**Store Management System**

**Project Description**

**1. Aim of the Project:**

The aim of the project is to create a comprehensive and efficient inventory and sales management system for a stationary shop. This system is designed to facilitate various operational tasks, enhance customer service, and provide insights into sales performance. The primary objectives include:

* **Efficient Inventory Management:**
* Track and manage stock levels of stationary products to ensure availability without overstocking.
* Allow easy addition, removal, and updating of products in the inventory.
* **Streamlined Sales Tracking:**
* Record sales transactions accurately to maintain a clear record of items sold and their prices.
* Provide insights into sales trends and performance over time.
* **User-Friendly Interface:**
* Develop an intuitive interface for shopkeepers to perform tasks such as adding new items, updating prices, and generating bills.
* Enhance customer experience by providing clear and detailed bills.
* **Operational Efficiency:**
* Automate processes such as bill generation and sales history recording to reduce manual effort and errors.
* Optimize decision-making through comprehensive sales reports and analytics.
* **Enhanced Customer Service:**
* Provide transparency and clarity in product pricing through detailed bill generation.
* Offer discounts on products to attract and retain customers, managed through the system.
* **Centralized Management:**
* Maintain a centralized platform where all inventory and sales-related operations are managed.
* Ensure data integrity and security to protect sensitive customer and business information.

By achieving these objectives, the project aims to modernize and optimize the stationary shop's operations, providing a robust system that supports growth, efficiency, and customer satisfaction.

**2. Business Problem or Problem Statement:**

The business problem or problem statement that the `Store` class and its associated methods aim to address is related to managing inventory and sales operations effectively within a stationary shop. Here's a detailed breakdown of the problem statement:

Business Problem:

Inefficient Inventory Management:

- Current Issue: The stationary shop lacks a systematic approach to manage its inventory, resulting in challenges such as overstocking or stockouts.

- Impact: This leads to suboptimal utilization of resources and potentially lost sales opportunities.

Manual and Error-Prone Sales Tracking:

- Current Issue: Sales transactions are recorded manually, which is time-consuming and prone to errors.

- Impact: Inaccurate records can lead to discrepancies in stock levels and financial reporting, affecting decision-making.

Lack of Customer Engagement and Transparency:

- Current Issue: Customers do not have a clear understanding of product pricing and discounts.

- Impact: This can lead to dissatisfaction and reduced customer retention.

Solution Requirements:

1. Comprehensive Inventory Management:

- Implement a system to track stock levels, add new products, update prices, and remove discontinued products efficiently.

2. Automated Sales Tracking:

- Develop mechanisms to automate the recording of sales transactions, including generating bills and maintaining a sales history.

3. Enhanced Customer Experience:

- Provide a user-friendly interface for customers to view product prices, apply discounts, and receive detailed bills.

4. Operational Efficiency:

- Streamline administrative tasks such as applying discounts, updating prices, and managing sales history to improve overall efficiency.

5. Data Security and Integrity:

- Ensure robust data security measures to protect sensitive customer information and maintain data integrity.

Project Goals:

- Develop a `Store` Class: Implement methods within the `Store` class to manage products, generate bills, track sales history, and handle customer interactions.

- User-Friendly Interface: Create a menu-driven interface that allows shopkeepers to perform tasks such as displaying products, applying discounts, updating prices, and viewing sales history easily.

- Automation and Accuracy: Automate routine tasks like bill generation and sales history recording to reduce manual effort and minimize errors.

- Customer Satisfaction: Enhance customer satisfaction by providing transparent pricing, discounts, and detailed bills.

Expected Outcomes:

- Improved Inventory Control: Ensure optimal stock levels by tracking inventory accurately and efficiently.

- Enhanced Operational Efficiency: Streamline operations through automation, reducing administrative burden and enhancing productivity.

- Better Decision-Making: Provide insights into sales trends and performance metrics to support informed decision-making.

By addressing these aspects, the `Store` class aims to transform the stationary shop's operations, making them more efficient, customer-focused, and conducive to growth and profitability.

**3. Project Description:**

**Goals:**

* **Inventory Management:**
* Add Product: Allow adding new products with their prices to the inventory.
* Display Products: Display all available products with their respective prices.
* Update Product Price: Enable updating prices of existing products.
* Remove Product: Support removing products that are no longer available.
* **Sales and Billing:**
* Generate Bill: Generate a detailed bill for a given shopping cart, including product quantities, individual prices, and total amount.
* View Sales History: Display a history of all sales transactions, including products sold and total amounts.
* **Customer Interaction:**
* Apply Discount: Provide functionality to apply discounts to products.
* Search Product: Allow searching for a product by its name and displaying its price.
* **Operational Efficiency:**
* Automation: Automate repetitive tasks such as bill generation and sales history recording to improve operational efficiency.
* User-Friendly Interface: Implement a user-friendly menu-driven interface for shopkeepers to perform operations easily and accurately.

**Features:**

* Add Product: Add new products to the inventory along with their prices.
* Display Products: Show all available products with their respective prices.
* Update Product Price: Modify the price of any existing product.
* Remove Product: Delete a product from the inventory.
* Generate Bill: Calculate and display a bill based on the products and quantities selected by the customer.
* View Sales History: Review a record of all past sales transactions.
* Apply Discount: Apply a discount percentage to a specific product's price.
* Search Product: Find and display a product's price by entering its name.
* Data Integrity: Ensure accurate tracking of inventory and sales to maintain data integrity and operational transparency.

**Expected Outcomes:**

* Improved Inventory Control: Maintain optimal stock levels and avoid overstocking or stockouts.
* Enhanced Customer Experience: Provide transparent pricing, accurate billing, and discounts to improve customer satisfaction.
* Operational Efficiency: Automate routine tasks to save time and reduce errors.
* Business Insights: Gain insights into sales trends and popular products through sales history analysis.

**4. Functionalities Implemented:**

* Add Product (add\_product method):
* Allows adding a new product to the inventory with a specified price.
* Products are stored in a dictionary (self.products) where the key is the product name and the value is its price.
* Display Products (display\_products method):
* Prints a list of all available products along with their prices.
* Iterates through the self.products dictionary and displays each product and its corresponding price.
* Generate Bill (generate\_bill method):
* Computes and prints a bill for a given shopping cart (shopping\_cart).
* Calculates the total price for each product in the shopping cart based on its quantity and displays detailed information including product, quantity, unit price, and total price.
* Computes the total amount to be paid (total\_amount) and displays it at the end of the bill.
* Take Quantity Input (take\_quantity\_input method):
* Prompts the user to enter quantities for each product in the inventory.
* Validates the input to ensure that quantities are integers and not negative.
* Returns a dictionary (shopping\_cart) where keys are product names and values are quantities.
* Update Product Price (update\_product\_price method):
* Allows updating the price of an existing product.
* Checks if the product exists in the inventory (self.products) and updates its price if found.
* Prints a confirmation message upon successful update.
* Remove Product (remove\_product method):
* Allows removing a product from the inventory.
* Checks if the product exists in the inventory (self.products) and removes it if found.
* Prints a confirmation message upon successful removal.
* Search Product (search\_product method):
* Allows searching for a product by its name and displays its price.
* Checks if the product exists in the inventory (self.products) and prints its price if found.
* Prints a message if the product is not found in the inventory.
* Add to Sales History (add\_to\_sales\_history method):
* Adds a sales transaction to the sales history (self.sales\_history).
* Each transaction is stored as a dictionary with keys 'cart' (containing the shopping cart) and 'total\_amount' (containing the total amount of the transaction).
* Calculate Total Bill (calculate\_total\_bill method):
* Computes the total amount to be paid for a given shopping cart.
* Iterates through the shopping cart and calculates the total amount by multiplying each product's price by its quantity.
* Returns the total amount.
* View Sales History (view\_sales\_history method):
* Displays the complete sales history stored in self.sales\_history.
* Iterates through each transaction and prints the details including products sold, quantities, and total amount.
* Apply Discount (apply\_discount method):
* Allows applying a discount percentage to a product's price.
* Checks if the product exists in the inventory (self.products) and calculates the discounted price.
* Updates the product's price to the discounted price and prints a confirmation message.
* Prints a message if the product is not found in the inventory.
* Main Menu (main method):
* Implements the main menu loop for interacting with the store functionalities.
* Displays a menu with options to perform actions such as displaying products, applying discounts, updating prices, etc.
* Continues to loop until the user chooses to exit ('9').

**5. Code Implementation:**

\_\_init\_\_(self)

* Initializes the Store object with empty dictionaries for products and sales\_history.

add\_product(self, product\_name, price)

* Adds a new product to the products dictionary with the given product\_name as key and price as value.

display\_products(self)

* Prints out all products currently in the store along with their prices.

generate\_bill(self, shopping\_cart)

* Generates and prints a bill based on the items in shopping\_cart, calculating total prices and displaying them.

take\_quantity\_input(self)

* Prompts the user to input quantities for each product in the store and returns a dictionary (shopping\_cart) mapping each product to its quantity.

update\_product\_price(self, product\_name, new\_price)

* Updates the price of a product specified by product\_name to new\_price.

remove\_product(self, product\_name)

* Removes a product from the store specified by product\_name.

search\_product(self, product\_name)

* Searches for a product specified by product\_name and prints its price if found.

add\_to\_sales\_history(self, shopping\_cart, total\_amount)

* Adds a transaction to sales\_history, storing the items in shopping\_cart and the total\_amount of the transaction.

calculate\_total\_bill(self, shopping\_cart)

* Calculates the total amount to be paid for the items in shopping\_cart, based on their quantities and prices.

view\_sales\_history(self)

* Prints out the entire sales history, including details of each transaction stored in sales\_history.

apply\_discount(self, product\_name, discount\_percent)

* Applies a discount (specified as discount\_percent) to the price of a product specified by product\_name.

main(self)

* The main menu-driven interface where users interact with the store functionalities. It allows displaying products, applying discounts, updating product prices, removing products, searching for products, adding transactions to sales history, generating bills, viewing sales history, and exiting the program.

**Example Usage:**

python

Copy code

if \_\_name\_\_ == "\_\_main\_\_":

store = Store()

store.main()

**Example Run-through:**

* Adding Products: The main method initializes the store with several products and prices.
* Menu Options: Users can choose from various options like displaying products, applying discounts, updating prices, etc.
* Interaction: Users input data such as product names, quantities, new prices, discount percentages, etc., to perform operations

.

* Output: The program provides feedback on each operation, displaying results like updated prices, bills, sales history, etc.

**Improvements and Considerations:**

* Error Handling: Enhance input validation and error handling for edge cases (e.g., non-numeric inputs).
* Data Persistence: Consider saving and loading data from a file/database to maintain state between program runs

.

* User Interface: For a graphical user interface (GUI), consider using libraries like Tkinter or PyQt.

**6.Class Overview:**

* Initialization (\_\_init\_\_):
* Initializes products as an empty dictionary to store product names and prices

.

* Initializes sales\_history as an empty list to store past transactions.
* Adding a Product (add\_product):
* Adds a new product to the products dictionary with its name as the key and its price as the value.
* Displaying Products (display\_products):
* Prints a formatted list of all products currently stored in the products dictionary.
* Generating a Bill (generate\_bill):
* Calculates and displays a bill based on the shopping\_cart dictionary provided.
* Shows each product, its quantity, unit price, and total price, and computes the total amount to be paid.
* Taking Quantity Input (take\_quantity\_input):
* Prompts the user to enter quantities for each product currently in the store.
* Validates input to ensure quantities are positive integers and updates the shopping\_cart accordingly.
* Updating Product Price (update\_product\_price):
* Updates the price of a specified product in the products dictionary to a new price.
* Removing a Product (remove\_product):
* Removes a specified product from the products dictionary if it exists.
* Searching for a Product (search\_product):
* Searches for a product in the products dictionary by name and prints its current price if found.
* Adding to Sales History (add\_to\_sales\_history):
* Adds a new entry to the sales\_history list containing a cart (dictionary of products and quantities) and total\_amount (total price of the transaction).
* Calculating Total Bill (calculate\_total\_bill):
* Computes the total amount to be paid based on the products and quantities in the provided shopping\_cart.
* Viewing Sales History (view\_sales\_history):
* Prints a formatted list of all past sales transactions stored in sales\_history, including details of each sale and the total amount.
* Applying Discount (apply\_discount):
* Applies a discount (specified as a percentage) to the price of a specified product in the products dictionary.
* Main Program (main):
* Initializes the store with a predefined set of products and enters a loop where users can interact with the store through a menu-driven interface.
* Allows users to display products, apply discounts, update prices, remove products, search for products, add transactions to sales history, generate bills, view sales history, and exit the program.

**Running the Program:**

* Initialization:
* Upon running the program, the Store class initializes with a predefined set of products (e.g., Apple, Banana, Orange, etc.).
* Menu Interface:
* Users are presented with a menu (Main Menu) where they can choose various options (from 1 to 9) to perform different operations on the store's inventory and sales history.
* User Interaction:
* Users input their choice and provide additional information such as product names, quantities, prices, discount percentages, etc., depending on the operation selected.
* Output:
* The program provides immediate feedback for each operation performed, displaying results such as updated prices, generated bills, sales history entries, etc.
* Exiting the Program:
* Users can choose to exit the program by selecting option 9 from the menu, which terminates the loop and ends the program execution.

Example Usage:

Here's an example of how a user might interact with your Store class through the menu interface:

mathematica

Copy code

===== Main Menu =====

1. Display Products

2. Apply Discount

3. Update Product Price

4. Remove Product

5. Search Product

6. Add to Sales History

7. Generate Bill

8. View Sales History

9. Exit

Enter your choice (1-9): 1

* Choosing option 1 would display all available products and their prices.

Available Products:

-------------------

Apple: 226

Banana: 37

Orange: 169

Cherry: 220

Avocado: 242

Kiwi: 90

Grapes: 80

Mango: 130

Plum: 140

Papaya: 60

-------------------

**8. Conclusion:**

Your Store class provides a comprehensive framework for managing an inventory, handling transactions, and maintaining a sales history. It's structured to support interaction through a command-line interface, making it suitable for basic store management tasks. Further enhancements could include additional error handling, input validation, or integration with external data storage for more robust functionality.