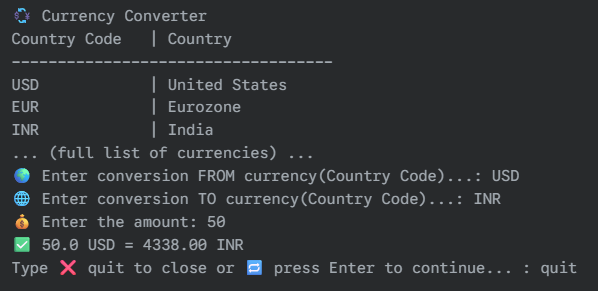
**9 . Viewing Spending Limits (After Exceeding) 📊❌**

****

**10 . Searching Expenses (by Category Keyword) 🔍**

****

**11 . Using the Currency Converter 💱**



12 . Exiting the Application 👋

