**Command-Line Password Manager**

***Project Description***

This project is a secure Python-based command-line application designed to manage and protect user passwords. It uses a **master password** to control access and applies **AES encryption** (via the cryptography library) to store all credentials safely in a local file.

Users can **add**, **view**, and manage credentials for multiple websites or apps. Passwords are encrypted and stored in a JSON file (vault.json), and can only be accessed after entering the correct master key. The project uses getpass to securely input passwords without displaying them on the screen.

**Key Highlights**

* Local password storage with AES encryption
* Simple command-line interface (CLI)
* Passwords hidden during input using getpass
* No internet or external database required — completely offline and secure

**Technologies Used**

* **Python 3.x** – Core language for scripting
* **cryptography** – Used for AES encryption (Fernet)
* **getpass** – Hides password input in terminal
* **json** – For storing encrypted data in a structured format
* **base64**, os, hashlib – Utilities for secure key derivation and file handling

**Architectural Highlights & Design Choices**

* **Master Password-Based Access**:  
  Users must enter a master password at runtime. This password is used to derive an AES encryption key via PBKDF2HMAC with a salt.
* **AES Encryption via Fernet**:  
  All site credentials (username + password) are encrypted using a symmetric key with the Fernet protocol, ensuring high security.
* **Local Vault File (vault.json)**:  
  The password vault is stored as an encrypted JSON file locally. No cloud or third-party storage is involved, reducing attack surfaces.
* **Modular Design**:  
  The program separates concerns like encryption, file handling, and user interaction into functions for easy updates and debugging.

**Challenges and Solutions**

| **Challenge** | **Solution** |
| --- | --- |
| Securely storing passwords | Used cryptography.fernet to encrypt data |
| Protecting user input | Implemented getpass to hide password entries |
| Preventing brute force or misuse | Encrypted data using a derived key from a salted master password |
| First-run file setup | Automatically generates salt and vault file on the first execution |

**Future Enhancements**

* **Search Functionality**: Quickly find credentials by site name
* **Delete or Update Entries**: Allow modifying or removing existing passwords
* **Export/Backup**: Secure export and import of encrypted vaults
* **Auto-Lock**: Lock the vault after a period of inactivity
* **Brute Force Protection**: Add attempt limits and cooldown timers
* **Optional Cloud Sync**: Encrypted vault sync to cloud (with user consent)

# Expense & Budget Tracker

## **1. Project Overview**

The **Expense & Budget Tracker** is a GUI-based personal finance tool built in **Python using Tkinter**.  
Its primary purpose is to help users **log daily expenses**, **set monthly budgets**, and **analyze their spending habits** through clear reports and visual charts.

This project simplifies everyday money management by allowing users to enter expense details, store them locally, and view summaries that promote better financial decision-making.

## **2. Key Features**

| **Feature** | **Description** |
| --- | --- |
| 💵 Add Daily Expense | Users can enter an amount, category (e.g., Food, Travel), and notes |
| 🧾 Monthly Budget | Allows setting a fixed monthly budget and checks how much remains |
| 📊 View Reports | Shows total spent, remaining budget, and breakdown per category |
| 📈 Pie Chart Visualization | Generates a visual summary of expenses using Matplotlib |
| 💾 Persistent Storage | All expense data is stored in a CSV file for portability |

## **3. Technologies Used**

| **Technology** | **Purpose** |
| --- | --- |
| **Python 3.x** | Core programming language |
| **Tkinter** | GUI framework to create forms and buttons |
| **CSV** | Lightweight file-based storage for expenses |
| **Matplotlib** | To create pie charts and visual summaries |
| **Datetime** | For recording current date with each expense |

## **4. Architectural Highlights & Design Choices**

### GUI-Centric Design:

* Chose **Tkinter** for simplicity and wide support with Python.
* Designed the app for **non-technical users**—everything is done via buttons and input fields, no terminal use.

### File-Based Data Storage:

* **CSV** was chosen over databases to keep it **lightweight, portable, and dependency-free**.
* Data entries are **timestamped**, enabling future filtering by date.

### Modular Functional Layout:

* UI and logic are **separated cleanly**.
* Each action (add expense, view report, show chart) is encapsulated in a method.

### Visual Reporting:

* Used **Matplotlib** to turn expense data into an easy-to-understand pie chart.
* Helps users make informed decisions visually.

## **5. Challenges and Solutions**

| **Challenge** | **Solution** |
| --- | --- |
| Input validation (e.g., non-numeric values) | Added try-except blocks and error prompts using messagebox |
| Avoiding UI clutter | Created a compact layout using grid() with proper padding |
| Data storage with no SQL | Used structured CSV format, readable and easy to manipulate |
| File not found / CSV missing on first run | Script checks and auto-creates data.csv if missing |
| User forgets category or input | Required field checks and defaults were added |

## **6. Future Enhancements**

| **Improvement** | **Description** |
| --- | --- |
| User Accounts | Add login system to support multiple users |
| Switch to SQLite or MongoDB | Replace CSV with a real database for scalability |
| Date-Based Filtering | Allow users to filter by week/month or custom date range |
| Export Reports | Export summaries to PDF or Excel for offline access |
| Web Version | Rebuild using Flask or Django to make it available online |
| Mobile App | Create an Android version using Kivy or React Native |

## **Dynamic Web Scraper + Export to Excel**

### ****Project Description****

This project is a **dynamic web scraping tool** built using Python, Selenium, and OpenPyXL that extracts structured data (book titles and prices) from a paginated website: Books to Scrape. It automates browser interaction, handles dynamic content loading, and seamlessly exports the scraped data to an Excel spreadsheet.

The scraper is easily customizable for other sites and is intended to demonstrate real-world web automation, data extraction, and reporting using modern Python tools.

### ****Features****

* 🔄 **Pagination Support**: Automatically navigates through multiple pages.
* 🧠 **Dynamic Content Handling**: Uses Selenium to interact with JavaScript-rendered content.
* 📤 **Excel Export**: Data is saved in a clean Excel file (.xlsx) using OpenPyXL.
* 🔍 **Targeted Data Extraction**: Scrapes specific information like title and price.
* 🧩 **Modular Code Structure**: Organized functions for scraping, exporting, and handling navigation.
* 🛠️ **Easy Customization**: Adaptable for scraping other websites with minimal changes.

### ****Technologies Used****

| **Tool** | **Purpose** |
| --- | --- |
| Python 3.x | Core programming language |
| Selenium | Web browser automation and scraping |
| OpenPyXL | Writing data to Excel files |
| ChromeDriver | Driver for controlling Chrome browser |
| Google Chrome | Headed browser for rendering web content |

### ****Architectural Highlights & Design Choices****

* **Selenium for Rendering Dynamic Content**  
  Chosen over BeautifulSoup because the target website could include JavaScript-rendered elements. Selenium handles real browser interaction.
* **Pagination Loop**  
  Implemented using a loop that dynamically updates the page URL (page-1.html, page-2.html, etc.), enabling scalable data scraping.
* **Excel Export via OpenPyXL**  
  A structured and well-supported method to export tabular data into .xlsx format, allowing easy viewing and further processing.
* **Delays with time.sleep()**  
  Used for simplicity and reliability to wait for pages to load. Can be replaced with WebDriverWait for more advanced handling.

### ****Challenges and Solutions****

| **Challenge** | **Solution** |
| --- | --- |
| Unicode errors from Windows file paths | Used raw strings (r"path\to\file") to fix \u escape errors. |
| Module errors like No module named selenium | Clarified proper pip install selenium usage within PyCharm/terminal. |
| Matching ChromeDriver version | Added instructions to match driver with the installed Chrome version. |
| Dynamic element loading delays | Introduced time.sleep() to allow pages to fully render. |

### ****Future Enhancements****

* ✅ **Headless Mode**: Run Chrome in headless mode to make the scraper faster and silent.
* ✅ **CSV Export Option**: Allow users to choose between .xlsx or .csv formats.
* ✅ **WebDriverWait**: Use explicit waits instead of time.sleep() for better control.
* ✅ **Logging System**: Add logs to track scraping progress and errors.
* ✅ **User-Agent Rotation / Proxies**: Add support for anonymous scraping and anti-blocking measures.
* ✅ **GUI Interface**: Build a simple UI using Tkinter or PyQt to allow non-coders to run the tool.