# Daily Calorie Tracker – Project Overview

The **Daily Calorie Tracker** is a simple Python console application designed to help users record and monitor their daily food intake and calories. Through a command-line interface, the user specifies how many meals they will log, enters each meal’s name and calorie count, and then receives a summary of total and average calories. The program also allows setting a daily calorie limit, checks whether the user stays within that limit, and optionally saves a detailed report to a log file. This helps users keep track of their eating habits over time. The project is implemented in standard Python 3 and does **not** rely on any external libraries (beyond the built-in datetime module for timestamps).

## Project Structure and Files

The project is contained in a single directory named Calorie-Tracking-Console-main, which includes the following files:

* **README.md**: A markdown file providing a high-level description of the project, its features, and usage instructions.
* **tracker.py**: The main Python script containing the core program logic (input handling, calculation, output, and file writing).
* **calorie\_log.txt**: A text file used as a simple log to record saved sessions. The program appends data to this file when the user chooses to save their input. Initially it may contain example entries.
* **output.png**: An example screenshot or image illustrating the program’s output format (sample console output). This is for demonstration purposes and not part of the program logic itself.
* **Calorie-Tracking-Console-main/**: A top-level folder (the root of the project archive) that contains all the above files.

Each file has a clear purpose: the README documents the project, tracker.py implements the application, calorie\_log.txt stores data, and output.png shows an example of the running program’s output.

## Components and Functionality

### 1. tracker.py – Main Script

The heart of the project is the tracker.py script. It defines a single function, main(), which coordinates all user interaction and computation. When the script is run (if \_\_name\_\_ == "\_\_main\_\_": main()), it performs the following sequence:

1. **Welcome Message**: Prints a greeting to the user.
2. **Ask for Number of Meals**:
3. Prompts the user: *“How many meals would you like to enter?”*.
4. Tries to convert the input to an integer. If the input is not a valid integer, it prints an error message and exits.
5. Checks that the number is positive. If not (e.g., zero or negative), it prints a message and exits.
6. **Collect Meal Names and Calories**:
7. Initializes two lists: meal\_names and meal\_calories.
8. For each meal (from 1 to the number specified), the program:
   * Prompts: *“Enter the name of meal X”* and appends the name to meal\_names.
   * In a loop, prompts: *“Enter the calorie amount for '<meal name>'”*.
   * Attempts to convert the input to a floating-point number (to allow decimal calories).
   * If the input is not numeric, it catches the ValueError, prints a friendly message, and repeats the prompt.
   * If the number is negative, it prints a warning ("Calories cannot be negative") and asks again.
   * Once a valid non-negative number is entered, it appends it to meal\_calories.
9. **Calculate Total and Average Calories**:
10. After collecting all meals, it computes:
    * total\_calories = sum(meal\_calories)
    * average\_calories = total\_calories / num\_meals
11. These values represent the sum and mean of the entered calories.
12. **Display the Meal Table**:
13. Prints a neatly formatted table showing each meal and its calories, followed by the total and average.
14. The format uses f-strings and string formatting (f"{name:<20}{cal}") to align columns. For example:

* Meal Name Calories  
  ------------------------------  
  Breakfast 350.0  
  Lunch 600.0  
  Dinner 800.0  
  ------------------------------  
  Total 1750.0  
  Average 583.33

1. The average is displayed to two decimal places using "{average\_calories:.2f}".
2. **Calorie Limit Check**:
3. Prompts the user: *“Enter your daily calorie limit:”*.
4. Tries to convert this input to a float. If the input is invalid, it prints an error message and exits.
5. Compares total\_calories with the provided limit.
   * If total\_calories > limit, it prints a warning: "Warning: You have exceeded your daily calorie limit!"
   * Otherwise, it prints: "You are within your daily calorie limit."
6. **Option to Save Session**:
7. Asks: *“Would you like to save this session to a file? (yes/no)”*.
8. Converts the answer to lowercase and checks if it is 'yes'.
9. If the user chooses to save:
   * Records the current timestamp using datetime.now().strftime("%Y-%m-%d %H:%M:%S").
   * Opens calorie\_log.txt in append mode (with open(filename, 'a') as file:).
   * Writes the timestamp and the same table of meals, total, average, limit, and status into the file, with formatting similar to the console output. For example:

* Timestamp: 2025-10-24 09:01:54  
  Meal Name Calories  
  ------------------------------  
  Breakfast 350.0  
  Lunch 600.0  
  Dinner 800.0  
  ------------------------------  
  Total 1750.0  
  Average 583.33  
  Calorie Limit: 2000.0  
  Status: Within limit
  + After writing, it prints a confirmation: e.g., "Session saved to calorie\_log.txt."
  + If the user answers anything other than 'yes', the program simply ends.

1. **Program Termination**:
2. After completing these steps (and possibly saving), the main() function ends, and since there is no loop for multiple days, the script terminates.

#### Input Validation and Error Handling

* The program rigorously checks user inputs. At each stage where numeric input is required (number of meals, each meal’s calories, daily limit), it wraps the conversion in a try/except block to catch ValueError. If a conversion fails, it prints a message like “Invalid input. Please enter a numeric value” and exits or repeats as appropriate.
* For calorie inputs, there is an additional check that disallows negative values, prompting the user until a non-negative value is given.
* For the number of meals, it ensures the integer is positive; otherwise, it prints a message and stops.

### 2. calorie\_log.txt – Data Log

This plain text file serves as a simple log or diary of saved sessions. When the user chooses to save a session, the program appends the session data to calorie\_log.txt. Each saved session entry includes:

* A **timestamp** (date and time) indicating when the entry was recorded.
* A formatted table of meal names and calories (mirroring the console output).
* The total and average calories.
* The specified calorie limit for that session.
* A status line ("Within limit" or "Exceeded limit").

Each session entry is separated by a blank line for readability. The file grows cumulatively as more sessions are saved, allowing the user to review past data over multiple runs.

### 3. README.md – Documentation

The README provides an overview and usage guidance. Key points include:

* **Description**: It introduces the tracker as a command-line tool for logging meals and calories.
* **Features**: Lists features like meal logging, calorie summation, limit checking, formatted reports, and file saving.
* **Getting Started**: Explains how to run the program (though the excerpt in the README is truncated in the project, it likely contains instructions).
* **Technologies**: Notes that the project uses Python 3.x and only the standard library.
* **Author Info**: Includes the author’s name and course details.
* **Acknowledgments**: Credits any references or libraries (in this case, Python docs).
* **Academic Integrity Note**: States that the work is done individually.

The README is mostly for human readers and does not affect the code execution.

### 4. output.png – Example Output (Illustration)

This image file (output.png) appears to be a screenshot of a sample run of the program. It likely shows what the console output looks like when the program is used (for example, the formatted table of meals and calories). This is purely illustrative and not part of the executable code. Including an example screenshot can help users see what to expect, but the logic is fully captured by the code description above.

## Code Interactions and Workflow

The program flow is entirely driven by user input and the main() function. There are no classes or multiple modules—just one script with one main function. The high-level workflow is:

1. **User Input Phase**: The script sequentially prompts the user for all needed data (number of meals, meal details, calorie limit, and save confirmation). Each input influences the next steps.
2. **Computation Phase**: Once all meals are entered, the program computes summaries (total and average).
3. **Output Phase**: It prints a report to the console. Then, depending on the user’s choice, it may write a similar report to a file.
4. **Termination**: The script ends after displaying messages (and saving if chosen).

Key interactions:

* The lists meal\_names and meal\_calories are kept in parallel: each meal’s name and corresponding calorie entry share the same index in these lists. The code iterates through them using zip(meal\_names, meal\_calories) when printing or writing the table.
* User’s decision to save (yes/no) controls whether the file-writing block executes.
* The datetime module is only used to timestamp saved sessions; no other external functionality is used.

## Important Logic and Formatting

* **Summation and Averages**: The logic for total (sum(meal\_calories)) and average (total\_calories / num\_meals) is straightforward arithmetic.
* **Table Formatting**: The code uses Python f-strings with formatting specifiers to align columns. For example, f"{name:<20}{cal}" ensures that the meal name is left-justified in a 20-character field, making the output table neat.
* **Threshold Check**: A simple if total\_calories > limit determines if the user has exceeded the daily goal.
* **File Writing**: The report is written with lines and separators ("-" \* 30) to clearly delineate columns and values. Using with open(..., 'a') ensures the file is properly closed after writing.

There are no complex algorithms beyond basic loops and arithmetic. The control flow is mainly linear, with conditional branches for error handling and the save/no-save decision.

## Dependencies

This project uses only Python’s standard library:

* **datetime (from datetime import datetime)**: To obtain the current date and time string for timestamping saved logs.
* No third-party packages are required. All other functionality (input/output, file handling, error catching) is built into base Python.

## Potential Improvements and Extensions

While the current project fulfills its basic purpose, several enhancements could be considered:

* **Modularization**: Break the main() function into smaller functions (e.g., one for input collection, one for computing, one for printing, one for saving). This would improve readability and make unit testing easier.
* **Data Validation**: Enhance validation (e.g., handle extremely large numbers, ensure meal names are valid strings, confirm save input options beyond yes/no).
* **User Interface**: Upgrade from simple text prompts to a menu-driven interface (looping menu) so the user can perform multiple actions (like re-running or editing entries) without restarting the program.
* **Persistent Data Structure**: Instead of appending to a single log file, consider storing data in a structured format (like CSV or JSON) for easier analysis or use a simple database.
* **Daily Tracking**: Allow the user to track multiple days, perhaps with a menu option to view past logs or summarize weekly/monthly intake.
* **Error Handling**: Improve robustness if the log file is not writable or handle keyboard interrupts gracefully.
* **Feature Expansion**: Add categories (carbs, protein, fat), incorporate nutritional information lookup, or implement a GUI/graphical chart for visual feedback.
* **Input Flexibility**: Accept inputs in different units or support copy-paste of meal lists.

Each of these would extend the functionality and user-friendliness of the tracker beyond its basic console form.

## Summary

In summary, the *Daily Calorie Tracker* is a concise console application that guides a user through logging meals and calories for the day, computes totals and averages, checks against a user-specified limit, and optionally saves the session to a log file. Its structure is minimal—comprised of a single script (tracker.py) and supporting files—but it cleanly demonstrates input handling, basic calculations, formatted output, and simple file I/O in Python. The code is straightforward and well-documented (via the README), making it easy for a developer to understand or extend further.