**Assignment: Python Programming for GUI Development**

Name: B.Pavan Kumar Reddy

Register Number:192311382

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

**Start**

**End**

Reorder Calculation Algorithm

Inventory Tracking Application

User Input (Inventory, Parameters)

Generate Reports (Inventory Turnover, Stockouts, Cost Implications)

Alert on Low Stock

Data Processing (Prepare Data)

**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)

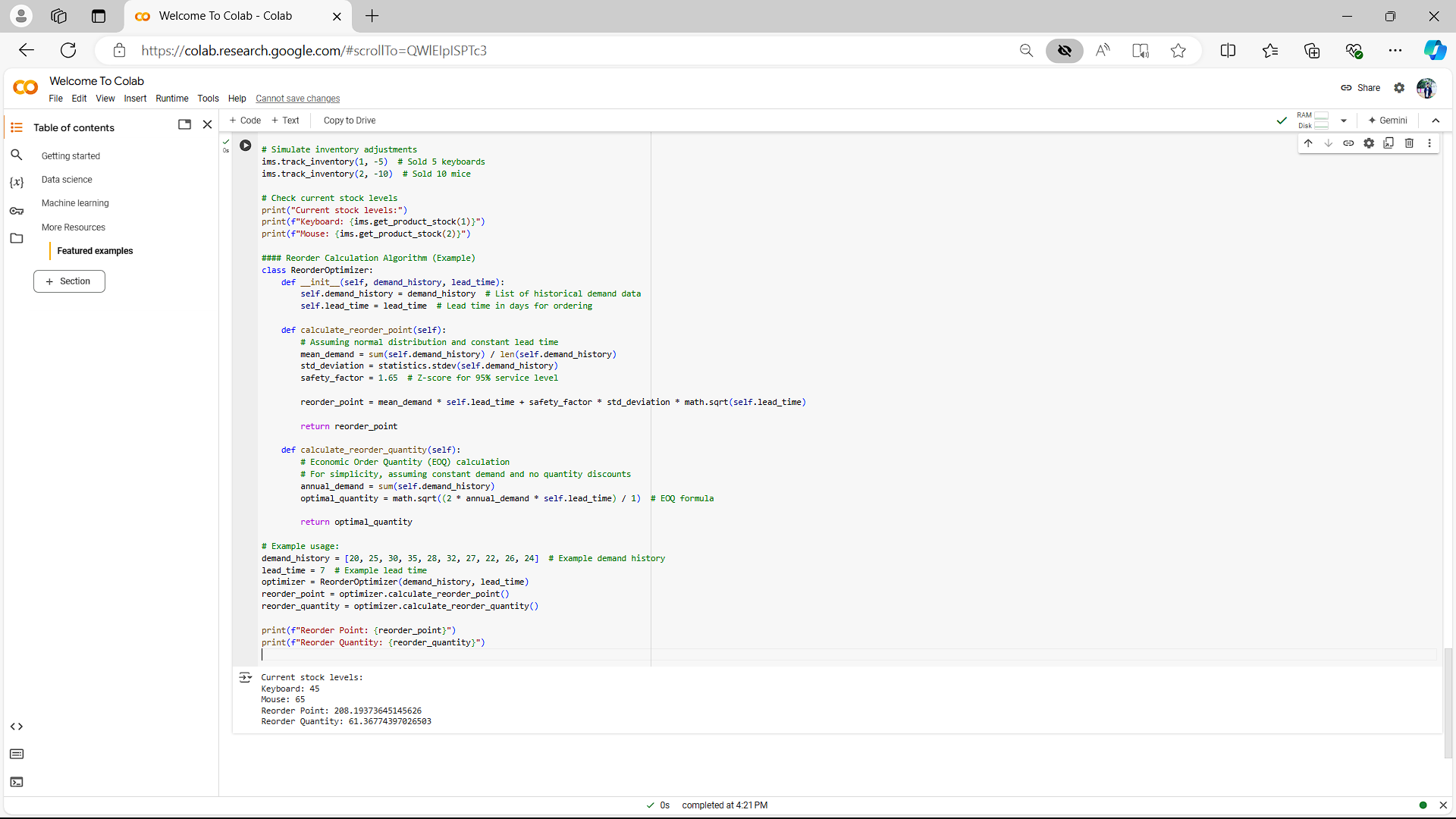
reorder\_point = optimizer.calculate\_reorder\_point()

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

* 1. - \*\*Demand Patterns\*\*: Assumes demand follows a normal distribution or can be approximated as such for reorder point calculations.
  2. - \*\*Lead Times\*\*: Assumes lead times are constant and known, impacting the reorder point but not the EOQ calculations.

User Interface

* 1. 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:
  2. - Input product IDs or names to view current stock levels and receive alerts on low stock.
  3. - Generate reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
  4. - Display recommended reorder quantities and points based on historical data and algorithms implemented.

Improvements

* 1. - \*\*Dynamic Lead Time\*\*: Incorporate variability in lead times to enhance accuracy in reorder point calculations.
  2. - \*\*Demand Forecasting\*\*: Implement more sophisticated forecasting models (e.g., ARIMA, exponential smoothing) for demand predictions.
  3. - \*\*Integration with ERP Systems\*\*: Integrate the inventory management system with enterprise resource planning (ERP) systems for seamless data flow and automation.
  4. - \*\*Supplier Collaboration\*\*: Establish partnerships with suppliers to streamline ordering processes and reduce lead times.
  5. 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.