Assignment: Python Programming for Innovation

Name: A.Dhatrika

Register Number:192311429

Department: CSE

Problem 2: Inventory Management System Optimization
Scenario:
You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.
Tasks:

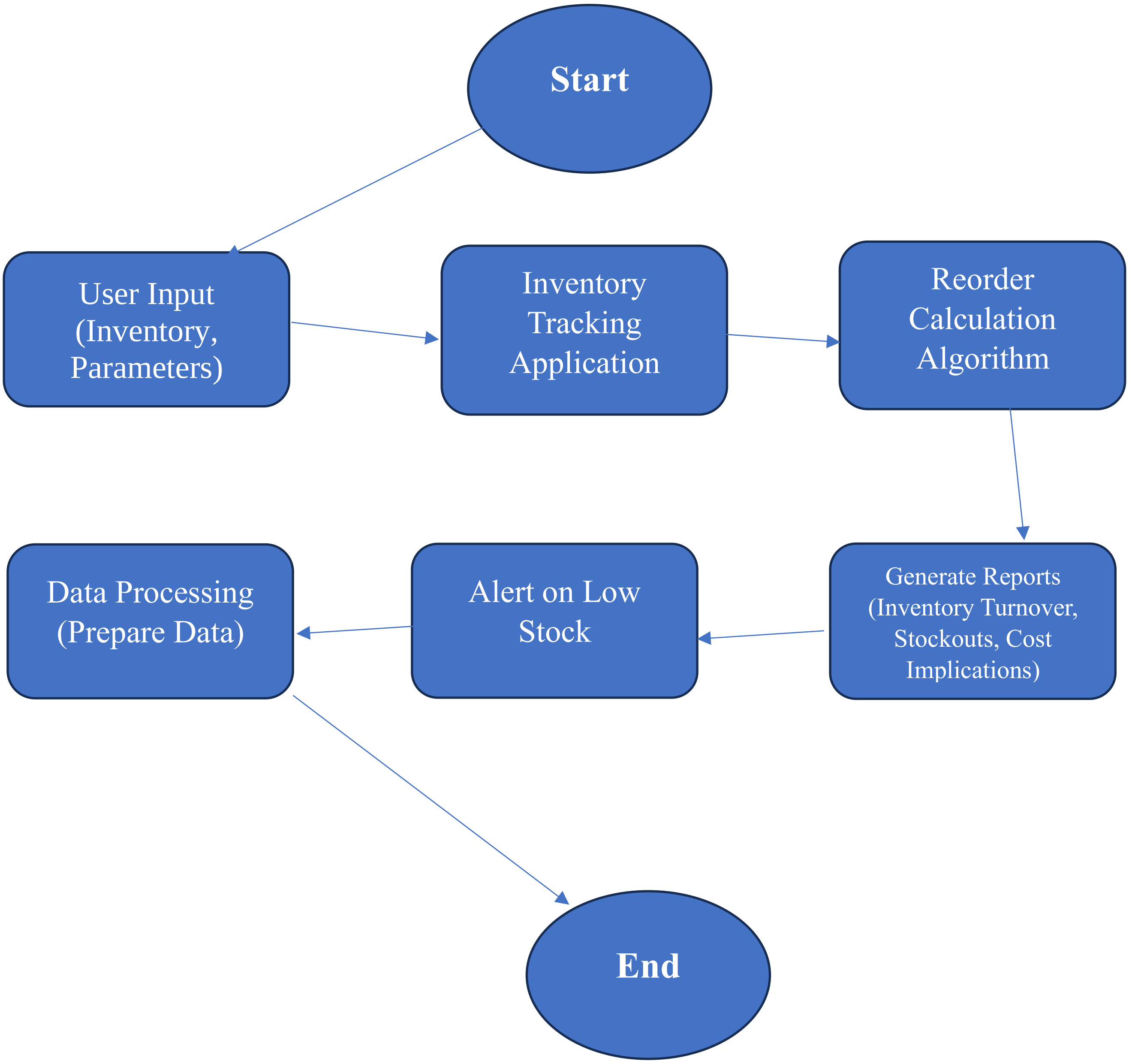
1. Model the inventory system: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. Implement an inventory tracking application: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. Optimize inventory ordering: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

Deliverables:

- Data Flow Diagram: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
- Pseudocode and Implementation: Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
- Documentation: Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
- User Interface: Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
- Assumptions and Improvements: Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency

and accuracy.

Solution:
Inventory Management System Optimization
1.Data Flow Diagram



Implementation:

INVENTORY TRACKING APPLICATION:-

```
import statistics
import math


class Product:
    def __init__(self, product_id, name, initial_stock, reorder_point, reorder_quantity):
        self.product_id = product_id
        self.name = name
        self.stock = initial_stock
        self.reorder_point = reorder_point
        self.reorder_quantity = reorder_quantity

    def update_stock(self, adjustment):
        self.stock += adjustment

class InventoryManagementSystem:
    def __init__(self):
        self.products = {}

    def add_product(self, product_id, name, initial_stock, reorder_point, reorder_quantity):
        if product_id not in self.products:
            self.products[product_id] = Product(product_id, name, initial_stock, reorder_point,
reorder_quantity)
        else:
            print(f"Product with ID {product_id} already exists.")

    def track_inventory(self, product_id, adjustment):
        if product_id in self.products:
            self.products[product_id].update_stock(adjustment)
            if self.products[product_id].stock < self.products[product_id].reorder_point:
                self.generate_reorder_alert(product_id)
```

```
        else:
            print(f"Product with ID {product_id} does not exist.")

    def generate_reorder_alert(self, product_id):
        print(f"Alert: Product {self.products[product_id].name} is below reorder point. Current stock: {self.products[product_id].stock}")

    def get_product_stock(self, product_id):
        if product_id in self.products:
            return self.products[product_id].stock
        else:
            return None

# Example usage:
ims = InventoryManagementSystem()
ims.add_product(1, "Keyboard", 50, 10, 50)
ims.add_product(2, "Mouse", 75, 15, 30)

# Simulate inventory adjustments
ims.track_inventory(1, -5)  # Sold 5 keyboards
ims.track_inventory(2, -10)  # Sold 10 mice

# Check current stock levels
print("Current stock levels:")
print(f"Keyboard: {ims.get_product_stock(1)}")
print(f"Mouse: {ims.get_product_stock(2)}")

#### Reorder Calculation Algorithm (Example)
class ReorderOptimizer:
    def __init__(self, demand_history, lead_time):
        self.demand_history = demand_history  # List of historical demand data
        self.lead_time = lead_time  # Lead time in days for ordering
```

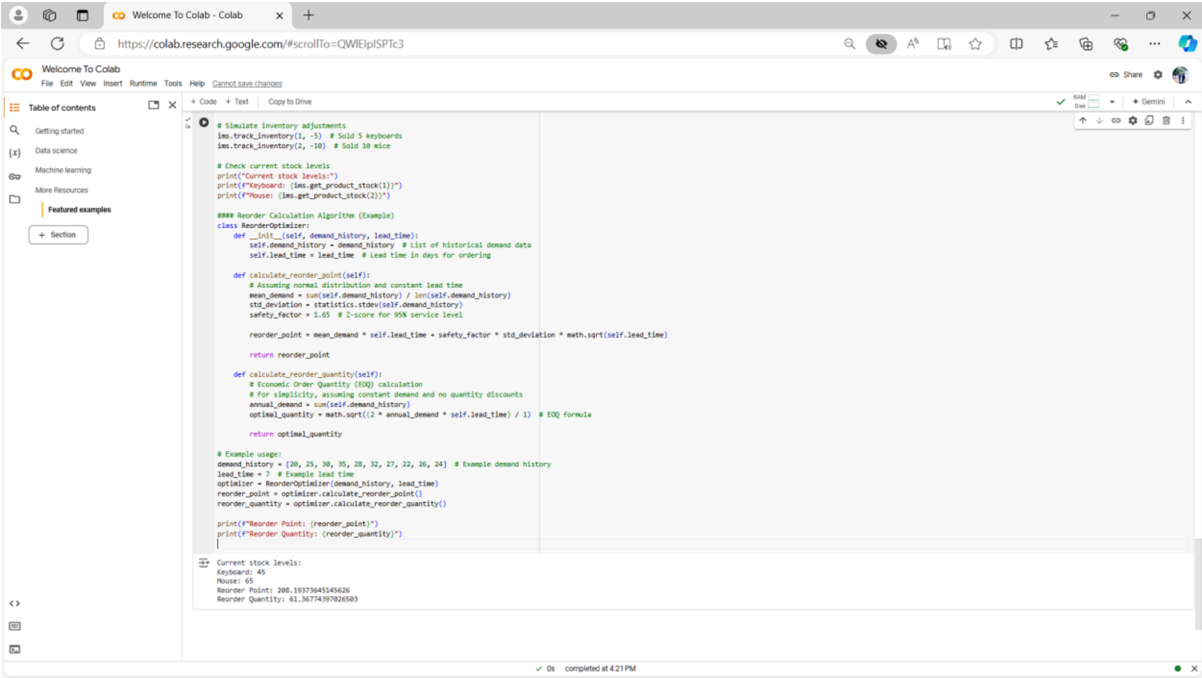
```
    def calculate_reorder_point(self):
        # Assuming normal distribution and constant lead time
        mean_demand = sum(self.demand_history) / len(self.demand_history)
        std_deviation = statistics.stdev(self.demand_history)
        safety_factor = 1.65  # Z-score for 95% service level

        reorder_point = mean_demand * self.lead_time + safety_factor * std_deviation *
math.sqrt(self.lead_time)

        return reorder_point

    def calculate_reorder_quantity(self):
        # Economic Order Quantity (EOQ) calculation
        # For simplicity, assuming constant demand and no quantity discounts
        annual_demand = sum(self.demand_history)
        # EOQ formula
        optimal_quantity = math.sqrt((2 * annual_demand * self.lead_time) / 1)
        return optimal_quantity
demand_history = [20, 25, 30, 35, 28, 32, 27, 22, 26, 24]  # Example demand history
lead_time = 7  # Example lead time
optimizer = ReorderOptimizer(demand_history, lead_time)
reorder_point = optimizer.calculate_reorder_point()
reorder_quantity = optimizer.calculate_reorder_quantity()
print(f"Reorder Point: {reorder_point}")
print(f"Reorder Quantity: {reorder_quantity}")
```

User Input:



Documentation:

Algorithms for Reorder Optimization

1. 1. **Reorder Point Calculation**:
2.   - Uses historical demand data to calculate the average demand over a period and the standard deviation to estimate variability.
3.   - Incorporates a safety factor (e.g., Z-score for a desired service level) to determine the reorder point that minimizes stockout risk.
4. 2. **Reorder Quantity Calculation**:
5.   - Implements the Economic Order Quantity (EOQ) model to determine the optimal order quantity.
6.   - Assumes constant demand and lead time, aiming to minimize total ordering and holding costs.

Assumptions

7. - **Demand Patterns**: Assumes demand follows a normal distribution or can be approximated as such for reorder point calculations.

8. - **Lead Times**: Assumes lead times are constant and known, impacting the reorder point but not the EOQ calculations.

User Interface

9. A user-friendly interface can be developed using a GUI framework (e.g., Tkinter for desktop applications or Flask/Django for web applications). This interface would allow users to:
10.
11. - Input product IDs or names to view current stock levels and receive alerts on low stock.
12. - Generate reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
13. - Display recommended reorder quantities and points based on historical data and algorithms implemented.
14.

Improvements

15.
16. - **Dynamic Lead Time**: Incorporate variability in lead times to enhance accuracy in reorder point calculations.
17. - **Demand Forecasting**: Implement more sophisticated forecasting models (e.g., ARIMA, exponential smoothing) for demand predictions.
18. - **Integration with ERP Systems**: Integrate the inventory management system with enterprise resource planning (ERP) systems for seamless data flow and automation.
19. - **Supplier Collaboration**: Establish partnerships with suppliers to streamline ordering processes and reduce lead times.
20.
21. By addressing these improvements, the inventory management system can achieve higher efficiency in inventory turnover, reduce stockouts, and optimize ordering decisions to minimize costs associated with overstock situations.