HyperionDev

Data Structures – Lists and Dictionaries

Model Answer Approach

Visit our website

Auto-graded task 1

This approach demonstrates a simple user interaction by prompting the user to enter their name and checking whether it matches the correct name, “john”. Initially, the program converts the user’s input to lowercase to ensure case insensitivity and stores it in the variable `name`. An empty list called `incorrect_names` is then initialised to keep track of any incorrect entries.

The program enters a `while` loop that continues until the user inputs “john”. If the entered name does not match “john”, the program appends the incorrect name to the `incorrect_names` list, allowing the user to see which names they have entered previously. The loop prompts the user to input their name again, repeating this process until the correct name is provided.

Once the user finally enters “john”, the program exits the loop and prints the list of incorrect names, providing a record of all previous attempts.

However, there are potential pitfalls to consider. If the user enters an empty string or a name that contains only whitespace characters, this will be stored as an incorrect entry, which might not be desirable. Additionally, if a user makes frequent mistakes, the `incorrect_names` list could grow large, potentially consuming unnecessary memory. To improve this program, one could implement input validation to handle empty inputs and provide more user-friendly feedback throughout the process. Overall, this program effectively illustrates basic looping and list manipulation concepts in Python while highlighting the importance of user input validation.

Auto-graded task 2

This approach calculates the total value of stock for a café by leveraging dictionaries to store item quantities and prices. It begins by defining a menu list and creating two dictionaries: `stock`, which holds the quantities of each menu item, and `price`, which contains the unit prices for those items.

The program initialises a variable called `total_stock` to zero, which will be used to accumulate the total value of the stock. It then iterates over the `stock` dictionary, computing the value of each item by multiplying its quantity by its corresponding price. This value is added to `total_stock`, and at the end of the loop, the total value is printed in a formatted string, displaying the total value with two decimal places for currency representation.

After the initial calculation, the program addresses potential mismatches between the stock and price dictionaries. A new `stock` dictionary is defined, this time including an additional item, “cake”, which does not have a corresponding price defined in the `price` dictionary. The program reinitialises the `total_stock` variable to zero and iterates through the updated stock dictionary once more.

Within this loop, an `if-else` statement checks whether each item exists in the price dictionary. If an item is found, its value is calculated and added to the total. If an item is not found, a message is printed to inform the user that the price for that item is not defined. Finally, the total value of the stock is printed again.

Possible pitfalls in this implementation include the hardcoding of item names, which may lead to errors if any changes are made to the menu items or their corresponding prices. If new items are added to the stock without corresponding prices in the `price` dictionary, the program will notify the user but will not include their value in the total, which could lead to a misleading total stock value. To improve the program, one could consider implementing functions to encapsulate the stock valuation logic, which would enhance readability and maintainability. Additionally, using a consistent method for managing new menu items would help prevent mismatches between the `stock` and `price` dictionaries, ensuring that the total value accurately reflects the entire stock.

Challenge

This approach creates a list of favourite movies and utilises the `enumerate` function to iterate through the list while obtaining both the index and the movie title. The list, `fav_movies`, contains five popular titles, each ending with a period.

In the `for` loop, `enumerate` is employed to provide an index (starting from zero) alongside each movie title. The `print` statement inside the loop outputs the movie number (by adding 1 to the index for a user-friendly format) and the corresponding movie name.

One possible pitfall in this code is the inclusion of periods at the end of each movie title in the list, which might be unnecessary when displaying the titles in a formatted output. This could lead to inconsistent formatting, especially if further modifications are made to the list in the future. A better approach might be to store the titles without the trailing period and manage the formatting during output. Additionally, if the list were to be modified or expanded, maintaining clear and consistent formatting would be essential for readability and user experience. Overall, the current implementation effectively demonstrates how to use `enumerate` for indexed iteration over a list.