CSC 1024

PROGRAMMING PRINCIPLES

TITLE OF REPORT:

A PYTHON-BASED INVENTORY MANAGEMENT SYSTEM

Prepared by:

Ding Yee Qing 23080732

Lee Hui Wen 23079692

Lim Shan En 23080690

Sin Shen Nee 23050024

Tee Yu Qing 23077787

Prepared for:

ASSOC. PROF. DR ANWAR P.P. ABDUL MAJEED

Table of Contents

Video Presentation Link: https://youtu.be/ONqOAEXQnjg

1.0 Explanation of System Design and Architecture

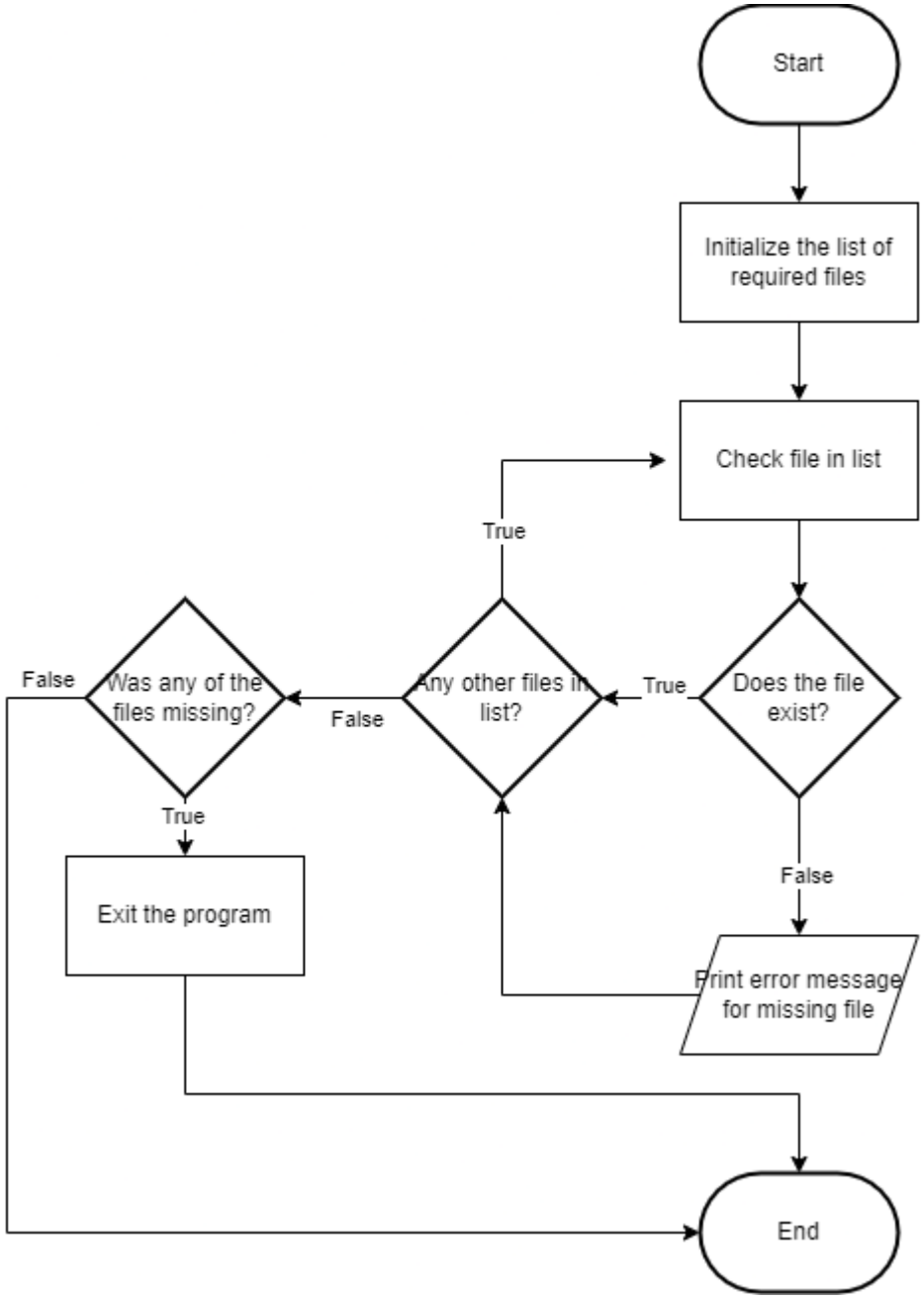
In this assignment, an Inventory Management System was created using Python to effectively and efficiently handle various inventory operations. Hence, this program has integrated various functionalities to facilitate the inventory management process, namely:

1. Check files function
2. Add a new product function
3. Update product details function
4. Add a new supplier function
5. Place an order function
6. View inventory function
7. Generate reports function
8. Main program function

1.1 Check Files

The check files function is used to ensure that all the files required in the program are presented in the working directory. First, the function begins by importing the necessary modules, including **os** for checking file existence, **sys** for terminating the program if necessary, and **datetime**. The **required_files** list contains the names of the required files for the program to run, which are **products.txt** file, **orders.txt** file, and **suppliers.txt** file. A for loop iterates through each file in the required_files list. The **os.path.exists(file)** function checks whether the file exists in the current directory. If any file is missing, the program displays an error message identifying which files are missing. To avoid the program from running with incomplete data, a second for loop is used to recheck the presence of all required files. If the file is missing, the program calls **sys.exit()** method to immediately terminate execution. This approach ensures that the program does not proceed without the necessary files, thereby preventing potential errors caused by missing data files.

Flowchart


Start
Initialize the list of required files
Check file in list
True
Does the file exist?
True
Any other files in list?
False
Was any of the files missing?
False
True
Exit the program
False
Print error message for missing file
End

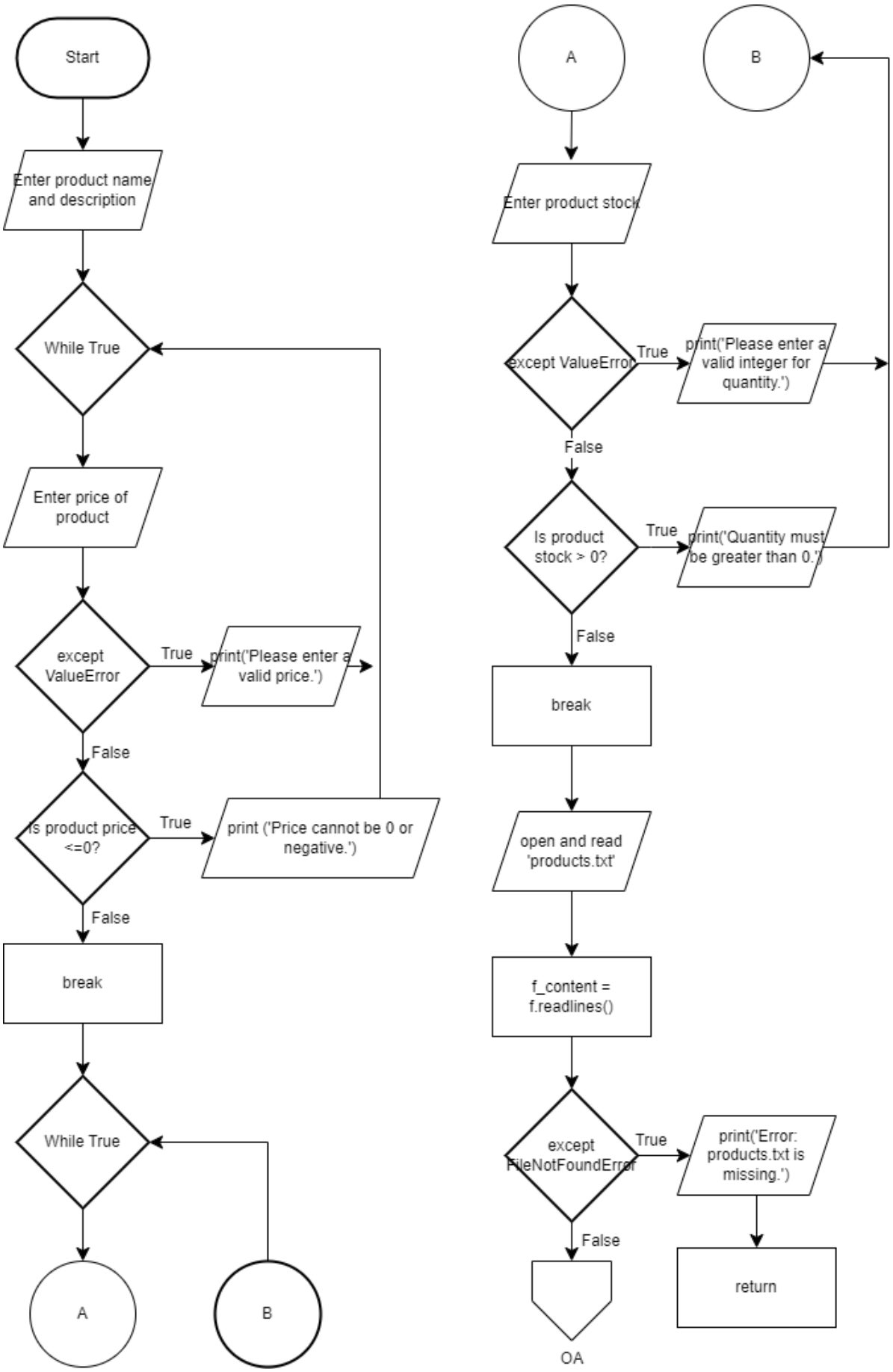
1.2 Add a New Product

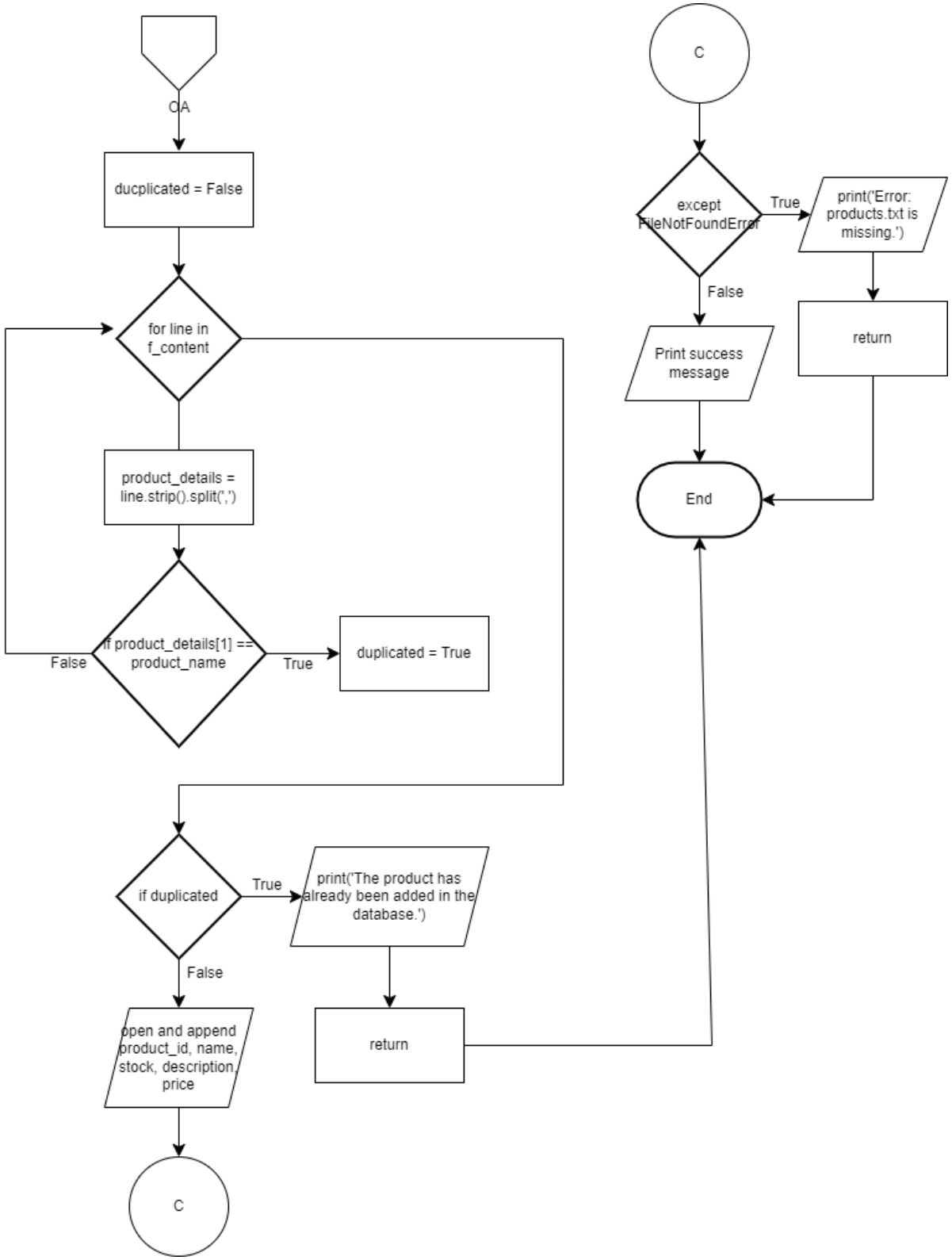
The add a new product function allows users to input a new product into the inventory. To uniquely identify products, the function starts by prompting the user to **input the product name** (product_name) and a **description** (product_description) as strings. Next, the user is prompted to enter the **product price** (product_price). A **while True** loop ensures the user enters a valid positive float value for the product price. Within the loop, a **try-except block** checks the input. If the user enters a number value less than or equal to zero, an error message is displayed. The **except ValueError** block catches exceptions where a non-numeric value is entered. The loop continues until a valid positive price is provided. The user is then prompted to input the **product stock quantity** (product_stock). Another while True loop is created to ensures the user enters a valid positive integer for the stock quantity. If the input is not an integer, or is less then or equal to zero, appropriate error messages are displayed using the if-else block and try-except block. After collecting valid inputs, the function attempts to open the **product.txt** file in read mode to check for existence again. If the file is missing, an error message is displayed, and the function exits early to avoid further processing.

For the duplication check, the function iterates through the lines of the file (f_content) to check if a product with the same name (product_name) already exists. The inputs have trailing and leading whitespaces removed using **strip()** method, and each line is split into components using **split(‘,’)** method. The second element (product_details[1]) is compared with the entered **product_name**. A duplicated flag is initialized as False, indicating that no duplicate product has been found initially. If a match is found, the duplicated flag is set to True, indicating that a product with the same name already exists in the database. The program displays the product is already in the system database and exits the function early. If no duplicate is found, the program generates a **new product ID** by opening the **products.txt** file in append mode. The product ID is generated based on the existing IDs in the file. It extracts the numeric portion of the existing product IDs by skipping the first character (P), identifies the largest number, and incremented by 1 to create a unique ID for the new product. The new product ID follows the format Pxxx, where xxx is a three-digit number (e.g., P001). Finally, the function appends the new product details, including ID, name, stock quantity, description and price (formatted to two decimal places) to the **products.txt** file. A success message confirms the addition.

Thus, this adds a new product function ensures that users enter a valid product name, description, price, stock quantity and cannot enter duplicated products.

Flowchart


Start
Enter product name and description
While True
Enter price of product
except ValueError
True
print('Please enter a valid price.')
False
Is product price <=0?
True
print ('Price cannot be 0 or negative.')
False
break
While True
A
B
A
B
Enter product stock
except ValueError
True
print('Please enter a valid integer for quantity.')
False
Is product stock > 0?
True
print('Quantity must be greater than 0.')
False
break
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
False
return
OA


CA
ducplicated = False
for line in f_content
product_details = line.strip().split(',')
if product_details[1] == product_name
False
True
duplicated = True
if duplicated
True
print('The product has already been added in the database.')
False
open and append product_id, name, stock, description, price
return
C
C
except FileNotFoundError
True
print('Error: products.txt is missing.')
False
Print success message
return
End

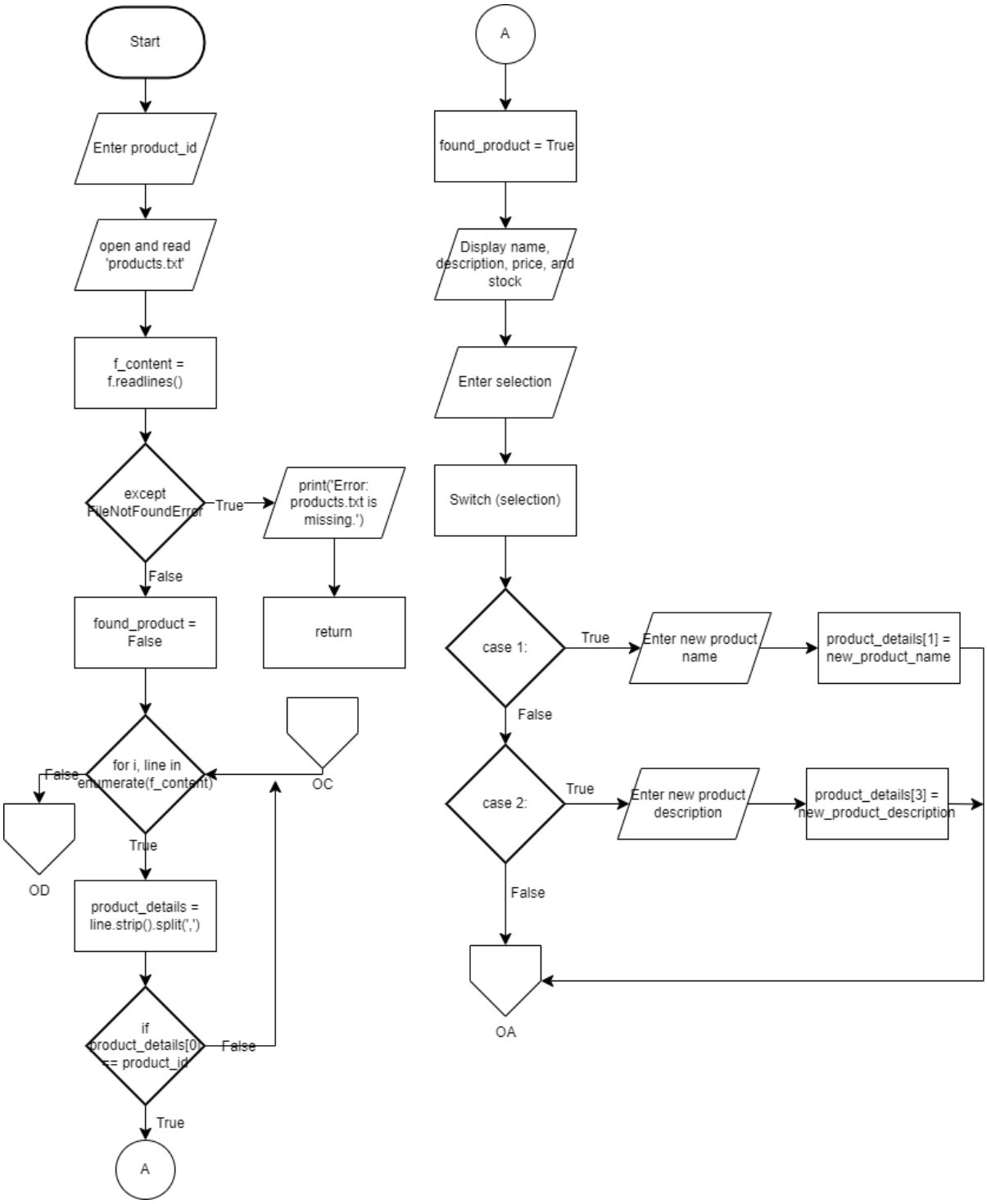
1.3 Update Product Details

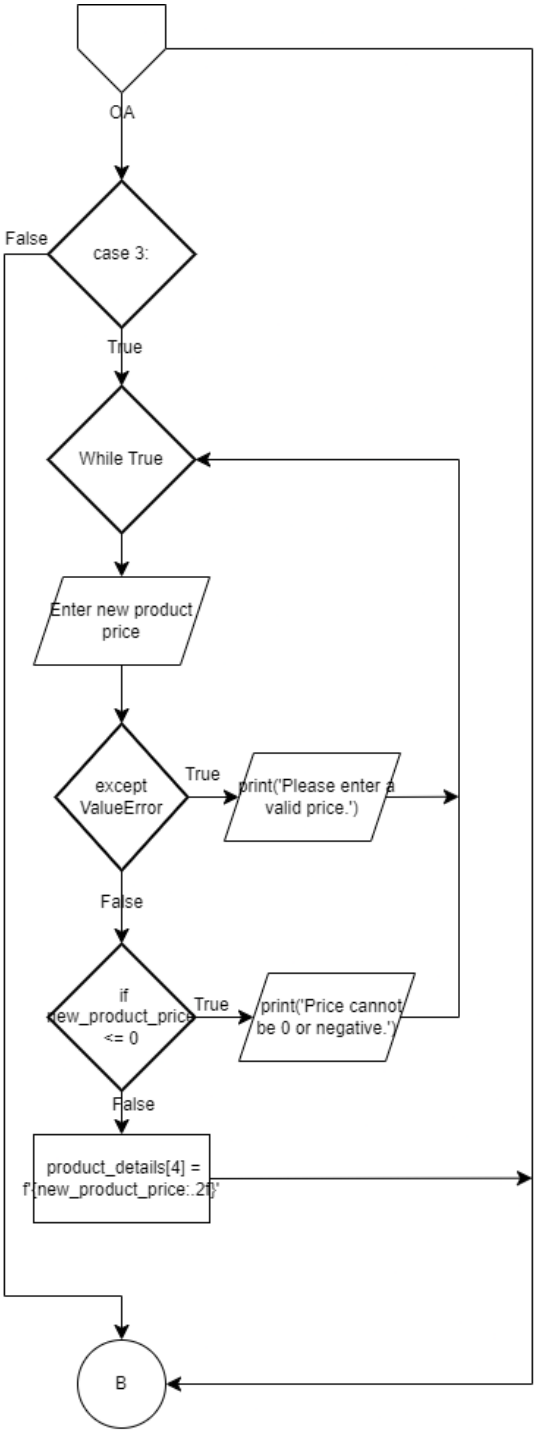
The update product details function allows users to input updates on the details of a product in the products.txt file. First, it prompts the user to enter the product ID (product_id) they want to update. The input is converted to uppercase to ensure consistency when comparing IDs. The function attempts to open the **product.txt** file in read mode, and its contents are read line by line and stored in a list of string for each line (f_content), where each line represents a product. If the product.txt file is missing, a FileNotFound is caught, an error message is displayed, and the function exits early to prevent further execution. To search a product, a flag **found_product** is initialized as False, to indicate whether the specified product is found. The function uses a for loop to iterate through each line in the file content (f_content) , where each line is stripped of any leading or trailing whitespace using the **strip()** method; then the line is split into a separate list (product_details) using the **split(',')** method. The product ID, which is the first element of the line (product_details[0]) is compared to the user-entered product ID (product_id). If a match is found, the **found_product** flag is set to True, and a menu with options to update: (1) **name**, (2) **description**, (3) **price** or (4) **stock** is displayed. The user is prompted to select which part of the product they want to update.

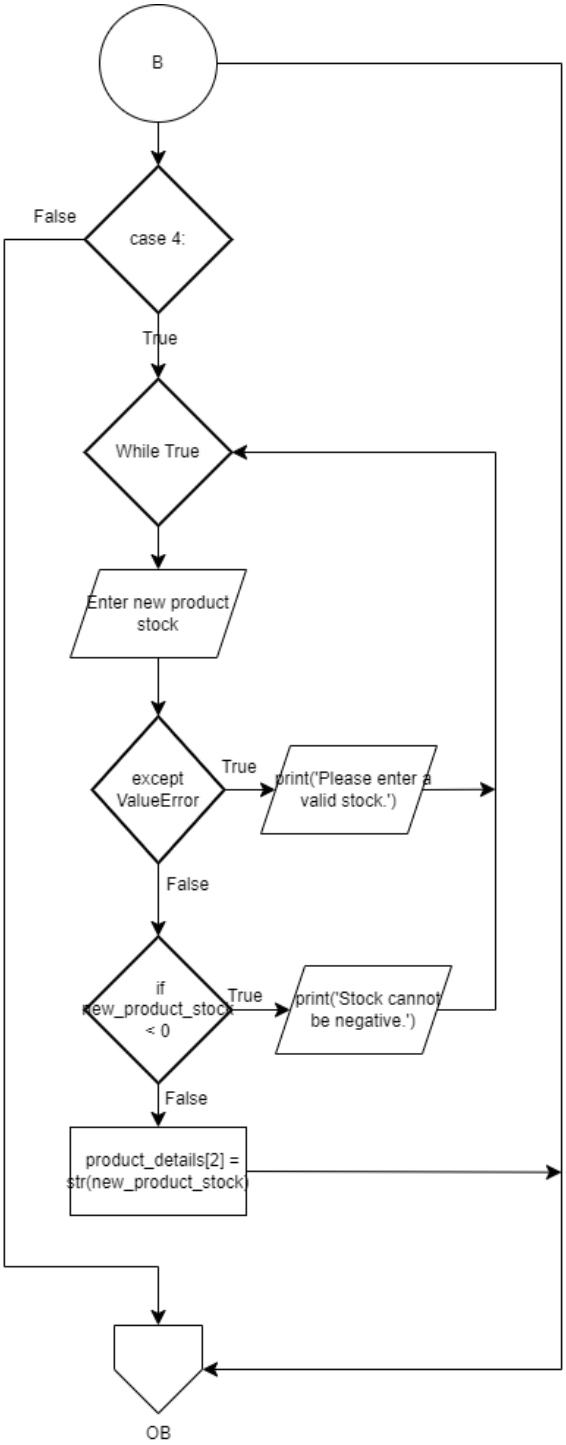
The menu is using a **match statement**. For case '1', the program prompts the user to enter the new product name. The second element is updated in the list of **product_details** (product_details[1] = new_product_name). For case '2', similar to case 1, the fourth element of the **product_details** list will be updated to a new product description (product_details[3] = new_product_description). For case '3', a while True loop and a try-except block ensure the user enters a valid positive float value for the new price. If the user enters a number value less than or equal to zero, or a non-numerical value, a specific error message is displayed and the user is continuously prompted until a valid input is provided. The price is then updated to two decimal places in the fifth element of the **product_details** list (product_details[3] = new_product_description). For case '4', a while True loop and try-except block also generated to ensure the user enters a valid positive integer for the new stock quantity. The third element of the product_detail list will be updated in string (product_details[2] = str(new_product_stock)). If the user enters an invalid option, an error message displayed, and the function exits early without making changes. The updated product details (product_details) are joined back into a string using **','.join(product_details)** and replace the corresponding line in **f_content** list, and a success message is displayed. If the product ID is not found in the file, the function displays an error message and exits early. If the product details are successfully updated, the modified content (f_content) is written back to the **products.txt** file using write mode. To handle potential issues, if the file is missing unexpectedly during the rewrite process, an error message is displayed, and the function exits.

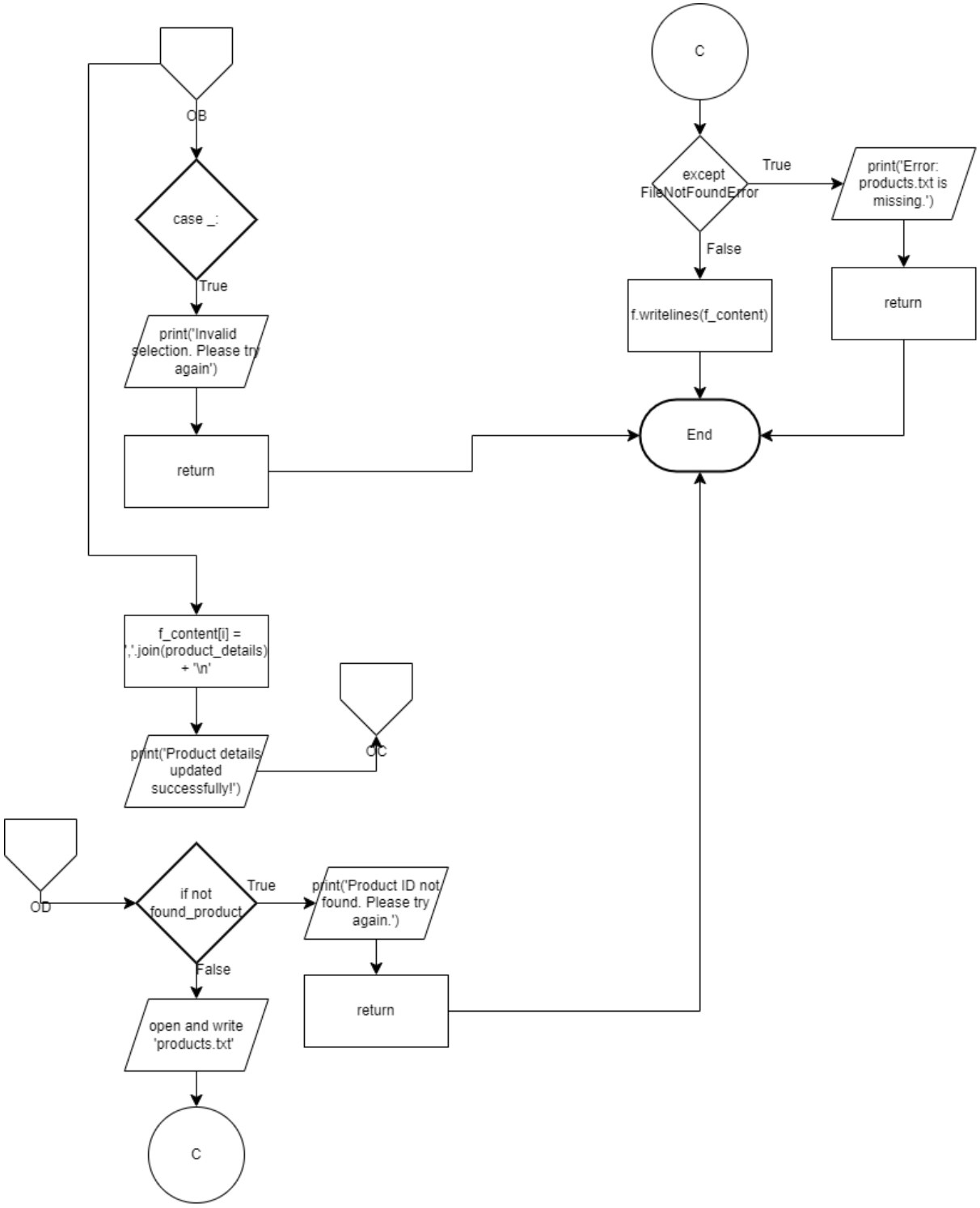
Overall, the update product details function allows users to update one of four fields ---- the product's name, description, price, or stock quantity in a single operation.

Flowchart




OA
case 3:
False
True
While True
Enter new product price
except ValueError
True
print('Please enter a valid price.')
False
if new_product_price <= 0
True
print('Price cannot be 0 or negative.')
False
product_details[4] = f'{new_product_price:.2f}'
B


B
case 4:
False
True
While True
Enter new product stock
except ValueError
True
print('Please enter a valid stock.')
False
if new_product_stock < 0
True
print('Stock cannot be negative.')
False
product_details[2] = str(new_product_stock)
OB


OB
case _:
True
print('Invalid selection. Please try again')
return
C
except FileNotFoundError
True
print('Error: products.txt is missing.')
False
f.writelines(f_content)
return
End
f_content[i] = ','.join(product_details) + '\n'
print('Product details updated successfully!')
CC
OD
if not found_product
True
print('Product ID not found. Please try again.')
False
return
open and write 'products.txt'
C

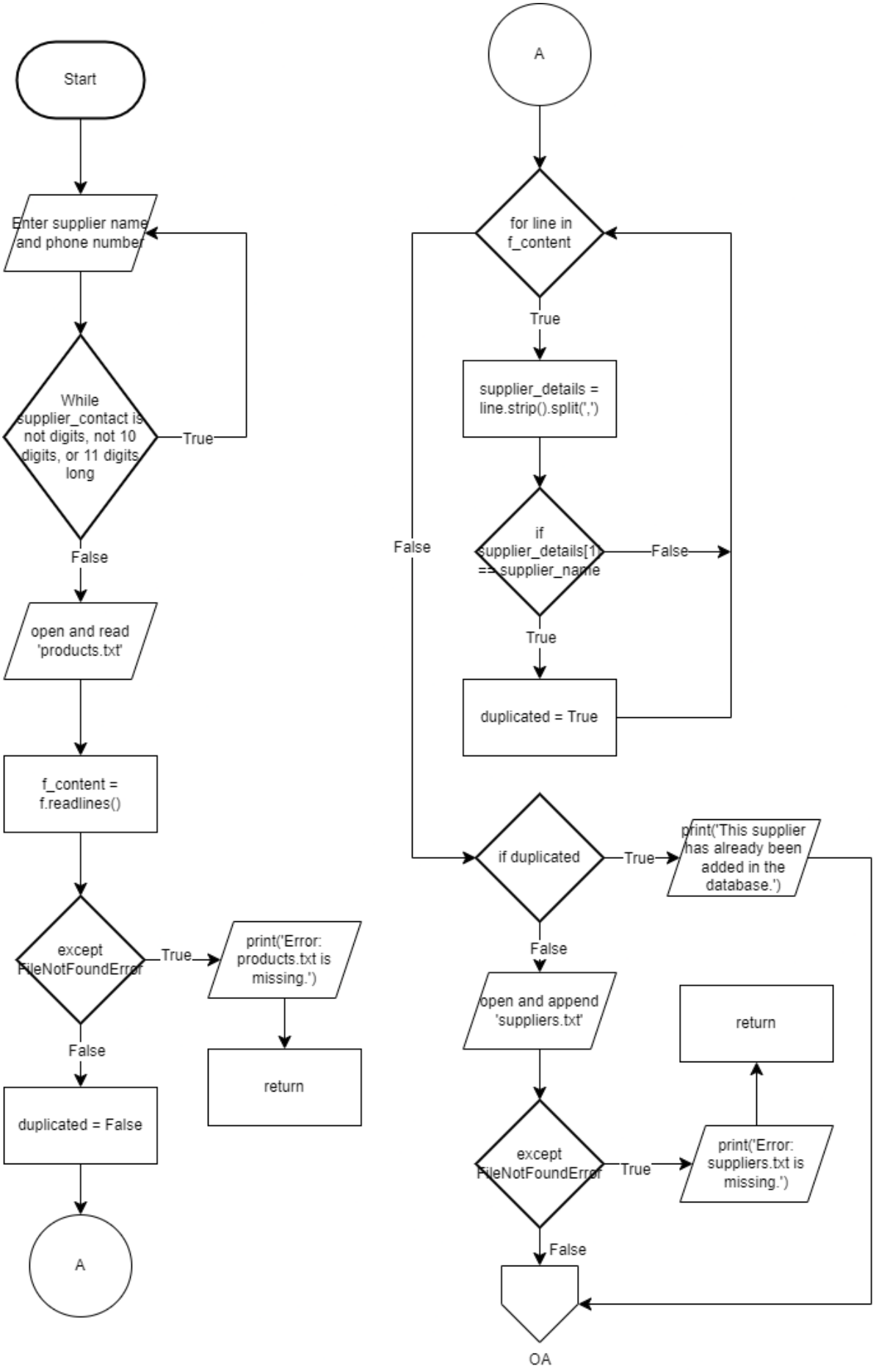
1.4 Add a New Supplier

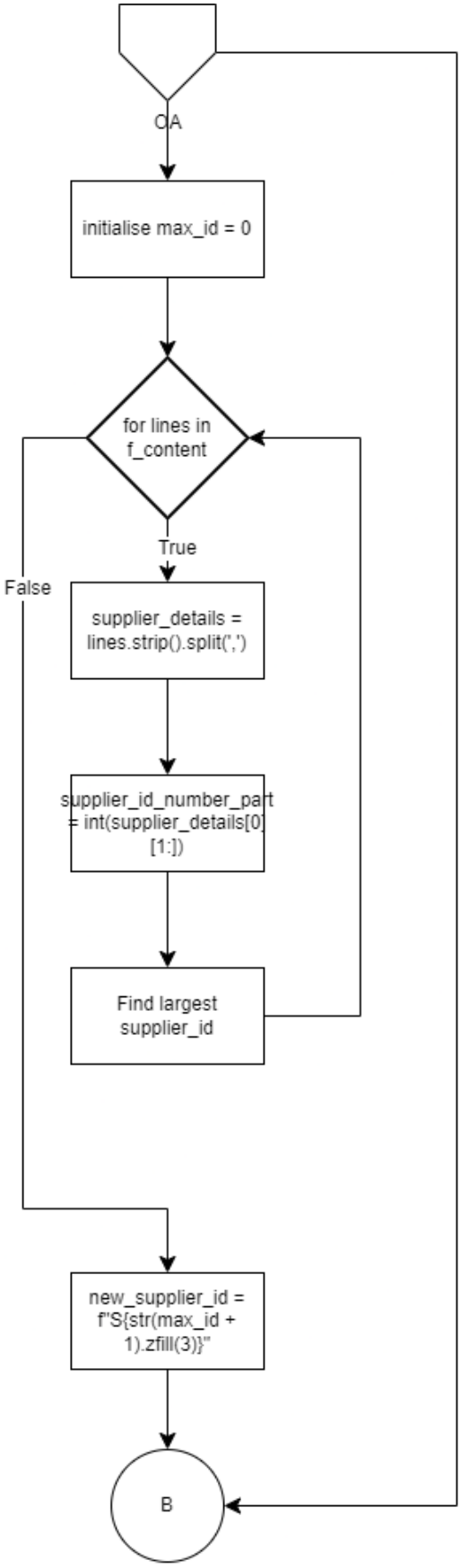
The add a new supplier function enables users to add a new supplier to the suppliers database system (suppliers.txt file). The function begins by prompting the user to input the **supplier's name** (supplier_name) and **contact number** (supplier_contact). The contact number is validated using a while loop to ensure it contains only numeric characters and has a length of 10 or 11 digits. If the input is invalid, the user is repeatedly prompted until a valid phone number is provided.

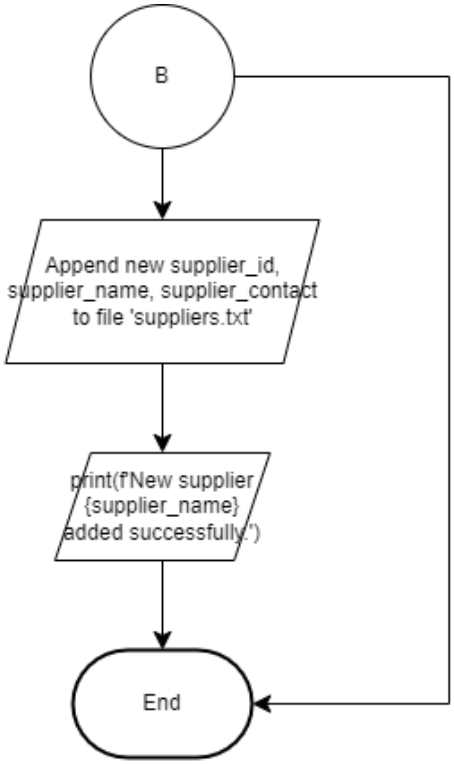
Next, the function attempts to open the supplier.txt file in read mode to check for duplicate supplier names. By using the try-except block, if they found is missing a FileNotFoundError is caught, an error message is displayed, and the function exits early. The contents of the file are then read line by line and stored in the list of strings (f_content). A duplicated flag is initialized as False to track whether the supplier's name already exists. Each line is stripped of leading and trailing whitespace using the **strip()** method and split into a list of supplier details using the **split(',')** method. The if-else block is used to compare the new input supplier name with the second element in each line (supplier_details[1] == supplier_name). If a match is found, the duplicated flag is set to True, and the program displays a message indicating that the supplier is already in the database, exiting the function early. If no duplicates are found, the function generates a unique new supplier ID by opening the **supplier.txt** file in append mode. A new supplier ID is generated based on the existing supplier IDs in the file. It extracts the numeric portion of the existing supplier IDs by skipping the first character (S), identifies the largest number, and increments it by 1 to create a new ID. The new ID follows the format Sxxx, where xxx is the three-digit number (e.g., S001). The new supplier details, including the ID, name, and contact number are appended to the **supplier.txt** file. A success message is displayed confirming the addition of the supplier. If the file is missing during the appending process, a second FileNotFoundError is caught, an error message is displayed, and the function exits immediately.

Hence, this adds supplier function ensures that users enter a valid supplier name, and properly formatted contact number, and prevents duplicate supplier entries.

Flowchart


Start
Enter supplier name and phone number
While supplier_contact is not digits, not 10 digits, or 11 digits long
True
False
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
return
False
duplicated = False
A
A
for line in f_content
True
supplier_details = line.strip().split(',')
if supplier_details[1] == supplier_name
False
True
duplicated = True
False
if duplicated
True
print('This supplier has already been added in the database.')
False
open and append 'suppliers.txt'
return
except FileNotFoundError
True
print('Error: suppliers.txt is missing.')
False
OA


OA
initialise max_id = 0
for lines in f_content
True
False
supplier_details = lines.strip().split(',')
supplier_id_number_part = int(supplier_details[0][1:])
Find largest supplier_id
new_supplier_id = f"S{str(max_id + 1).zfill(3)}"
B


B
Append new supplier_id, supplier_name, supplier_contact to file 'suppliers.txt'
print(f'New supplier {supplier_name} added successfully.')
End

1.5 Place an Order

The place an order function allows users to place an order from a supplier. The function starts by prompting the user to input their supplier ID (supplier_id) which is converted to uppercase to maintain consistency. Next, it checks the existence of **supplier.txt** file by opening the file in read mode. If the file is missing, a FileNotFoundError is caught, an error message is displayed, and the function exits early. Next, a **found_supplier** flag initialized as False to track whether the supplier already existed in the file. A for loop is created to iterate the contents of the file line by line to search for the supplier ID (supplier_id) entered by the user. Each line is stripped of leading and trailing whitespace using the **strip()** method and split into a list of supplier details using the **split(',')** method. In the if-else block, when 'supplier_details[0] == supplier_id', it indicates that a valid supplier is found and the **found_supplier** flag is set to True and proceed to next step. If the supplier is not found, an error message is displayed, and the function exits.

Since a valid supplier is found, the user is then prompted to input the product ID () they want to order. Same with the previous step, the function attempts to open the **products.txt** file in read mode to verify if the product exists. The except block ensures that if the **product.txt** file is missing, an error message is displayed, and the function exits. A **found_product** flag is initialized as False to track whether the specified product ID exists in the **product.txt** file. Also, an **order_status** flag is initialized as False to indicate whether the order was successfully processed based on stock availability. After that, the function iterates through the lines of **products.txt** file using a for loop to search for the product ID entered by the user. Each line represents a product details is stripped and split into a list. In the if-else block, when the 'product_details[0] == product_id', indicating that a valid product ID is found and the **product_found** flag is set to True. Next, a True while loop is used to ensure that the user enters a valid positive integer for order quantity. The user is prompted to enter a valid order quantity. The input is converted to an integer inside a try-except block to handle potential ValueError exceptions if the user enters a non-numeric input. The if-else block inside the try-except block ensures the quantity is greater than zero. If the user enters invalid data, specific error messages are displayed. If a valid quantity is entered, the product name (product_name = product_details[1]), product price (product_price = float(product_details[4])), and product stock (product_stock = int(product_details[2])) of the relevant product are extracted from the **product_details** list.

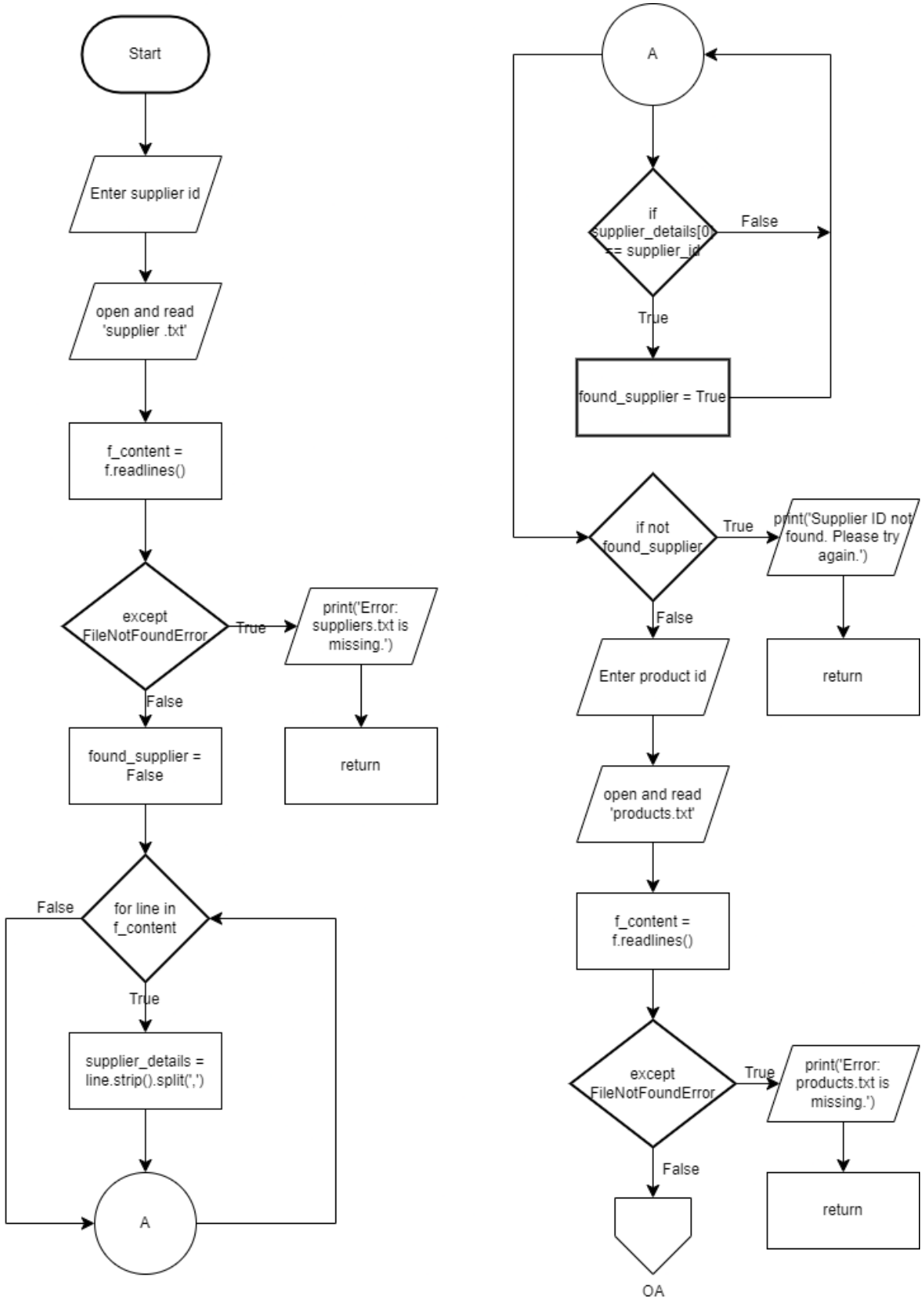
Now, the function checks whether the product chosen by the user is sufficient stock to fulfil the order. If the **product_stock** is greater than or equal to the **order_quantity**, the **order_status** flag is set to True. The total price is calculated as **order_quantity*product_price**; the stock quantity in the product details is updated to reflect

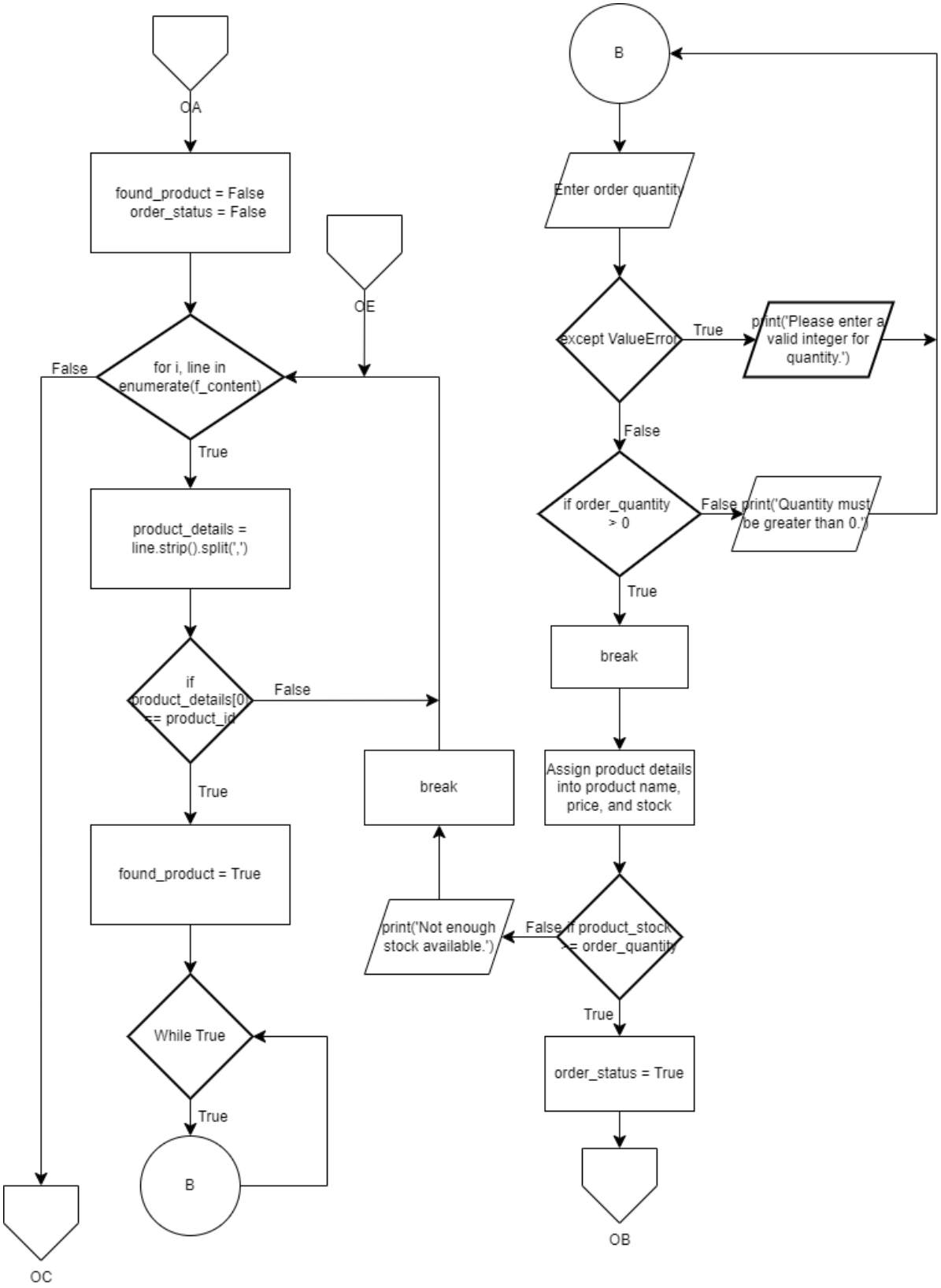
the reduced stock. The updated product details are converted back into a string using **','.join(product_details)**, and replaces the corresponding line in the **f_content** list. A success message is displayed, showing the ordered product name, quantity, and total ordered price. If there is not enough stock, an error message is displayed.

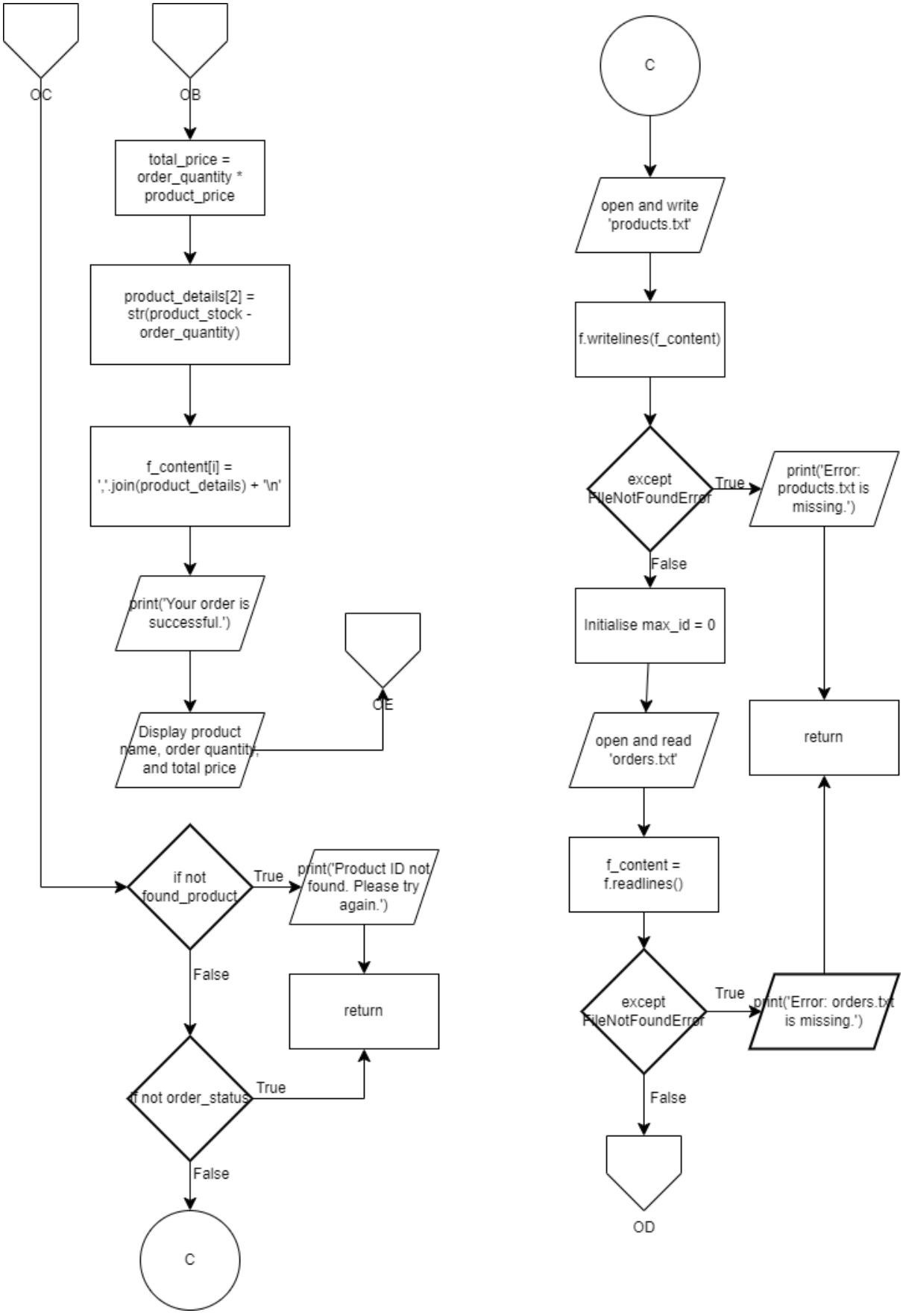
If the loop finishes without finding a matching product, the **found_product** flag remains False. A message informs the user that the product ID was not found, and the function exits. If the order was not processed successfully, the **order_status** flag remains False and the function exits without making any updates. While if the order is successful, the updated product stock data in **f_content** list is written back to **product.txt** file. The file is opened in write mode and all updated product details lines are written to the file. To generate a unique order ID, the **order.txt** file is read line by line to find the largest existing order ID. The numeric part of each order ID is extracted (skipping the first character O) and converted to an integer. The largest number is incremented by 1 to create a new order ID, formatted as Oxxx (e.g., O001). The current date is retrieved using **datetime.now()** and formatted as DD/MM/YYYY. Now, the new order details, including the new order ID, product ID, order quantity, purchase data and supplier ID are appended to **orders.txt** file. If the file is missing, an error message is displayed, and the function exits.

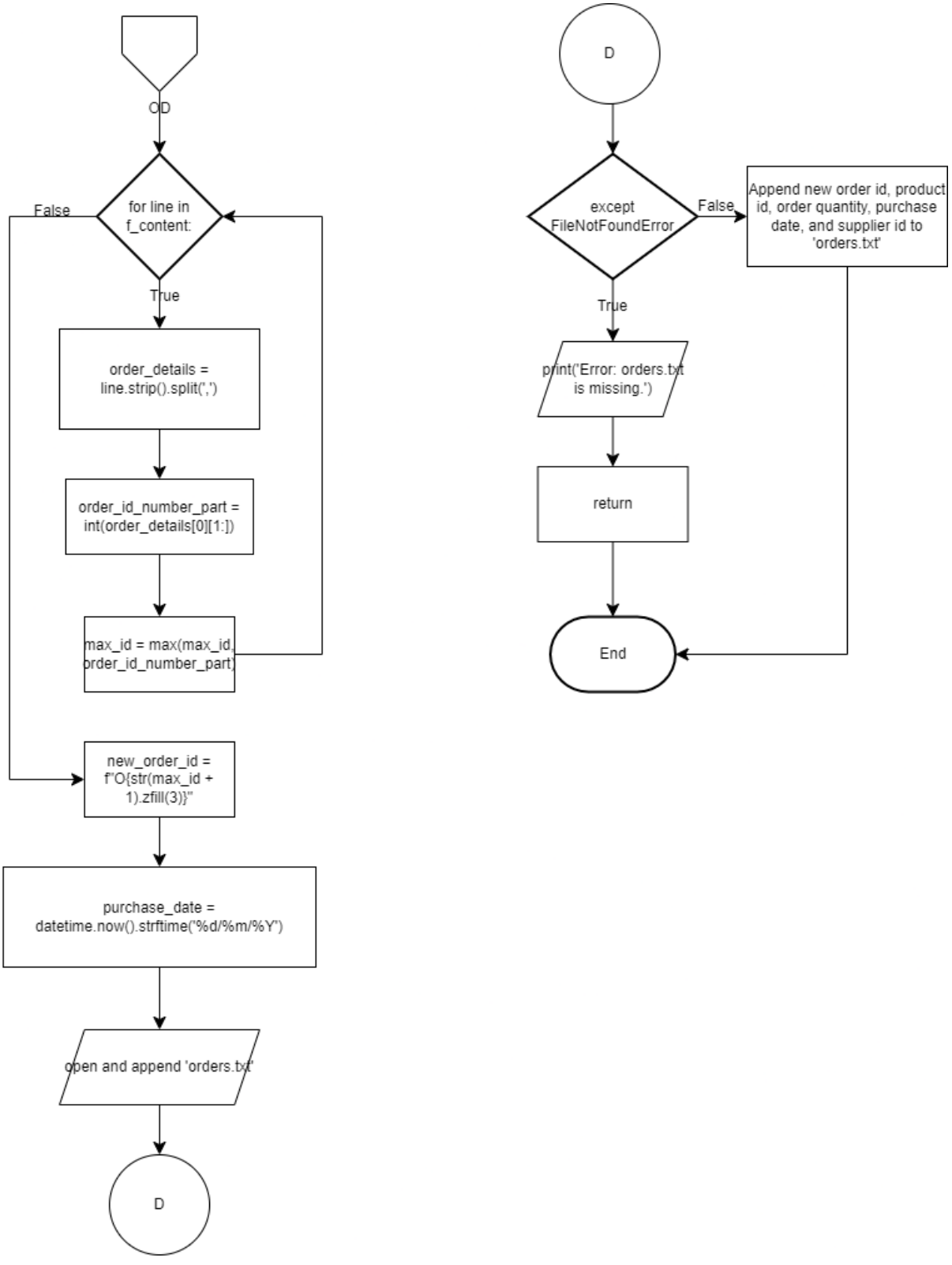
Overall, this place an order function ensures that users enter a valid supplier, product, and valid order quantity to place an order.

Flowchart


Start
Enter supplier id
open and read 'supplier .txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: suppliers.txt is missing.')
return
False
found_supplier = False
for line in f_content
False
True
supplier_details = line.strip().split(',')
A
A
if supplier_details[0] == supplier_id
False
True
found_supplier = True
if not found_supplier
True
print('Supplier ID not found. Please try again.')
return
False
Enter product id
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
return
False
OA


OA
found_product = False
order_status = False
OE
False
for i, line in enumerate(f_content)
True
product_details = line.strip().split(',')
if product_details[0] == product_id
False
True
found_product = True
While True
True
B
OC
break
B
Enter order quantity
except ValueError
True
print('Please enter a valid integer for quantity.')
False
if order_quantity > 0
False print('Quantity must be greater than 0.')
True
break
Assign product details into product name, price, and stock
if product_stock >= order_quantity
False
print('Not enough stock available.')
True
order_status = True
OB


OC
OB
total_price = order_quantity * product_price
product_details[2] = str(product_stock - order_quantity)
f_content[i] = ','.join(product_details) + '\n'
print('Your order is successful.')
OE
Display product name, order quantity, and total price
if not found_product
True
print('Product ID not found. Please try again.')
False
return
if not order_status
True
False
C
C
open and write 'products.txt'
f.writelines(f_content)
except FileNotFoundError
True
print('Error: products.txt is missing.')
False
Initialise max_id = 0
open and read 'orders.txt'
return
f_content = f.readlines()
except FileNotFoundError
True
print('Error: orders.txt is missing.')
False
OD

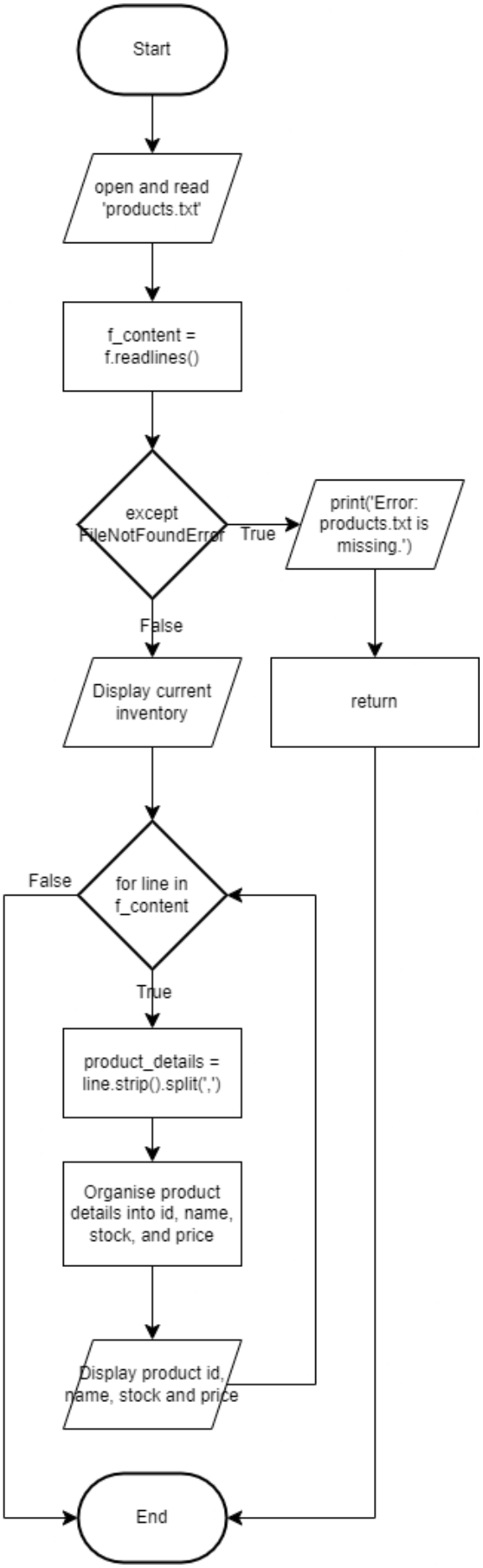

OD
for line in f_content:
False
True
order_details = line.strip().split(',')
order_id_number_part = int(order_details[0][1:])
max_id = max(max_id, order_id_number_part)
new_order_id = f"O{str(max_id + 1).zfill(3)}"
purchase_date = datetime.now().strftime("%d/%m/%Y")
open and append 'orders.txt'
D
D
except FileNotFoundError
False
Append new order id, product id, order quantity, purchase date, and supplier id to 'orders.txt'
True
print('Error: orders.txt is missing.')
return
End

1.6 View Inventory

The view inventory function is designed to display the current inventory of products stored in the **products.txt** file. The function starts with a try-except block to handle potential errors when opening the **products.txt** file. The try block ensures that the file is opened in read mode. All lines are read using **f.readlines** and stored in the **f_content** list, where each line represents a product details. For the except block, if the file is not found, the FileNotFoundError is caught, an error message is printed, and the function exits immediately using the return statement to prevent further execution. If the **product.txt** file is successfully read, the function proceeds to print the inventory header with a specific format to offer a structured and user-friendly interface. Next, a for loop iterates over each line in the **f_content** list, where each line represents a product details. The function processes each line by using **strip()** method to remove any trailing whitespaces, and then applying **split(‘,’)** method to break the line into a list of attributes. From this list, the product's ID (product_id = product_details[0]), name (product_name = product_details[1]), stock (product_stock = int(product_details[2])), and price (product_price = float(product_details[4])) are extracted from the first, second, third, and fifth elements, respectively. These details are then displayed in a structured format: **product_id** and **product_name** are left aligned with 10 and 20 characters, **products_stock** with 10 characters, and **product_price** formatted to two decimal places, prefixed with a dollar sign ($).

Thus, this view inventory function provides a reliable and user-friendly way for users to view product inventory.

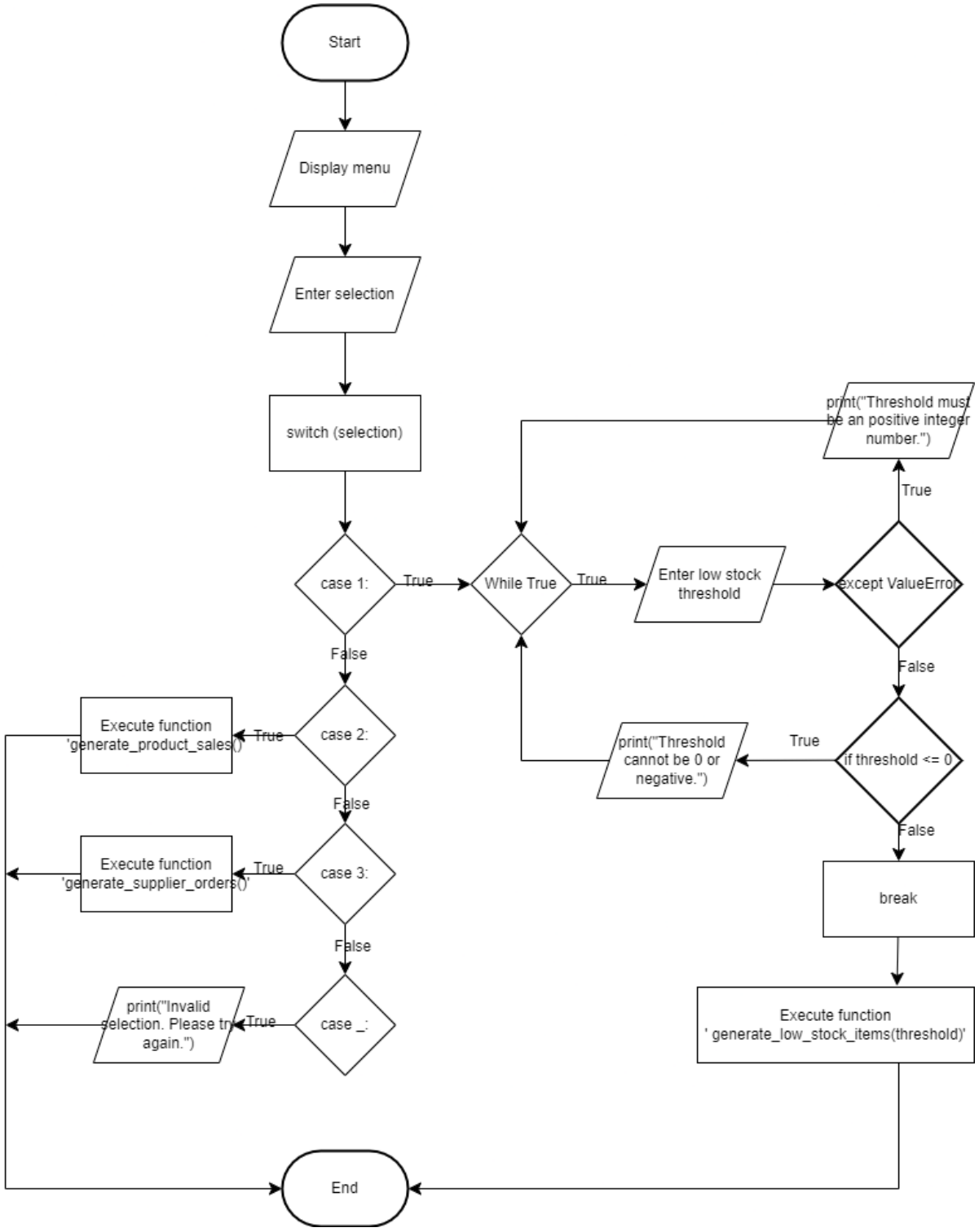
Flowchart


Start
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
False
Display current inventory
return
for line in f_content
False
True
product_details = line.strip().split(',')
Organise product details into id, name, stock, and price
Display product id, name, stock and price
End

1.7 Generate Reports

The generate report function is a menu-driven function that used to join all report generated functions. It allows users to generate specific types of reports. First, it presents the user with a menu of three report generated options: (1) low stock Items, (2) product sales, and (3) supplier orders. To run the selected decision by the user, a match-case will be utilized. The user is then prompted to enter their choice (‘1’, ‘2’, or ‘3’). If the user selects case ‘1’, the program prompts the user to input a threshold value for low stock. A while loop ensures that the input is valid, as the threshold value must be greater than zero (positive integer). If not, the user is prompted repeatedly until a valid input is provided. Once a valid threshold is entered, the **generate_low_stock_items function** (will be presented in 1.7.1) is called to generate the report. For option 2, the program directly calls the **generate_product_sales function** (will be presented in 1.7.2) to generate a sales report. Similarly, for option 3, it calls the **generate_supplier_order function** (will be presented in in 1.7.3) to produce a supplier orders report. However, if the user enters an invalid option, the program displays an error message and return back to the main function.

Flowchart

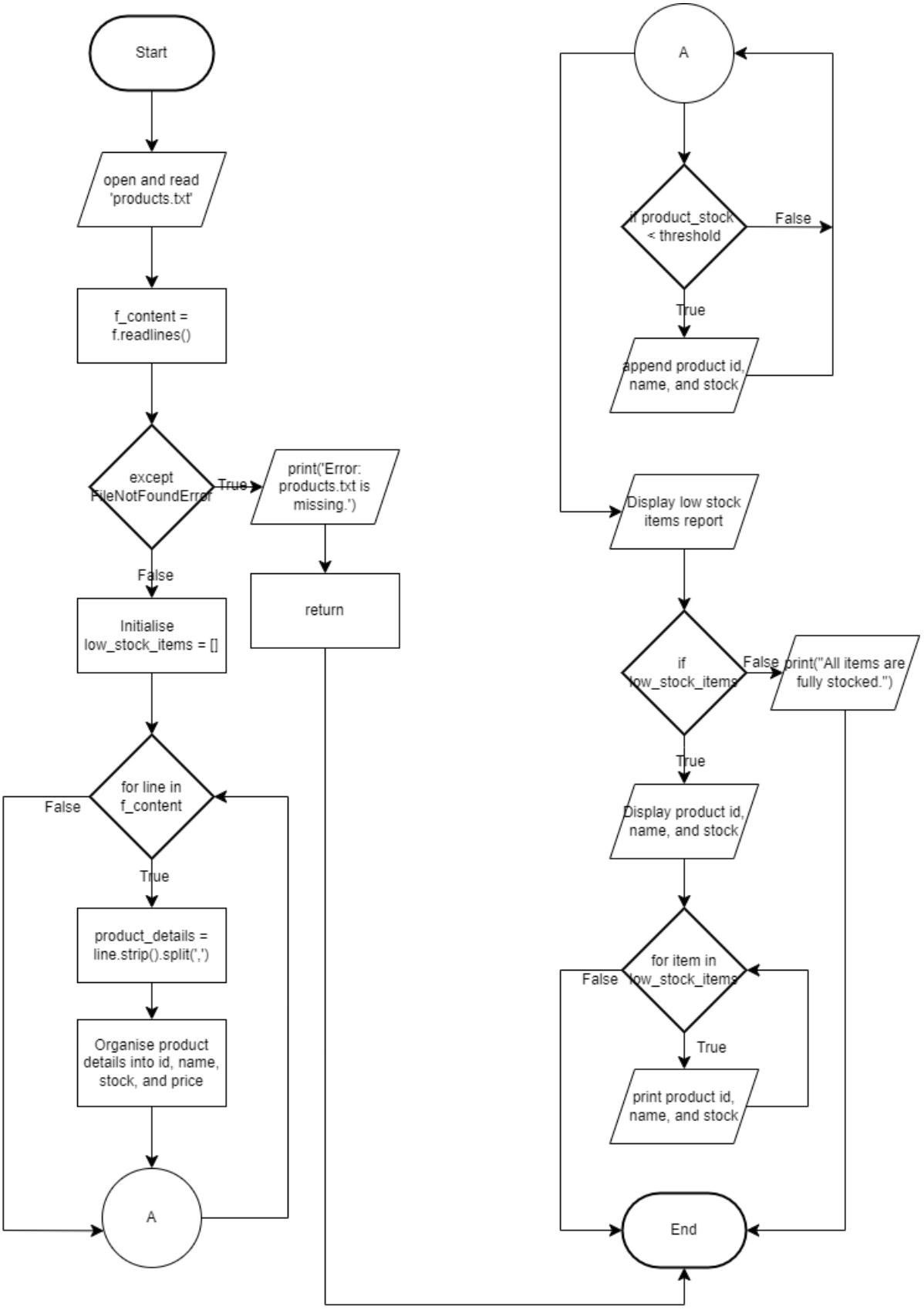

Start
Display menu
Enter selection
switch (selection)
case 1:
True
While True
True
Enter low stock threshold
except ValueError
print("Threshold must be an positive integer number.")
True
False
if threshold <= 0
True
print("Threshold cannot be 0 or negative.")
False
break
Execute function ' generate_low_stock_items(threshold)'
False
case 2:
True
Execute function 'generate_product_sales()'
False
case 3:
True
Execute function 'generate_supplier_orders()'
False
case _:
True
print("Invalid selection. Please try again.")
End

1.7.1 Display Low-stock Items

The program directly calls the **generate low stock items function** when the user selects option 1 to generate a report for low stock items in the **generate_reports function** (as presented in section 1.7). This function generates a report of products with stock levels below a specific threshold entered by the user. First, the function attempts to open and read data from the **products.txt** file in read mode. If the file is missing, a FileNotFoundError is caught, an error message is displayed, and the function exits early. Next, an empty list **low_stock_items[]** is initialized to store details of products with stock below the threshold. The function iterates through each line in the **f_content** list, where each line represents a product's details. It removes leading and trailing whitespaces using the **strip()** method and splits the line into a list using the **split(',')** method. The product ID (product_id = product_details[0]), name (product_name = product_details[1]), and stock (product_stock = int(product_details[2])) from the **product_details** list are extracted, with the stock value being converted to an integer for comparison. The function then checks if the stock quantity (product_stock) is below the threshold. If true, a dictionary containing the product ID (product_id), name (product_name), and stock level (product_stock) is appended to the **low_stock_items** list. Once all products have been processed, the function displays the relevant product details, including product IDs, names and stock levels in a clear and well-structured report format. The report columns are aligned with specific widths for each element group to enhance readability. If no products are below the threshold, a message is displayed stating that all items are fully stocked.

Thus, this generate low stock items function helps users identify products with insufficient stock in a clear and professional report format.

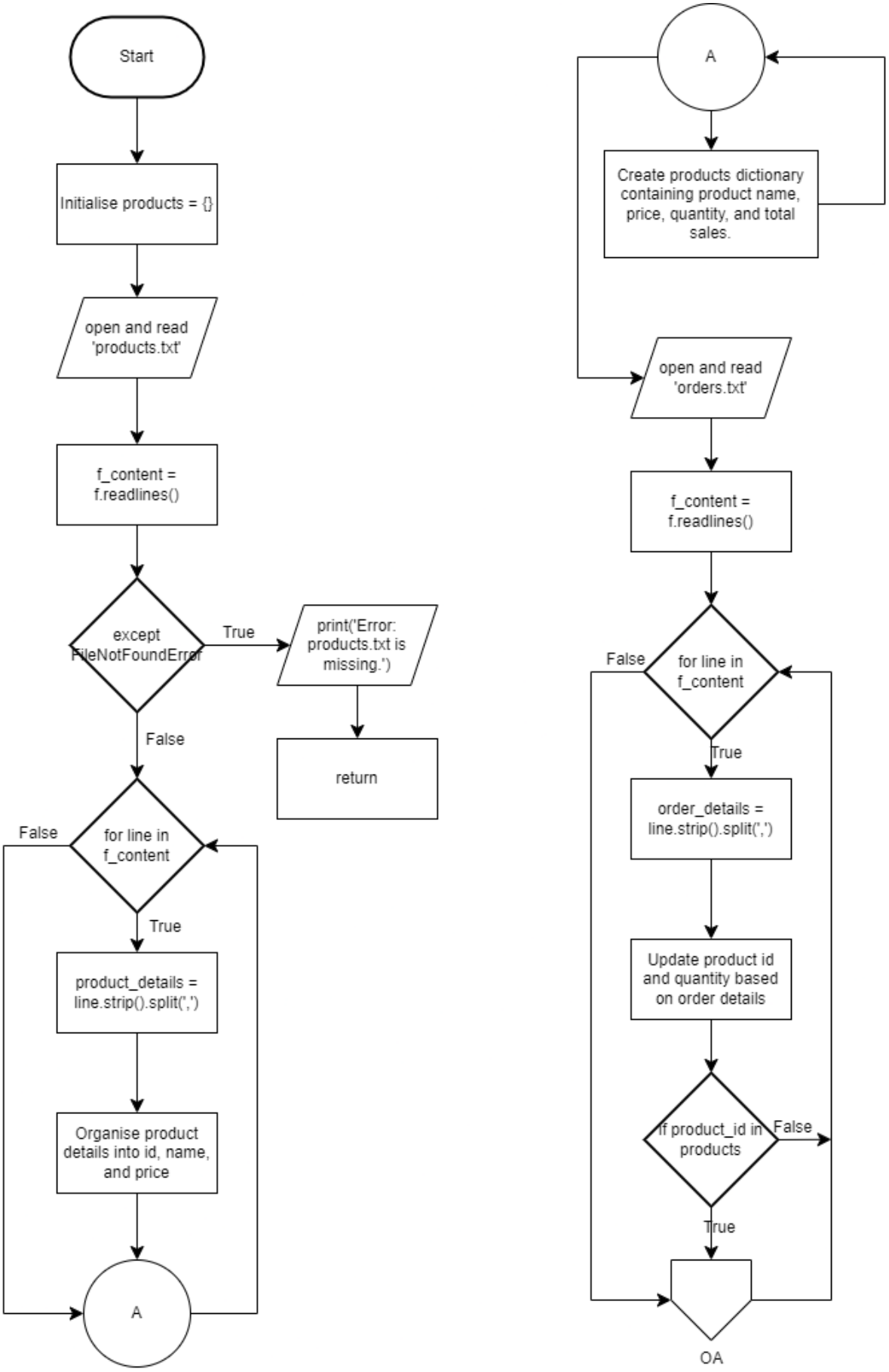
Flowchart

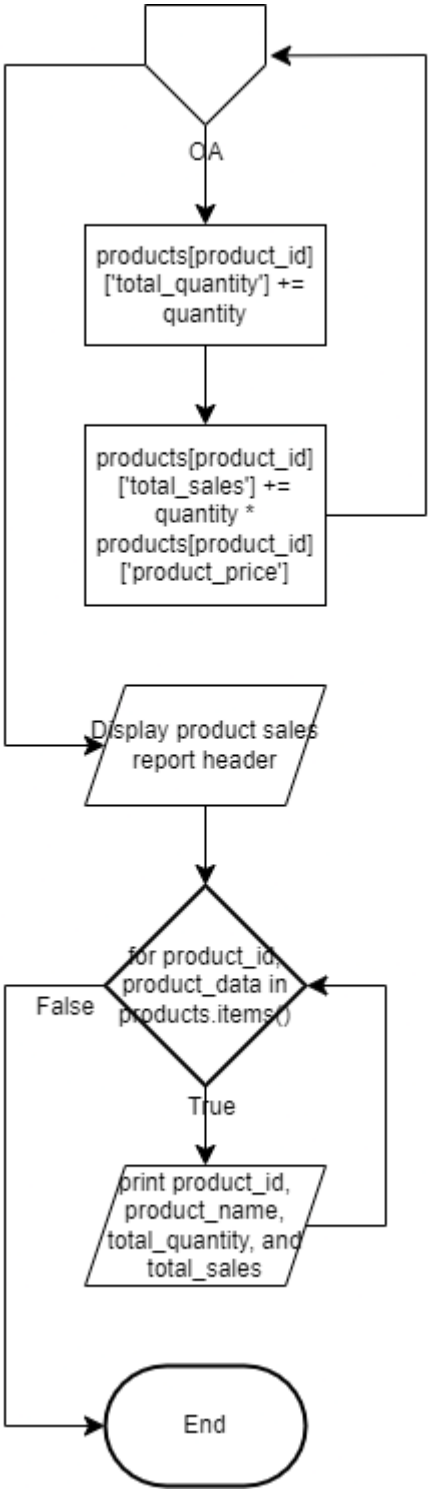

Start
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
return
False
Initialise low_stock_items = []
for line in f_content
False
True
product_details = line.strip().split(',')
Organise product details into id, name, stock, and price
A
A
if product_stock < threshold
False
True
append product id, name, and stock
Display low stock items report
if low_stock_items
False
print("All items are fully stocked.")
True
Display product id, name, and stock
for item in low_stock_items
False
True
print product id, name, and stock
End

1.7.2 Display Product Sales

Similar to the previous function, the program directly calls the **generate product sales function** when the user selects option 2 to generate a report for total product sales in the **generate reports function** (as presented in section 1.7). This function calculates and generates a report showing the total quantity and total sales for each product. First, an empty dictionary, **products**, is created to store product details. The function then tries to open the **products.txt** file in read mode. If the file is missing, a FileNotFoundError is caught, an error message is displayed, and the function exits early. For each line in the **products.txt** file, the whitespaces is removed using the **strip()** method and the line is split into product details using the **split(',')** method. The product ID (product_id = product_details[0]), name (product_name = product_details[1]), and price (product_price = float(product_details[4])) converted to float are extracted, and a new entry is added to the **products** dictionary with the product ID as the key. Each entry contains the product name, price, and initializes **total_quantity** and **total_sales** to zero. Next, the function opens the **orders.txt** file to read the orders placed. For each line in the **order.txt** file, the product ID and quantity are extracted. If the product ID exists in the products dictionary, the quantity ordered is added to the total quantity (total_quantity) for the relevant product, and the total sales (total_sales) are updated by multiplying the quantity by the product's price (product_price). Once all orders are processed, the function generates a report. It displays the product ID (product_id), product name (product_name), total quantity sold (total_quantity), and total sales amount (total_sales) in well-structured report format. The specific columns are aligned for readability, the product ID and the product name is left-aligned; the total quantity is right-aligned, and the total sales amount is right-aligned displayed with two decimal places prefixed by a dollar sign ($). Finally, the function prints out the sales data for each product in the products dictionary in a well-organized format. Overall, the generate product sales function displays the total quantity and total sales of each product to users in a structured and clear report format.

Flowchart


Start
Initialise products = {}
open and read 'products.txt'
f_content = f.readlines()
except FileNotFoundError
True
print('Error: products.txt is missing.')
return
False
for line in f_content
False
True
product_details = line.strip().split(',')
Organise product details into id, name, and price
A
A
Create products dictionary containing product name, price, quantity, and total sales.
open and read 'orders.txt'
f_content = f.readlines()
for line in f_content
False
True
order_details = line.strip().split(',')
Update product id and quantity based on order details
If product_id in products
False
True
OA

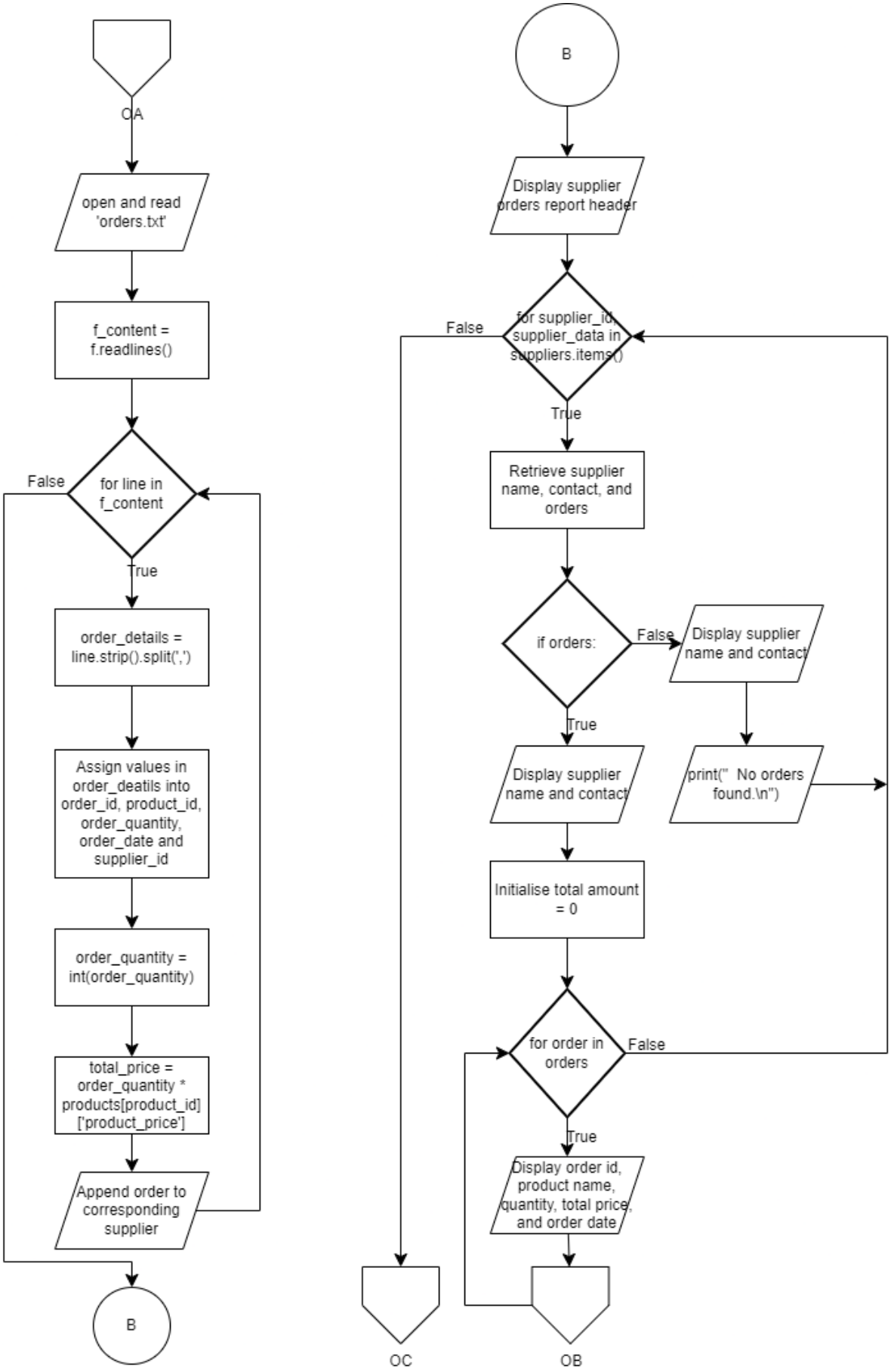

OA
products[product_id]['total_quantity'] += quantity
products[product_id]['total_sales'] += quantity * products[product_id]['product_price']
Display product sales report header
for product_id, product_data in products.items()
False
True
print product_id, product_name, total_quantity, and total_sales
End

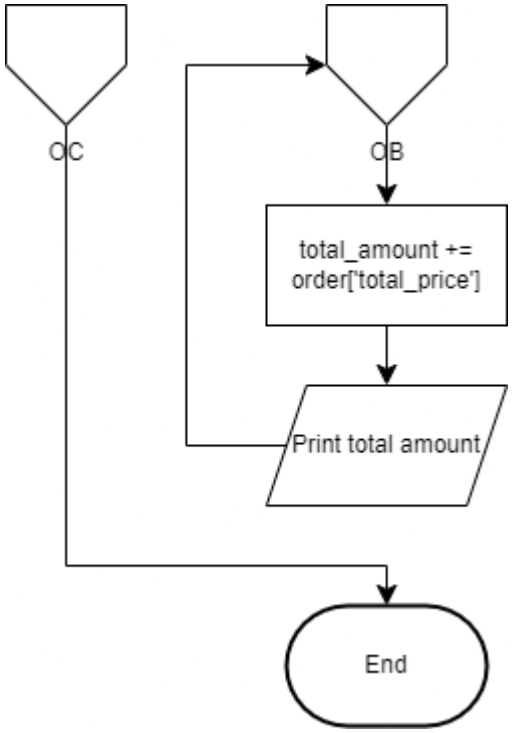
1.7.3 Display Supplier Orders

Similar to the previous functions, when the user selects option 3 to generate a report for supplier order in the **generate report function** (as presented in section 1.7). The program directly calls the **generate supplier orders function**. This function generates a report of supplier orders to users. First, two dictionaries, **suppliers** and **products** are initialized to store data about suppliers and products respectively. The function attempts to open the **suppliers.txt** file in read mode. If the file is missing, a FileNotFoundError is caught by except block, an error message is displayed, and the function exits early. The **supplier.txt** file is opened and read line by line. For each line, the leading and trailing whitespaces are removed using **strip()** method, the function splits the data by **splits(',')** and extracts the supplier ID (supplier_id = supplier_details[0]), name (second element) and contact number (third element). Each supplier's information is stored in the suppliers dictionary, with the supplier ID as the key and a dictionary containing the supplier's name (supplier_name), contact (supplier_contact), and an empty list for storing their orders (supplier_orders). After that, the function attempts to open the **product.txt** file in read mode. If the file is missing, a FileNotFoundError is caught by the except block, an error message is displayed, and the function exits early. The **products.txt** file is read line by line. Each product's details (product_details) are split and stored in the **products** dictionary with the product ID (products_id) as the key and the product's name (product_name) and price (product_price) as float. Similar to the previous step, the function tries to open the **orders.txt** file in read mode. If the file is missing, a FileNotFoundError is caught, an error message is displayed and the function exits immediately. The **orders.txt** file is read line by line. Each order's details (order_details), including the order ID (order_id), product ID (product_id), order quantity (order_quantity), order date (order_date) and supplier ID (supplier_id) are extracted. For each order, the total price (total_price) is calculated by the order_quantity * product_price from the products dictionary. The order details, such as the order ID, product name, order quantity, total price and order date are appended to the supplier_orders list of the corresponding supplier in the suppliers dictionary. After processing all orders, the function prints the report tittle, then iterates through each supplier in the **suppliers** dictionary. For each supplier, the supplier's name (supplier_name) and contact information (supplier_contact) are printed. If the supplier has orders, the details of each order, including the order ID, product name, order quantity and, total price and order data are displayed. The total amount (total_amount) for the supplier's orders is also calculated and printed. The report is structured with specific format and is easy to read. If no orders are found for the supplier, a message indicating no orders found is displayed. Hence, the generate supplier orders function displays all the suppliers' orders, including the order ID, product name, order quantity, total price of the product and total order amount in a clear report format.

Flowchart


Start
Initialise dictionaries:
suppliers = {}
products = {}
open and read
'suppliers.txt'
f_content =
f.readlines()
except FileNotFoundError
True
print('Error:
suppliers.txt is
missing.')
return
False
for line in
f_content
False
True
supplier_details =
line.strip().split(',')
supplier_id =
supplier_details[0]
Update 'suppliers'
dictionary with
supplier name,
contact, and orders
A
A
open and read
'products.txt'
f_content =
f.readlines()
except
FileNotFoundError
True
print('Error:
products.txt is
missing.')
return
False
for line in
f_content
False
True
product_details =
line.strip().split(',')
product_id =
product_details[0]
Update 'products'
dictionary with
product name and
price
OA

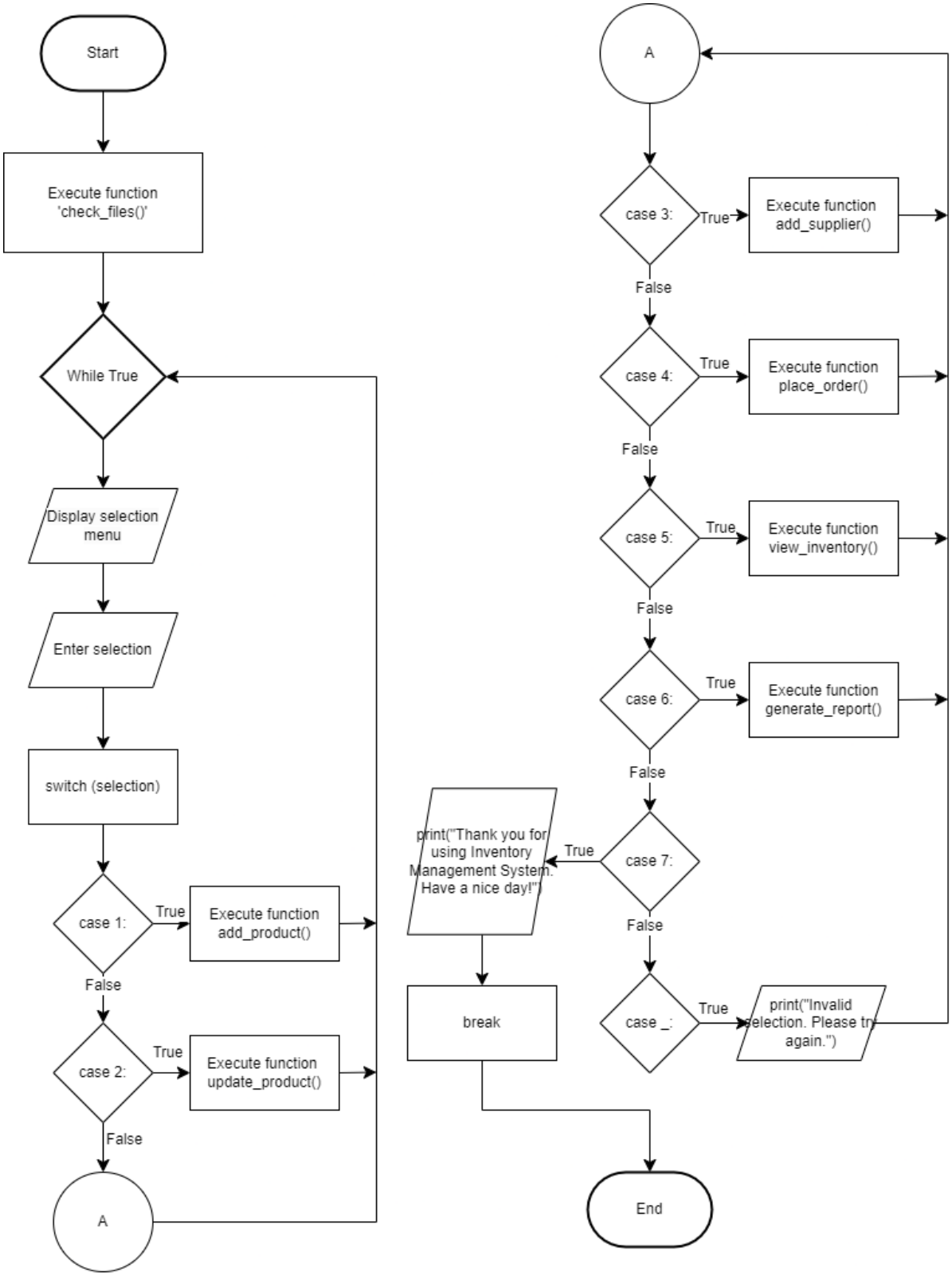

OA
open and read 'orders.txt'
f_content = f.readlines()
for line in f_content
False
True
order_details = line.strip().split(',')
Assign values in order_deatils into order_id, product_id, order_quantity, order_date and supplier_id
order_quantity = int(order_quantity)
total_price = order_quantity * products[product_id] ['product_price']
Append order to corresponding supplier
B
B
Display supplier orders report header
for supplier_id, supplier_data in suppliers.items()
False
True
Retrieve supplier name, contact, and orders
if orders:
False
Display supplier name and contact
print(" No orders found.\n")
True
Display supplier name and contact
Initialise total amount = 0
for order in orders
False
True
Display order id, product name, quantity, total price, and order date
OC
OB


OC
OB
total_amount +=
order['total_price']
Print total amount
End

1.8 Main program

The main program is used to join all functions together and run the whole program by providing the user with a selection page for them to select various functionalities via menu-driven interfaces. First, the function begins by calling the **check_files()** function (as presented in 1.1), which ensures that all necessary text files, including **products.txt** file, **suppliers.txt** file and **orders.txt** file are present. If any file is missing, the **check_files()** function displays an error message and exits the program early. A while True loop ensures the program continuously run until the user chooses to exit the program. Inside the loop, a menu is displayed using, providing 7 numbered options of functionalities namely (1) Add a new product, (2) Update existing product details, (3) Add a new supplier, (4) Place an order, (5) View inventory for all products, (6) Generate Reports, and (7) Exit program. The program is then prompting the user to enter their menu decision as number. The match statement is used to handle the user's input. For case '1' to '6', the program executes the corresponding function based on the user's selection. When the user selects case '7' to exit the program, a farewell message is displayed, the break statement exits the loop and terminates the program. Otherwise, if the user enters anything other than the valid menu options, the default case will be executed, an error message is displayed and the loop restarts.

Flowchart


Start
Execute function 'check_files()'
While True
Display selection menu
Enter selection
switch (selection)
case 1:
True
Execute function add_product()
False
case 2:
True
Execute function update_product()
False
A
A
case 3:
True
Execute function add_supplier()
False
case 4:
True
Execute function place_order()
False
case 5:
True
Execute function view_inventory()
False
case 6:
True
Execute function generate_report()
False
case 7:
True
print("Thank you for using Inventory Management System. Have a nice day!")
False
case _:
True
print("Invalid selection. Please try again.")
break
End

2.0 Discussion of Implementation Challenges and Solutions

1. **The problem of newline characters when processing data from files**
    - **Challenge:** When reading data from files such as products.txt, suppliers.txt, or orders.txt using the readlines() method, each line ends with a newline character (\n). This can cause issues when comparing file data with user input or manipulating the data.
    - **Solution:** The strip() method is used to remove leading and trailing whitespace, including newline characters. This ensures that the data read from files can be accurately compared or processed without interference from extraneous characters.
    - **Code Example:**

```
for line in f_content:
    product_details = line.strip().split(',') # separate elements in each line into a list by comma
```

2. **Difficulty in updating data within text files**
    - **Challenge:** Modifying information in text files requires reading the entire file, making changes, and rewriting the updated data back into the file. This process can be cumbersome and error-prone, especially when managing multiple files like products.txt or suppliers.txt.
    - **Solution:** The file content is first read into a list, where necessary modifications are made. After updating the list, the join() method is used to convert the list back to a string, which is then written back to the file using f.writelines(). This approach simplifies data updates and reduces potential errors.
    - **Code Example:**

```
        # check if still have stock
        if product_stock >= order_quantity:
            order_status = True
            total_price = order_quantity * product_price
            product_details[2] = str(product_stock - order_quantity) # update new stock
            f_content[i] = ','.join(product_details) + '\n' # update the specific line(product) details
            print('Your order is successful.')
            print(f'Product: {product_name}\nQuantity: {order_quantity}\nTotal price: ${total_price:.2f}')


        else:
```

```
            print('Not enough stock available.')
            break

    if not found_product:
        print('Product ID not found. Please try again.')
        return # exit function if product not found

    if not order_status:
        return # exit function if no stock

    # write back the new status(stock) of specific product if purchase success
    try:
        with open('products.txt', 'w') as f:
            f.writelines(f_content)

    except FileNotFoundError:
        print('Error: products.txt is missing.')
        return # exit the function if file not found
```

3. **Preventing Incorrect Order and Stock Updates**
    - **Challenge:** There is a risk of products being ordered multiple times without validation or stock levels being updated incorrectly, leading to false inventory data. This can cause significant disruptions in order processing and stock tracking.
    - **Solution:** Before placing or processing an order, the system validates the product's availability and stock levels. The order is rejected if the product is out of stock or insufficient. Two functions are created to handle stock validation and updates systematically. These ensure accurate inventory records and prevent incorrect orders.
    - **Code Example:**

```
product_stock = int(product_details[2])


            # check if still have stock
            if product_stock >= order_quantity:
                order_status = True
```

```
            total_price = order_quantity * product_price
            product_details[2] = str(product_stock - order_quantity) # update new stock
            f_content[i] = ','.join(product_details) + '\n' # update the specific line(product) details
            print('Your order is successful.')
            print(f'Product: {product_name}\nQuantity: {order_quantity}\nTotal price: ${total_price:.2f}')

        else:
            print('Not enough stock available.')
            break
```

4. **Complexity in placing an order**
    - **Challenge:** The place_order function is challenging to implement because it requires reading data from three separate files (products.txt, suppliers.txt, and orders.txt), updating stock levels, and appending a new order to the orders file. Additionally, the process involves cross-referencing product IDs and ensuring consistency.
    - **Solution:** Create modular functions to handle each task individually — reading and validating product data, updating stock levels, and appending new orders. This modular approach simplifies the implementation and debugging process.
    - **Code Example:**

```
product_stock = int(product_details[2])


        # check if still have stock
        if product_stock >= order_quantity:
```

3.0 Analysis of the System's Strengths and Limitations

3.1 Strengths

1. **Ability to Generate Informative Reports**

   The system includes functionality to produce comprehensive reports, such as low-stock alerts, order summaries, and supplier lists. These reports are formatted for clarity and provide valuable insights for inventory management.

2. **Automatic ID Generation**

   To ensure efficiency and prevent duplication, the system automatically generates unique IDs for products, suppliers, and orders. This eliminates the risk of human error and maintains data consistency.

3. **User-Friendly Interface**

   The program features a straightforward and intuitive interface. Input prompts are clear, and menu options are self-explanatory, allowing users to navigate easily and complete tasks without confusion.

4. **Effective Design and Robustness**
   The system reliably handles essential tasks like inventory updates, order placement, and report generation. Its modular design ensures smooth functionality, and error-handling mechanisms minimize disruptions during execution.

5. **Comprehensive Functionality**

   The system not only meets the basic requirements of an Inventory Management System but also incorporates additional features to enhance its utility. Functions such as validating unique IDs, checking stock levels before order placement, and generating detailed reports (e.g., low-stock alerts, supplier orders) extend the system's functionality and make it more versatile.

6. **Extensibility for Future Features**

   The modular design of the program allows for easy integration of new functionalities. With user-defined functions managing specific tasks, the codebase can be extended to include features like automated notifications for low stock or supplier restocking schedules.

3.2 Limitations

1. **No Option to Exit Mid-Task**

   The program requires users to complete their current task before returning to the main menu or switching to another operation. For example, users cannot interrupt the "Add

Product" process to perform a different action—they must finish or restart the task entirely.

2. **Input Errors Require Task Restart**

   If users make a mistake while entering some data (e.g., providing incorrect details for a product or supplier), the system does not allow for correction within the same task. Instead, users must restart the task, which can be time-consuming and frustrating.

3. **Absence of a Search Function**

   The current system lacks the ability to search for specific products or suppliers by attributes like name or ID. Users must view entire files (e.g., products.txt) to locate details, which can be inefficient and time-consuming.

4. **Stock Updates Depend on Precise Input**

   The system requires exact data alignment for stock updates. Any errors in data format or missing lines in the text files can lead to system crashes or inaccurate updates. This reliance on precise text file management is a potential vulnerability.

5. **Absence of Data Modification and Deletion Functions**

   Once data is stored, it cannot be edited (except for product) or removed. For instance, if supplier details need correction, the system does not provide a way to make those changes. This limitation can lead to outdated or incorrect information persisting in the system, which could confuse users and impact decision-making.

4.0 Suggestions for Future Improvements and Enhancements

1. **Implement a Graphical User Interface (GUI)**

   Transitioning from a text-based interface to a GUI using packages like PySimpleGUI or Tkinter would significantly enhance the user experience. A GUI would allow users to interact with buttons, drop-down menus, and checkboxes instead of typing commands, making the system more intuitive and accessible.

2. **Provide Options to Edit Stored Information**

   Introduce functionality to update or modify details of stored information directly from the program, such as correcting supplier contact details. This feature would eliminate the need for manual text file edits and reduce errors in the system.

3. **Introduce Mid-Task Exit Options**

   Allow users to exit or switch tasks mid-process without needing to complete the current task. For instance, if a user changes their mind while adding a new product, they should be able to cancel the operation and return to the main menu.

4. **Add an Undo Feature for Input Errors**

   Implement an undo option that allows users to correct recent input errors without restarting the entire task. This would streamline data entry and improve the overall experience.

5. **Support for Exporting and Importing Data**

   Enable data export to common file formats like CSV or Excel for external analysis. Similarly, it allows importing data from these formats to populate the system easily.