Warehouse Management System

Introduction to Programming

CMPT 120L

Cache Me If You Can

Marist College
School of Computer Science and Mathematics

Submitted To:
Dr. Reza Sadeghi

Date: Fall 2024

Table of Contents

Table of Figures

Project Description

PROJECT TITLE: WAREHOUSE MANAGEMENT SYSTEM

Summary: THE WAREHOUSE MANAGEMENT SYSTEM (WMS) PROVIDES AN ORGANIZED WAY OF STORING DIFFERENT PRODUCTS AND ELEMENTS IN A WAREHOUSE. YOU CAN CONSIDER A LIBRARY AS A WAREHOUSE, WHICH MAINTAINS BOOKS' DETAILS AND USER LIBRARIES. A GENERAL WMS STORES DETAILS OF NAME AND IDENTIFICATION NUMBER OF PRODUCTS, THEIR STORE TIME, THE REQUIRED STORAGE CONDITION, PRICE, WEIGHT, HEIGHT, ETC. FOLLOWING THIS, THIS SYSTEM ALLOWS GUEST USERS TO SEARCH FOR DIFFERENT CONTENT AND REQUEST TO BORROW/BUY THEM. YOUR WMS WILL STORE THE DATA OF DIFFERENT USER TYPES IN DISTINCT COMMA SEPARATED VALUE (CSV) FILES. THIS SYSTEM SHOULD AT LEAST SUPPORT THE FOLLOWING ITEMS:

1. ADMIN USER IS CAPABLE OF:
    a. HAVING ADMIN USER AND PASSWORD FOR LOG IN (A STRING OF AT LEAST 8 CHARACTERS)
    b. CHANGING THE ADMIN USER AND ADMIN PASSWORD
    c. ADDING A GUEST USER TO WMS BY CREATING A NEW USERNAME AND PASSWORD. A GUEST USER IS NOT ABLE TO DEFINE OR REMOVE OTHER USERS.
    d. REMOVING USERS FROM WMS BY REMOVING THEIR USERNAME, PASSWORD, AND CORRESPONDING RECORDED DATA.
    e. ADDING AN ITEM TO THE WAREHOUSE WITH VARIED DETAILS, SUCH AS:
        i. TYPE: FOOD, BOOKS, CARS, ETC.
        ii. STORED TIME IN THE WAREHOUSE
        iii. PICK OUT TIME FROM THE WAREHOUSE
        iv. ID: EACH ITEM IN YOUR LIBRARY SHOULD HAVE A UNIQUE IDENTIFICATION NUMBER WITH A SPECIFIC

FORMAT
v. NAME
vi. PROVIDER/CREATOR'S NAME
vii. QUANTITIES: THE NUMBER OF AVAILABLE ITEMS. FOR INSTANCE, ITEM X WITH A QUANTITY OF 2 IS A SIGN OF 2 AVAILABLE X ITEMS IN YOUR WAREHOUSE.
viii. PLACE: WHERE THE ITEM IS STORED
ix. PRICE.

f. DELETING AN ITEM FROM WAREHOUSE
g. EDITING AN ITEM IN WAREHOUSE
h. VIEWING THE LIST OF BORROWING REQUESTS
i. ACCEPTING OR REJECTING A BORROWING REQUEST

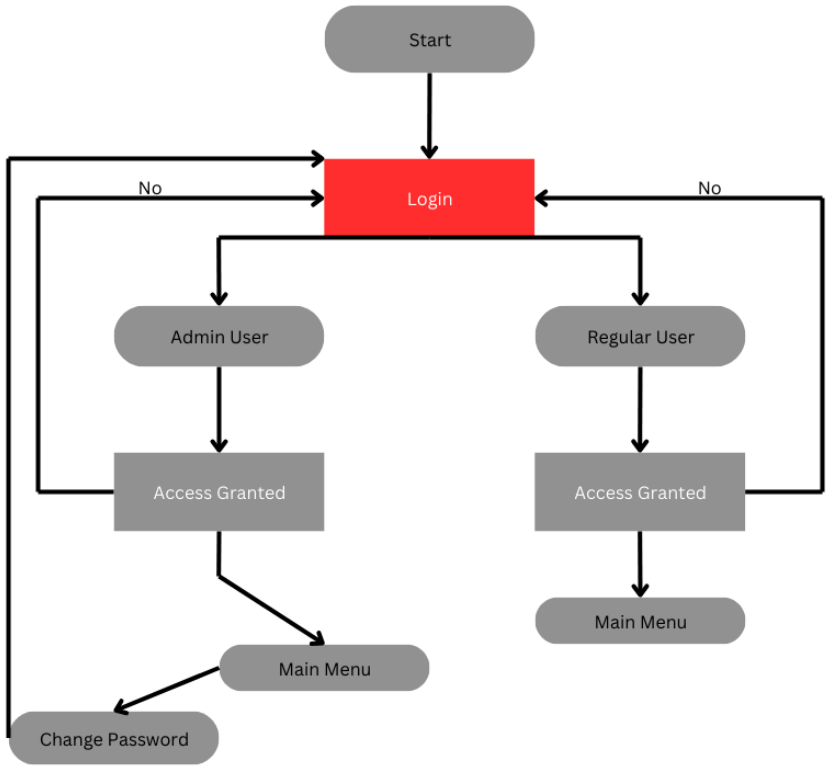
2. EACH USER SHOULD BE ABLE TO:
   a. SEARCH THROUGH WMS BASED ON ALL ITEMS' DETAILS, SUCH AS ID, NAME, AND PRODUCER.
   b. SAVE A LIST OF FAVORITE ITEMS
   c. REQUEST TO BORROW/BUY SOME ITEMS FOR A SPECIFIC TIME. FOR EXAMPLE, BORROWING ITEM A FOR 3 WEEKS.
   d. VIEW THE HISTORY OF BORROWED ITEMS
3. WMS SHOULD BE A USER-FRIENDLY SOFTWARE, SUCH THAT:
   a. IT SHOWS A WELCOME PAGE
   b. IT PROVIDES A MENU OF ALL FUNCTIONS TO THE USER IN ALL PAGES
   c. IT ILLUSTRATES THE REPORTS IN A TABULAR FORM. FOR INSTANCE, IT DISPLAYS A WELL-ORGANIZED LIST OF THE REQUESTED ITEMS.
   d. WMS SHOULD PROVIDE AN EXIT FUNCTION AND THANK THE USER FOR USING THIS SOFTWARE.
   e. IT SHOWS A WARNING IF:
      i. THE ADMIN USER TRIES TO ADD A NEW ITEM TO THE LIBRARY WITH AN EXISTING ID.
      ii. IF A GUEST USER TRIES TO BORROW MORE THAN 3 ITEMS.
      iii. A USER SEARCH REQUEST RETURNS NULL ITEMS.
4. WMS SHOULD PROTECT THE USER INFORMATION, SUCH THAT: a. OPTIONAL: WMS PASSWORDS AND THE RECORDED INFORMATION SHOULD BE CIPHERED. IN THE SIMPLEST CASE, YOU CAN USE CAESAR CIPHER METHODOLOGY. THE EASIEST WAY TO UNDERSTAND THE CAESAR CIPHER IS TO THINK OF CYCLING THE POSITION OF THE LETTERS. IN A CAESAR CIPHER WITH A SHIFT OF 3, A BECOMES D, B BECOMES E, C BECOMES F, ETC. WHEN REACHING THE END OF THE ALPHABET IT CYCLES AROUND, SO X BECOMES A, Y BECOMES B, AND Z BECOMES C.

Login Page:

Input: Username and Password in text

Output: Valid username and password enable the user to move onto the main menu. If the username and password match the admin username and password, the user will be taken to a menu where they may change the admin password, or proceed to the admin main menu . If the password is invalid, a message is displayed informing the user that their information is invalid, and prompts them to log in again.

Figure 1 Login Phase Flowchart [1]

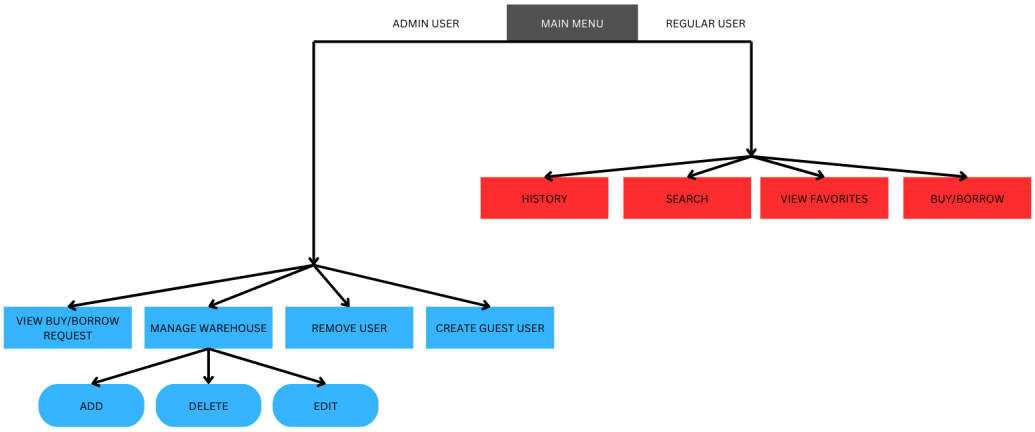


Main Menu:

Input: User will be provided with a list of options which will change depending on if the user used Admin or Regular login credentials. User will pick one option

Output: User will be taken to another page corresponding to the option they choose.

Figure 2 Main Menu Flowchart [2]

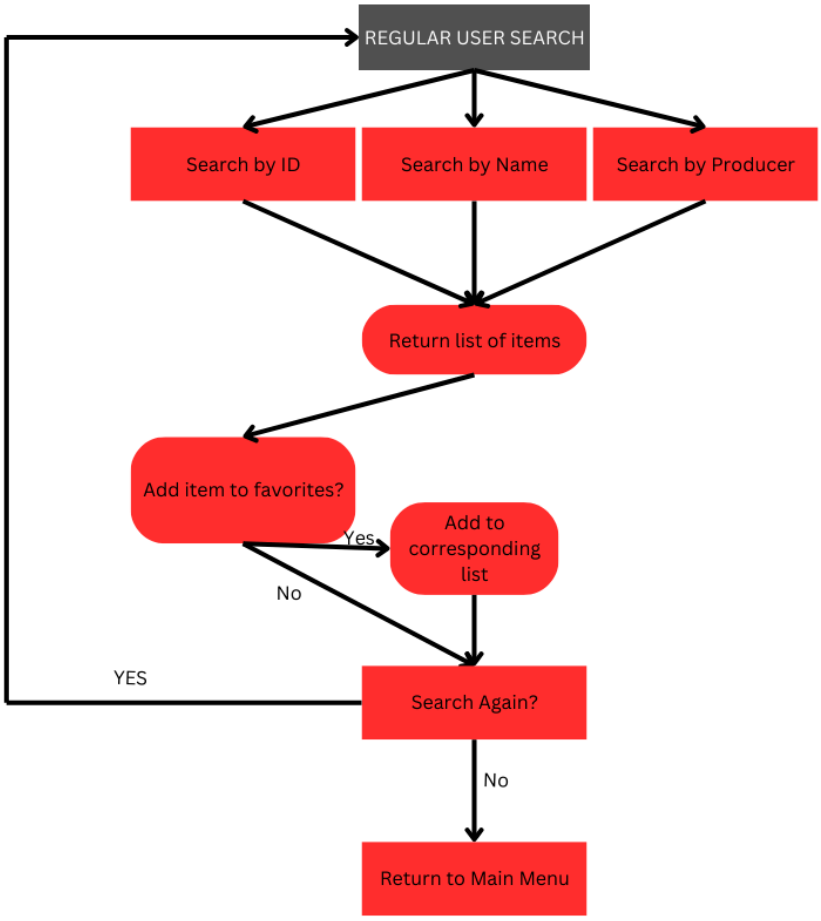


Regular User Search:

Input: Users will be prompted to choose if they want to search by ID, Product Name, or Producer. Then they will be prompted to search. After the output is provided, the user will also be prompted to add the item to their favorites list. Afterwards, they will be asked if they want to search again.

Output: A list of products, and follow up prompts asking for further inputs. If the user does not want to search again, they will be taken back to the regular user main menu,

Figure 3 Regular User Search Flowchart [3]

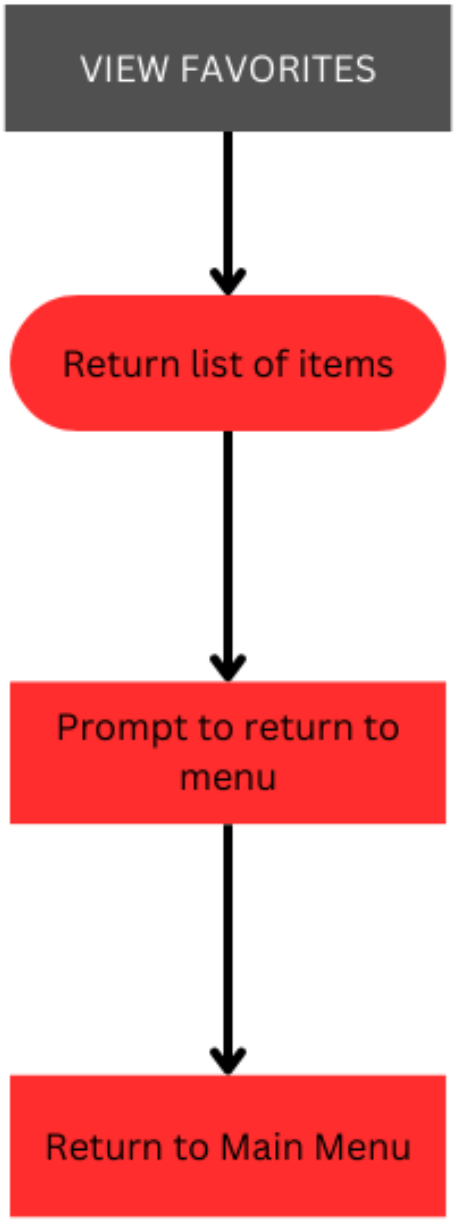


Regular User View Favorites

Input: Users will be prompted to return to the main menu.

Output: Users will be presented with a list of items in their favorite list.

Figure 4 View Favorites Flowchart [4]

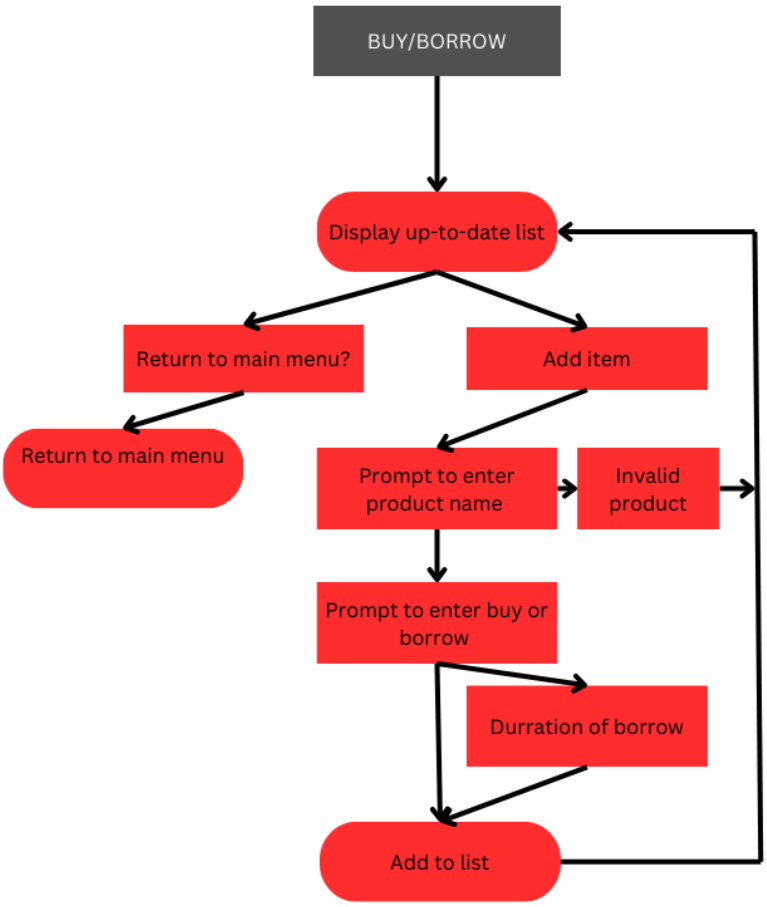


Regular User Buy or Borrow

Input: Users will be asked to provide the name of the product, if they are buying or borrowing it, and the duration of their borrowing. Users will also have the option to return to the main menu.

Output: List of items on the buy list presented after each item is added. Follow up questions are also part of output. If the product item is invalid, the user will be redirected to the first output.

Figure 5 Buy or Borrow Flowchart [5]

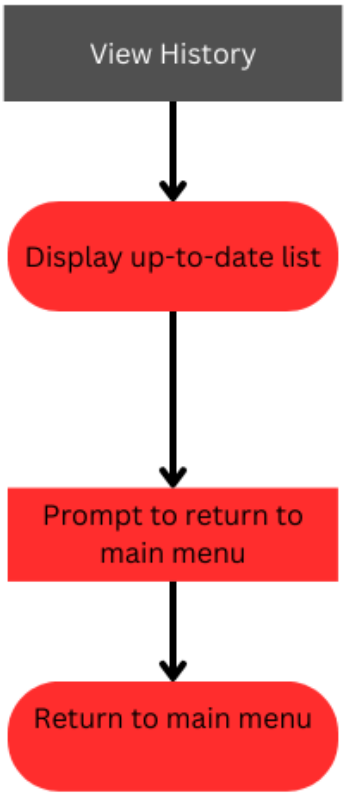


Regular User View History

Input: Users will be prompted to return to the main menu after the list is read.

Output: List of all purchases or borrows, and a prompt to return to the main menu.

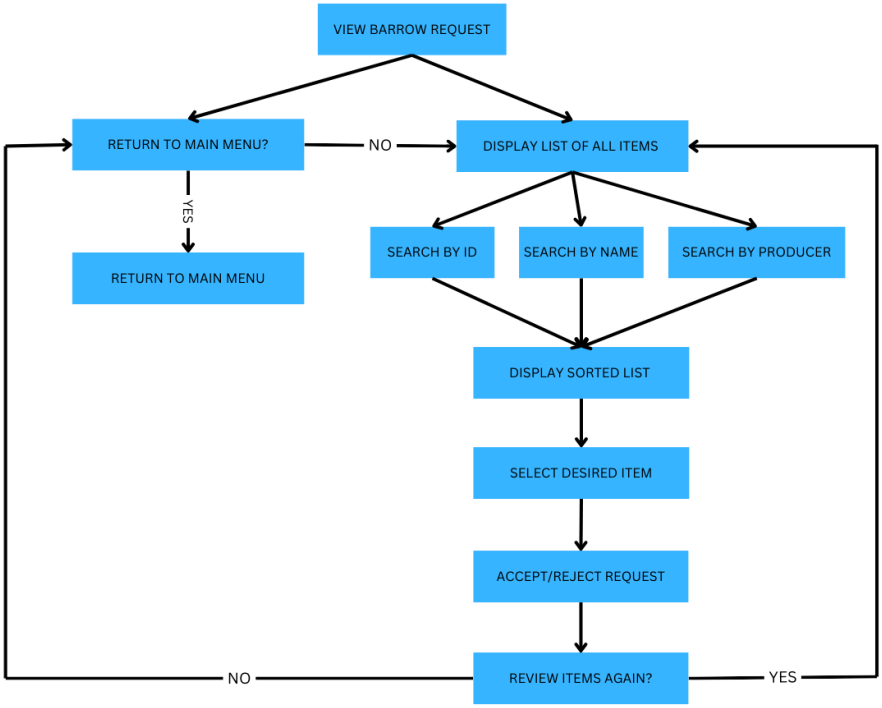
Figure 6 View History Flowchart [6]



Admin View Borrow Request:

Input: The admin will be asked to search the list of all items requested to be borrowed to find specific requests. They will either accept or reject the request and be prompted to return to the main menu or review more requests.
Output: If rejected, the request will be removed. If accepted, the request will be approved and moved into warehouse management.

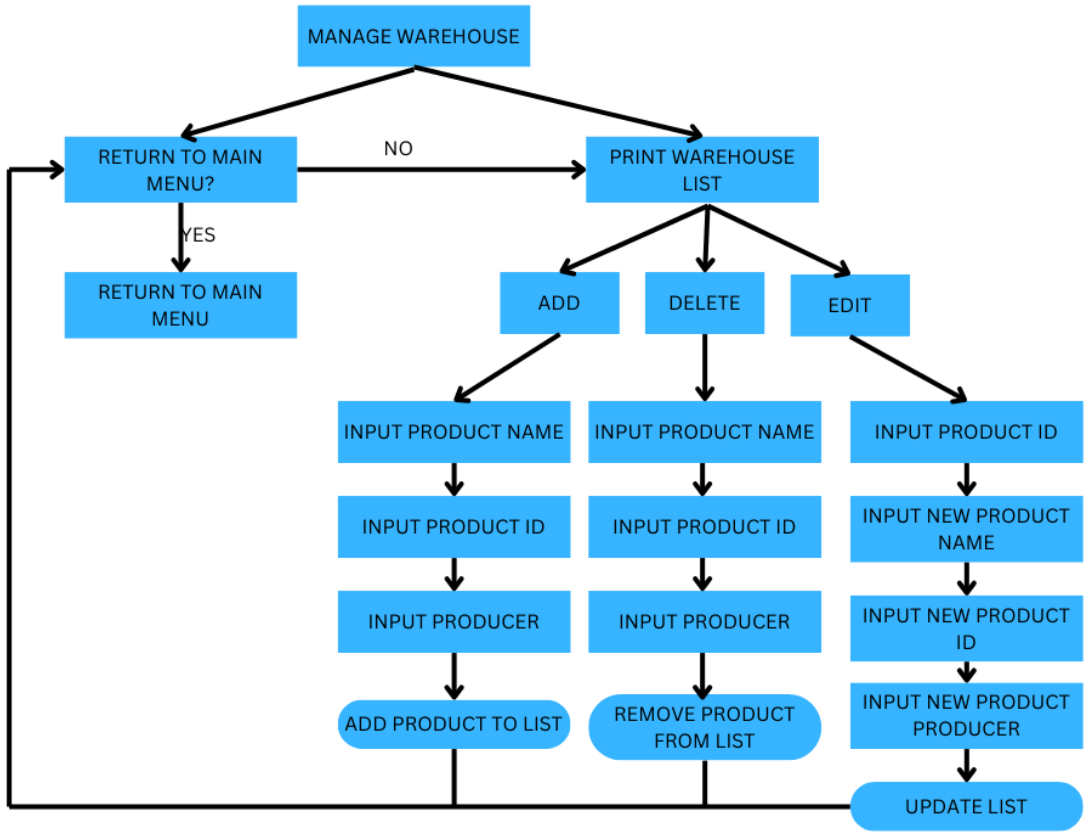
Figure 7 View Barrow Request [7]



Admin Manage Warehouse:

Input: The admin will be prompted to add, delete, or edit an item from the warehouse. For accuracy, all three actions will require the product's name, ID, and producer before updating the list and prompting the admin to return to the main menu or adjust the warehouse again.
Output: The warehouse list will be updated for the regular user to view

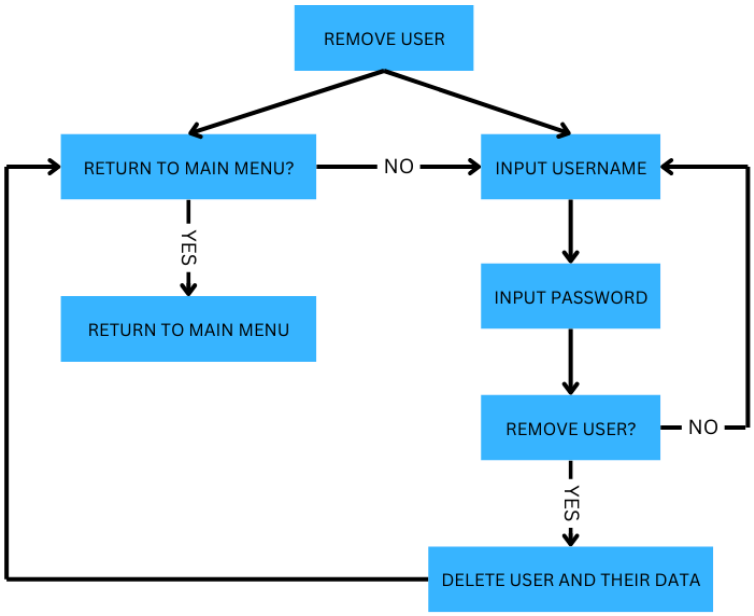
Figure 8 Manage Warehouse [8]



Admin Remove User:

Input: The admin will input the username and password of the desired user to remove before being prompted to confirm the removal of the user
Output: The desired user will be removed along with all of their corresponding data

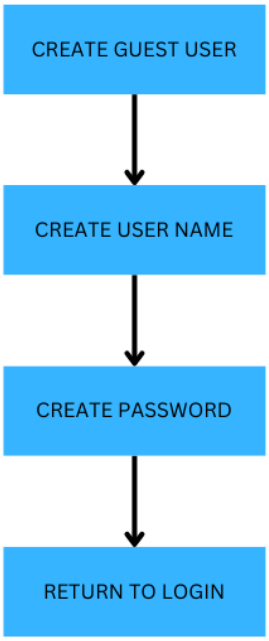
Figure 9 Remove User [9]



Admin Create Guest User:

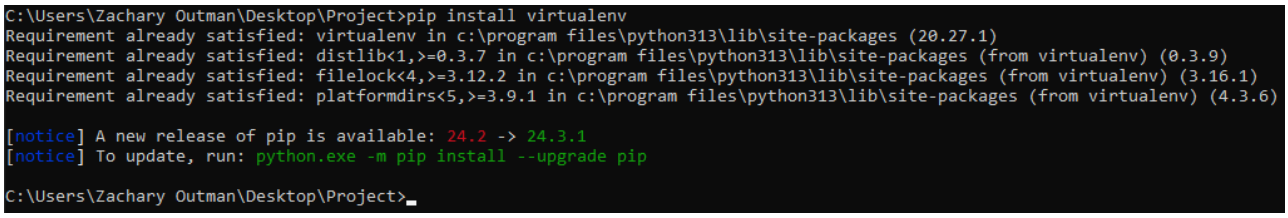
Input: The admin will input a new username and password for the corresponding username
Output: New guest user created, returning to the login page for the new user to log in

Figure 10 Create Guest User [10]



Ensure that virtualenv is installed/ install virtualenv using pip (installed with python)

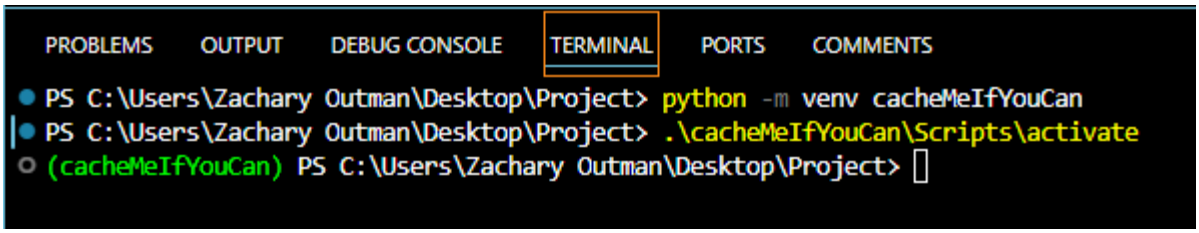
Figure 11 CMD installing virtualenv [11]



List of packages:
pip
virtualenv
Venv
Tkiner
Messagebox(From Tkiner)
os
csv
Datetime

Running a virtual environment in an IDE (VS Code)

Figure 12 Virtual Environment being created then run in VS Code [12]



Login Page: The Login Page allows users to input a username and password for authentication. If the credentials are correct, a success message is shown, and the user is directed to the Main Menu. If incorrect, an error message is displayed. Users can also register new accounts via the registration window, which adds the credentials to the system.

Figure 13 loadUsers function, saveUsers function, setupGui function

```
def loadUsers():
    if not os.path.exists(credFile):
        with open(credFile, "w", newline="") as f:
            pass
    else:
        try:
            with open(credFile, "r") as f:
                for row in csv.reader(f):
                    if len(row) == 2:
                        users[row[0]] = row[1]
        except Exception as e:
            messagebox.showerror("Error", f"Error reading user file: {e}"


def saveUser(user, pw):
    #Save a new user to the CSV file.
    try:
        with open(credFile, "a", newline="") as f:
            csv.writer(f).writerow([user, pw])
    except Exception as e:
        messagebox.showerror("Error", f"Error saving user: {e}")

def setupGui(root):
    #Set up the login and register GUI
    adminUser = "admin"
    adminPw = "1234"
    users[adminUser] = adminPw  # Add admin credentials by default
```

Figure 14 register function

```
def register():
    #Open the registration window to create a new account
    def addUser():
        #Register a new user.
        newUser = regUser.get().strip()
        newPw = regPw.get().strip()

        if not newUser or not newPw:
            messagebox.showerror("Error", "Fill all fields!")
        elif newUser in users:
            messagebox.showerror("Error", "User already exists!")
        else:
            users[newUser] = newPw
            saveUser(newUser, newPw)
            messagebox.showinfo("Success", "User registered successfully!")
            regWin.destroy()

    regWin = tk.Toplevel(root)
    regWin.title("Register")
    regWin.geometry("300x200")
    tk.Label(regWin, text="New Username").pack()
    regUser = tk.Entry(regWin)
    regUser.pack()
    tk.Label(regWin, text="New Password").pack()
    regPw = tk.Entry(regWin, show="*")
    regPw.pack()
    tk.Button(regWin, text="Register", command=addUser).pack()
```

Figure 15 login function

```
def login():
    #Prompt the user screen and handle login
    user = userEntry.get()
    pw = pwEntry.get()

    if user in users and users[user] == pw:
        if user == adminUser:
            messagebox.showinfo("Login", "Welcome, Admin!")
            showMenu("admin")
        else:
            messagebox.showinfo("Login", f"Welcome, {user}!")
            showMenu("user")
    else:
        messagebox.showerror("Error", "Invalid login.")
```

Main menu: The Main Menu dynamically changes based on the user type (admin or regular user), displaying relevant options as buttons. Clicking a button triggers a confirmation message indicating the selected option. The Login Page handles user authentication and registration, while the Main Menu serves as the application's navigation hub.

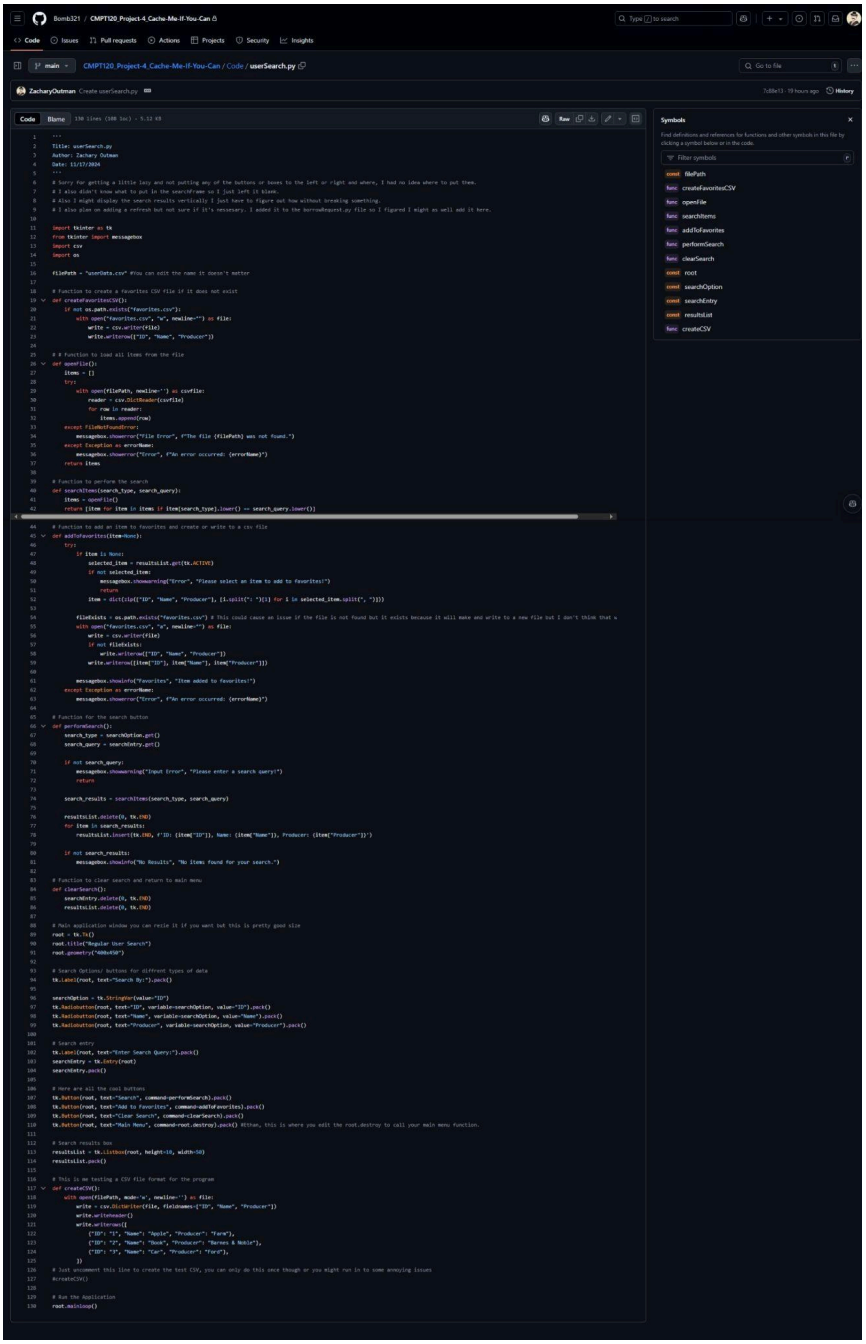
Figure 16 showMenu function

```
def showMenu(userType):
    #Display the menu depending on the Uset ype, Admin or Regular
    for widget in root.winfo_children():
        widget.destroy()

    tk.Label(root, text=f"Menu - {userType.capitalize()}").pack()
    if userType == "admin":
        options = ["View Requests", "Add User", "Manage Stock"]
    else:
        options = ["View History", "Search Items", "Buy"]

    for opt in options:
        tk.Button(root, text=opt, command=lambda o=opt: messagebox.showinfo("Option", f"{o} clicked")).pack()
```

Regular User search

The userSearch page allows regular users to search for items in "userData.csv" file by ID, Name, or Producer and display the results in a list box. Users can also add selected items to a favorites list, which is saved in favorites.csv. It includes buttons for performing a search, adding items to favorites, clearing the search, and returning to the main menu.

Figure 17 regular user search.



View favorites:

The view favorites page lets users view the items saved by product name, product ID, and product producer during the regular user search. The user is initially prompted by the program to either confirm that they wish to view their favorite products or to return to the main menu. Currently, the main menu function only breaks the program but will return the user to the main menu when we update it in phase 4.

Figure 18 viewFavorites function

```
'''
Title: View Favorite List
Author: Ryan Taylor
Date: 11/17/2024
'''
# import packages
import csv
import os
import tkinter as tk
from tkinter import ttk


filename= 'Favorites.csv'

# create the csv file for testing & used the same data
def createCSV():
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        fields = ['Product ID','Product Name','Product Producer']
        writer.writerow(fields)
        writer.writerow(["12345", "Hamburger", "McDonalds"])
        writer.writerow(["54321", "Book", "ABC Inc"])
        writer.writerow(["24601", "CD", "XYZ Corp"])

# GUI setup
root= tk.Tk()
root.title("View Favorites")
root.geometry("400x300")
root.columnconfigure(0,weight=1)
root.rowconfigure(0,weight=1)

def viewFavorites():
    # create treeview widget
    info_tree = ttk.Treeview(root, columns=("Product ID", "Product Name", "Product Producer"), show="headings")
    info_tree.grid(row=1, column=0, columnspan=2, sticky="ew")
    # columns and headings
    info_tree.heading("Product ID", text="Product ID")
    info_tree.column("Product ID", width=100)
    info_tree.heading("Product Name", text="Product Name")
    info_tree.column("Product Name", width=150)
    info_tree.heading("Product Producer", text="Product Producer")
    info_tree.column("Product Producer", width=150)
    # read the csv and add into the treeview
    with open(filename, mode='r') as csvfile:
        reader = csv.reader(csvfile)
        next(reader)
        for row in reader:
            info_tree.insert("",tk.END,values=row)

# function to main menu but this is a place holder
def mainMenu():
    root.destroy()

btn_show = tk.Button(root, text="View Favorites", command= viewFavorites)
btn_show.grid(row=0, column=0, pady=10, padx=10)

btn_return = tk.Button(root, text="Return to Main Menu", command= mainMenu) # main menu placeholder
btn_return.grid(row=0, column=1, pady=10, padx=10)

createCSV()

root.mainloop()
```

User Buy/Borrow

For user simplicity, this function is within the greater search function. A user is presented with an option to add a selected item to their buy or borrow list. They are then asked if they would like to buy or borrow it, and if they wish to borrow it, they are asked the duration of the loan. These results will then be written to 2 files, the borrow list that can be viewed by the admin, and the history list that is viewable by the user.

Figure 19, buy and borrow functions

```
+ def requestToBuy(item=None):
+     try:
+         if item is None:
+             selected_item = resultsList.get(tk.ACTIVE)
+             if not selected_item:
+                 messagebox.showwarning("Error", "Please select an item to request to buy or borrow")
+                 return
+             item = dict(zip(["ID", "Name", "Producer"], [i.split(": ")[1] for i in selected_item.split(", ")]))
+         fileExists = os.path.exists("BorrowList.csv")
+         with open("BorrowList.csv", "a", newline="") as file:
+             write = csv.writer(file)
+             if not fileExists:
+                 write.writerow(["ID", "Name", "Producer", "Buy or borrow"])
+             write.writerow([item["ID"], item["Name"], item["Producer"], "Buy"])
+         # this segment adds a copy to the user history list
+         with open("UserHistory.csv", "a", newline="") as file:
+             write = csv.writer(file)
+             if not fileExists:
+                 write.writerow(["ID", "Name", "Producer",])
+             write.writerow([item["ID"], item["Name"], item["Producer"]])
+         messagebox.showinfo("Buy/Borrow list", "Item requested!")
+     except Exception as errorName:
+         messagebox.showerror("Error", f"An error occurred: {errorName}")
+
+ def requestToBorrow(item=None):
+     duration = simpledialog.askstring("How long would you like to borrow this item? (please enter a # of days")
+     try:
+         if item is None:
+             selected_item = resultsList.get(tk.ACTIVE)
+             if not selected_item:
+                 messagebox.showwarning("Error", "Please select an item to request to buy or borrow")
+                 return
+             item = dict(zip(["ID", "Name", "Producer"], [i.split(": ")[1] for i in selected_item.split(", ")]))
+         fileExists = os.path.exists("BorrowList.csv")
+         with open("BorrowList.csv", "a", newline="") as file:
+             write = csv.writer(file)
+             if not fileExists:
+                 write.writerow(["ID", "Name", "Producer", "Buy or borrow", "Borrow duration"])
+             write.writerow([item["ID"], item["Name"], item["Producer"], "Borrow", (str(duration))])
+         with open("UserHistory.csv", "a", newline="") as file:
+             write = csv.writer(file)
+             if not fileExists:
+                 write.writerow(["ID", "Name", "Producer",])
+             write.writerow([item["ID"], item["Name"], item["Producer"]])
+         messagebox.showinfo("Buy/Borrow list", "Item requested!")
+     except Exception as errorName:
+         messagebox.showerror("Error", f"An error occurred: {errorName}")
+
```

View History

The view history page is presented in a tree view, where users see a copy of all items they've requested to borrow or buy. There is nothing to do on this page other than view the history. There is a button to return to the main menu. The code creates a CSV file for testing purposes – when the code is put together, the writing of this CSV file will occur when the user requests to borrow or buy an item.

Figure 20, User view history.

```
'''
User_View_History
Version 2
Thomas Weston
11/17/2024
'''

import csv
import tkinter as tk
from tkinter import ttk


# writing to file for testing purposes
filename = "UserHistory.csv"

with open(filename, 'w', newline='') as file:
    writer = csv.writer(file)
    fields = ['ID','Name','Producer']

    writer.writerow(["1", "Hamburger", "McDonalds"])
    writer.writerow(["2", "Book", "ABC Inc"])
    writer.writerow(["3", "CD", "XYZ Corp"])



root = tk.Tk()
root.title("User view history")
root.geometry("500x500")

tree = ttk.Treeview(root)

tree['columns'] = (fields)
# columns
tree.column("#0", width=0, minwidth=0)
tree.column("ID", anchor="w", width=120, minwidth=25)
tree.column("Name", anchor="center", width=120, minwidth=25)
tree.column("Producer", anchor="w", width=120, minwidth=25)

#headings
tree.heading("#0", text="", anchor="w")
tree.heading("ID", text="Product ID", anchor="w")
tree.heading("Name", text="Product Name", anchor="center")
tree.heading("Producer", text="Product Producer", anchor="w")

# reading the file and inputting data
with open(filename, 'r',) as file:
    reader = csv.reader(file)
    for row in reader:
        tree.insert("", "end", values=(row))

tree.grid(row=0, column=0)

#buttons and functions for returning to main menu
def mainMenu():
    root.destroy()

#obviously this will redirect in the full version and just terminate.

btn_return = tk.Button(root, text="Return to Main Menu", command= mainMenu) #temp command for testing purposes
btn_return.grid(row=1, column=0)

root.mainloop()
```

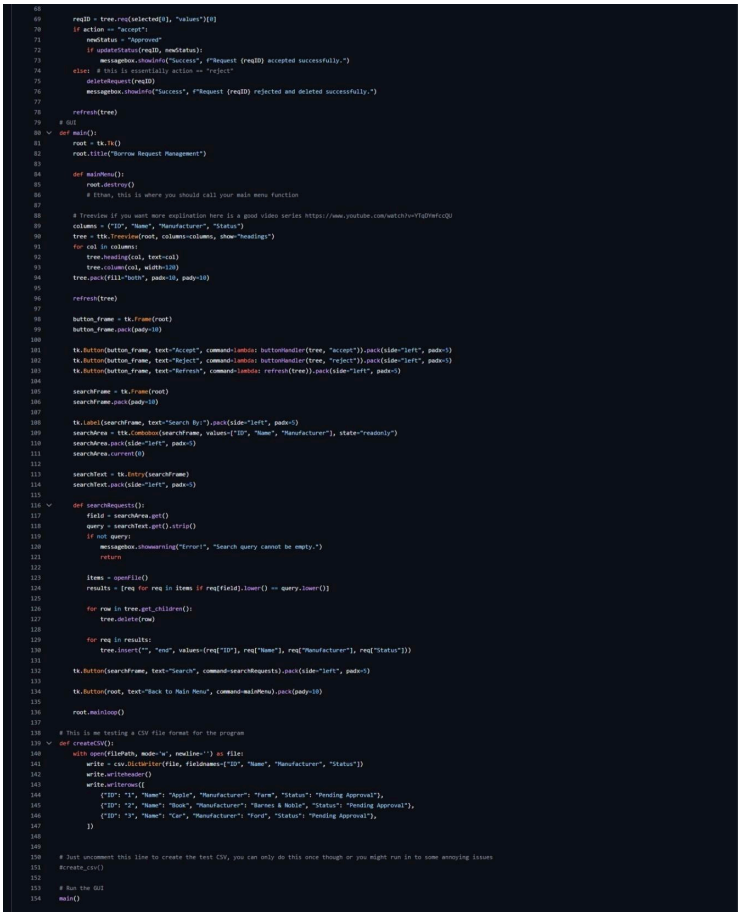
Admin View borrow requests

The Admin view Borrow Requests page presents a tree view where administrators can see all borrow requests from a CSV file called "borrowRequests.csv". Admins can accept or reject requests to borrow from the application. It includes buttons for accepting, rejecting, and refreshing the list of requests, as well as a search functionality to filter requests by ID, Name, or Manufacturer. There is also a button to return to the main menu.

Figure 21 first half of Admin View Borrow code

```
'''
Title: Admin view Borrow Requests
Author: Zachary Outman
Date: 11/17/2024
'''
import tkinter as tk
from tkinter import ttk, messagebox
import csv

# This is the file name for the borrow requests don't change it without changing the file name (Also the create_csv(): function will need to be updated if you didn't make the csv yet)
filePath = "borrowRequests.csv"

# Function to load all items from the file
def openFile():
    items = []
    try:
        with open(filePath, mode='r') as file:
            read = csv.DictReader(file)
            for i in read:
                items.append(i)
    except FileNotFoundError:
        messagebox.showerror("File Not found", "file not found. Contact cacheMeIfYouCan for help.")
    return items

# Function to update the status of an req
def updateStatus(reqID, newStatus):
    items = openFile()
    updated = False

    with open(filePath, mode='w', newline='') as file:
        write = csv.DictWriter(file, fieldnames=["ID", "Name", "Manufacturer", "Status"])
        write.writeheader()

        for req in items:
            if req['ID'] == reqID:
                req['Status'] = newStatus
                updated = True
            write.writerow(req)

    return updated

# Function to delete a rejected request in the file
def deleteRequest(reqID):

    with open(filePath, mode='w', newline='') as file:
        write = csv.DictWriter(file, fieldnames=["ID", "Name", "Manufacturer", "Status"])
        write.writeheader()

        for req in items:
            if req['ID'] != reqID or req['Status'] != "Rejected":
                write.writerow(req)

# Live refresh the treeview if edits are made to the csv file. Also this is where I found how to refresh because I forgot: https://stackoverflow.com/questions/76407618/how-to-refresh-data-i
def refresh(tree):
    for row in tree.get_children():
        tree.delete(row)

    items = openFile()
    for req in items:
        tree.insert("", "end", values=(req["ID"], req["Name"], req["Manufacturer"], req["Status"]))


def buttonHandler(tree, action):
    selected = tree.selection()
    if not selected:
        messagebox.showwarning("Error!", "Please select an req to perform this action.")
        return

    reqID = tree.req(selected[0], "values")[0]
    if action == "accept":
        newStatus = "Approved"
        if updateStatus(reqID, newStatus):
            messagebox.showinfo("Success", f"Request {reqID} accepted successfully.")
    else:  # this is essentially action == "reject"
        deleteRequest(reqID)
        messagebox.showinfo("Success", f"Request {reqID} rejected and deleted successfully.")

    refresh(tree)
# GUI
```

Figure 22, second half of admin view borrow code

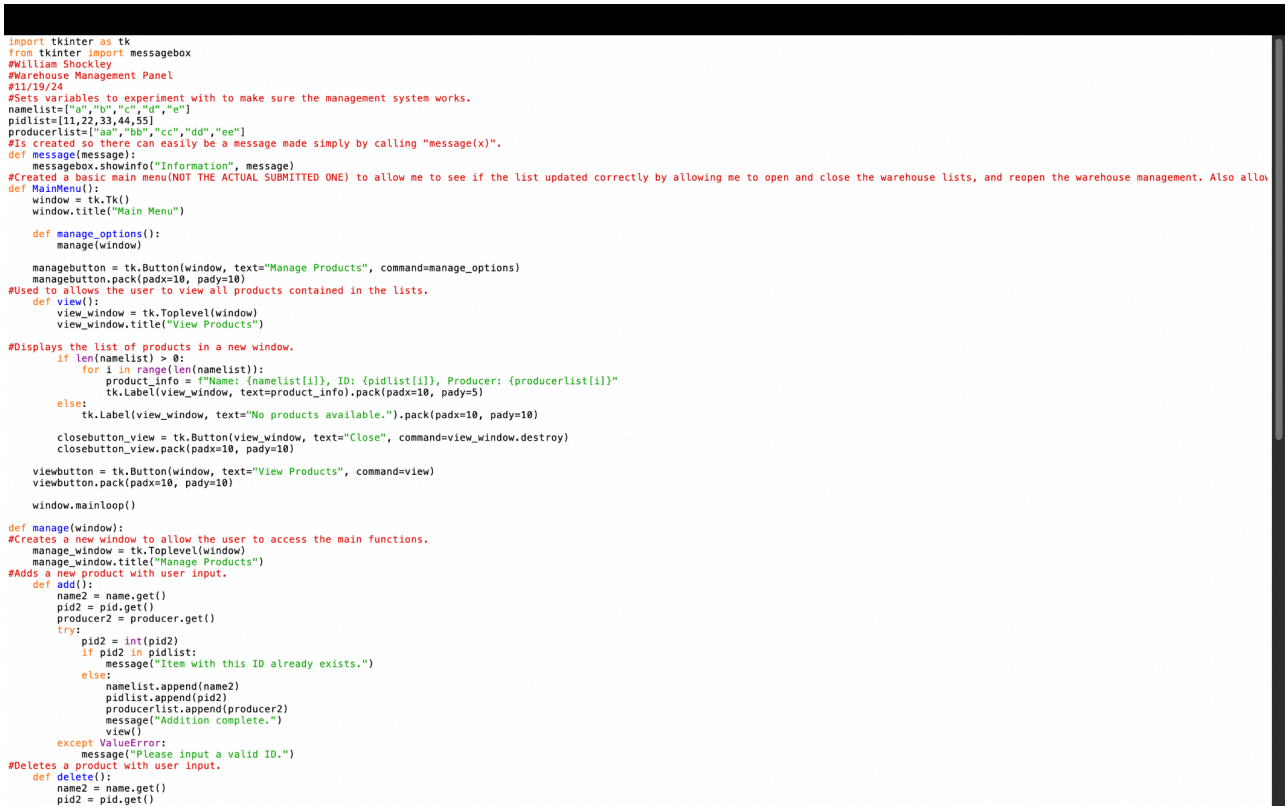


```
    reqID = tree.req(selected[0], "values")[0]
    if action == "accept":
        newStatus = "Approved"
        if updateStatus(reqID, newStatus):
            messagebox.showinfo("Success", f"Request {reqID} accepted successfully.")
    else:  # this is essentially action == "reject"
        deleteRequest(reqID)
        messagebox.showinfo("Success", f"Request {reqID} rejected and deleted successfully.")

    refresh(tree)
# GUI
def main():
    root = tk.Tk()
    root.title("Borrow Request Management")

    def mainMenu():
        root.destroy()
        # Ethan, this is where you should call your main menu function

    # Treeview if you want more explination here is a good video series https://www.youtube.com/watch?v=YTqDYmFccQU
    columns = ("ID", "Name", "Manufacturer", "Status")
    tree = ttk.Treeview(root, columns=columns, show="headings")
    for col in columns:
        tree.heading(col, text=col)
        tree.column(col, width=130)
    tree.pack(fill="both", pady=10, padx=10)

    refresh(tree)

    button_frame = tk.Frame(root)
    button_frame.pack(pady=10)

    tk.Button(button_frame, text="Accept", command=lambda: buttonHandler(tree, "accept")).pack(side="left", padx=5)
    tk.Button(button_frame, text="Reject", command=lambda: buttonHandler(tree, "reject")).pack(side="left", padx=5)
    tk.Button(button_frame, text="Refresh", command=lambda: refresh(tree)).pack(side="left", padx=5)

    searchFrame = tk.Frame(root)
    searchFrame.pack(pady=10)

    tk.Label(searchFrame, text="Search By:").pack(side="left", padx=5)
    searchArea = ttk.Combobox(searchFrame, values=["ID", "Name", "Manufacturer"], state="readonly")
    searchArea.pack(side="left", padx=5)
    searchArea.current(0)

    searchText = tk.Entry(searchFrame)
    searchText.pack(side="left", padx=5)

    def searchRequests():
        field = searchArea.get()
        query = searchText.get().strip()
        if not query:
            messagebox.showwarning("Error!", "Search query cannot be empty.")
            return

        items = openFile()
        results = [req for req in items if req[field].lower() == query.lower()]

        for row in tree.get_children():
            tree.delete(row)

        for req in results:
            tree.insert("", "end", values=(req["ID"], req["Name"], req["Manufacturer"], req["Status"]))

    tk.Button(searchFrame, text="Search", command=searchRequests).pack(side="left", padx=5)

    tk.Button(root, text="Back to Main Menu", command=mainMenu).pack(pady=10)

    root.mainloop()

# This is me testing a CSV file format for the program
def createCSV():
    with open(filePath, mode='w', newline='') as file:
        write = csv.DictWriter(file, fieldnames=["ID", "Name", "Manufacturer", "Status"])
        write.writeheader()
        write.writerows([
            {"ID": "1", "Name": "Apple", "Manufacturer": "Farm", "Status": "Pending Approval"},
            {"ID": "2", "Name": "Book", "Manufacturer": "Barnes & Noble", "Status": "Pending Approval"},
            {"ID": "3", "Name": "Car", "Manufacturer": "Ford", "Status": "Pending Approval"},
        ])


# Just uncomment this line to create the test CSV, you can only do this once though or you might run in to some annoying issues
#create_csv()

# Run the GUI
main()
```

Admin manage warehouse

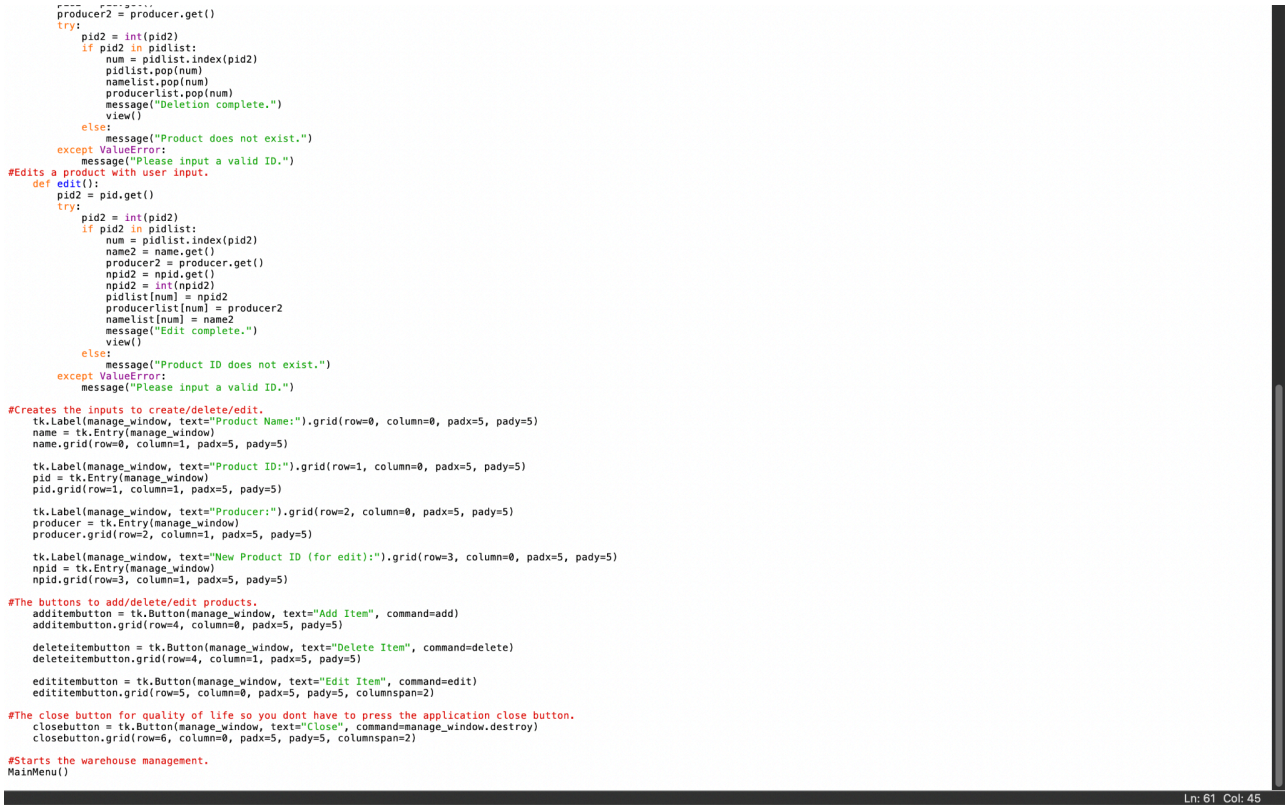
On program start it opens an example main menu page on program start allowing you to view a basic product list and open the warehouse management panel if you are an admin. Then if you open the warehouse management page you see four columns, product ID, product name, product producer, and new product ID for the warehouse edit function. You can enter any product details to add a product and then press add if a product with the same ID does not exist and the given ID is an integer. To delete you enter the product details including the product ID, name, and producer, and it will delete the product from the warehouse. However, if a product with those details does not exist a message saying such will display. To edit an item you enter the new name in the product name entry, put the product ID of the product you want to change in the product ID entry, enter the new producer, and finally, enter the new product ID. Also after each function the new updated list is displayed.

Figure 23, first half of Admin Manage Warehouse.



```
import tkinter as tk
from tkinter import messagebox
#William Shockley
#Warehouse Management Panel
#11/19/24
#Sets variables to experiment with to make sure the management system works.
namelist=["a","b","c","d","e"]
pidlist=[11,22,33,44,55]
producerlist=["aa","bb","cc","dd","ee"]
#Is created so there can easily be a message made simply by calling "message(x)".
def message(message):
    messagebox.showinfo("Information", message)
#Created a basic main menu(NOT THE ACTUAL SUBMITTED ONE) to allow me to see if the list updated correctly by allowing me to open and close the warehouse lists, and reopen the warehouse management. Also allo
def MainMenu():
    window = tk.Tk()
    window.title("Main Menu")

    def manage_options():
        manage(window)

    managebutton = tk.Button(window, text="Manage Products", command=manage_options)
    managebutton.pack(padx=10, pady=10)
#Used to allows the user to view all products contained in the lists.
    def view():
        view_window = tk.Toplevel(window)
        view_window.title("View Products")

#Displays the list of products in a new window.
        if len(namelist) > 0:
            for i in range(len(namelist)):
                product_info = f"Name: {namelist[i]}, ID: {pidlist[i]}, Producer: {producerlist[i]}"
                tk.Label(view_window, text=product_info).pack(padx=10, pady=5)
        else:
            tk.Label(view_window, text="No products available.").pack(padx=10, pady=10)

        closebutton_view = tk.Button(view_window, text="Close", command=view_window.destroy)
        closebutton_view.pack(padx=10, pady=10)

    viewbutton = tk.Button(window, text="View Products", command=view)
    viewbutton.pack(padx=10, pady=10)

    window.mainloop()

def manage(window):
#Creates a new window to allow the user to access the main functions.
    manage_window = tk.Toplevel(window)
    manage_window.title("Manage Products")
#Adds a new product with user input.
    def add():
        name2 = name.get()
        pid2 = pid.get()
        producer2 = producer.get()
        try:
            pid2 = int(pid2)
            if pid2 in pidlist:
                message("Item with this ID already exists.")
            else:
                namelist.append(name2)
                pidlist.append(pid2)
                producerlist.append(producer2)
                message("Addition complete.")
                view()
        except ValueError:
            message("Please input a valid ID.")
#Deletes a product with user input.
    def delete():
        name2 = name.get()
        pid2 = pid.get()
```

Figure 24, second half of Admin Manage Warehouse



```
        producer2 = producer.get()
        try:
            pid2 = int(pid2)
            if pid2 in pidlist:
                num = pidlist.index(pid2)
                pidlist.pop(num)
                namelist.pop(num)
                producerlist.pop(num)
                message("Deletion complete.")
                view()
            else:
                message("Product does not exist.")
        except ValueError:
            message("Please input a valid ID.")
#Edits a product with user input.
    def edit():
        pid2 = pid.get()
        try:
            pid2 = int(pid2)
            if pid2 in pidlist:
                num = pidlist.index(pid2)
                name2 = name.get()
                producer2 = producer.get()
                npid2 = npid.get()
                npid2 = int(npid2)
                pidlist[num] = npid2
                producerlist[num] = producer2
                namelist[num] = name2
                message("Edit complete.")
                view()
            else:
                message("Product ID does not exist.")
        except ValueError:
            message("Please input a valid ID.")

#Creates the inputs to create/delete/edit.
    tk.Label(manage_window, text="Product Name:").grid(row=0, column=0, padx=5, pady=5)
    name = tk.Entry(manage_window)
    name.grid(row=0, column=1, padx=5, pady=5)

    tk.Label(manage_window, text="Product ID:").grid(row=1, column=0, padx=5, pady=5)
    pid = tk.Entry(manage_window)
    pid.grid(row=1, column=1, padx=5, pady=5)

    tk.Label(manage_window, text="Producer:").grid(row=2, column=0, padx=5, pady=5)
    producer = tk.Entry(manage_window)
    producer.grid(row=2, column=1, padx=5, pady=5)

    tk.Label(manage_window, text="New Product ID (for edit):").grid(row=3, column=0, padx=5, pady=5)
    npid = tk.Entry(manage_window)
    npid.grid(row=3, column=1, padx=5, pady=5)

#The buttons to add/delete/edit products.
    additembutton = tk.Button(manage_window, text="Add Item", command=add)
    additembutton.grid(row=4, column=0, padx=5, pady=5)

    deleteitembutton = tk.Button(manage_window, text="Delete Item", command=delete)
    deleteitembutton.grid(row=4, column=1, padx=5, pady=5)

    edititembutton = tk.Button(manage_window, text="Edit Item", command=edit)
    edititembutton.grid(row=5, column=0, padx=5, pady=5, columnspan=2)

#The close button for quality of life so you dont have to press the application close button.
    closebutton = tk.Button(manage_window, text="Close", command=manage_window.destroy)
    closebutton.grid(row=6, column=0, padx=5, pady=5, columnspan=2)

#Starts the warehouse management.
MainMenu()
```

Ln: 61 Col: 45

Admin remove user

Open an example main menu where you can open the user removal page. On the user removal page, you can see the names of the current users by pressing the “View Users” button, or you can enter a username and password to remove any non-admin user. If a user is an admin it will return that an admin cannot be removed. The password you enter is shown as * for privacy purposes, it will return if a user does not exist or if the password you entered is incorrect. If you enter a guest user's name and password it will return that the user was removed and they will no longer be visible on the userlist.

Figure 25, Admin remove user.

```
        removebutton = tk.Button(admin, text="Remove Guest", command=remove)
        removebutton.pack(padx=10, pady=10)

        # View users button
        viewbutton = tk.Button(admin, text="View Users", command=view)
        viewbutton.pack(padx=10, pady=10)

        # Close button
        closebutton = tk.Button(admin, text="Close", command=admin.destroy)
        closebutton.pack(padx=10, pady=10)

    # Manage user tab which only allows the removal of users in this program
    adminbutton = tk.Button(window, text="Manage Users", command=management)
    adminbutton.pack(padx=10, pady=10)

    window.mainloop()

# Starts the program
MainMenu()
```

Ln: 61 Col: 16

Admin Create Guest User: The *Create Guest User* function intends to create a feature in which a non-admin can create an account to access the warehouse. The entirety of the function operates cohesively. By Utilizing an If, Else Method, the program systematically ensures that the username and password entered by the “guest” have not previously been created. The program can display message boxes after it is provided with the “Tkinter” module.

Figure 26, First Half of Admin Create Guest User.

```
    else:
        user_database[username] = password
        save_user_to_csv(username, password)
        messagebox.showinfo("Success", "New Guest User Successfully Created!")
        username_entry.delete(0, tk.END)
        password_entry.delete(0, tk.END)

def display_user_count():
    user_count = len(user_database)
    messagebox.showinfo("User Count", f"There are currently {user_count} guest users.")

def display_date():
    current_date = datetime.datetime.now().strftime("%Y-%m-%d")
    messagebox.showinfo("Current Date", f"Today's date is: {current_date}")

# This is where main menu is called
def main_menu():
    root.destroy()

# Opens the main window
root = tk.Tk()
root.title("Guest User Creation")

# Creates widgets
tk.Label(root, text="Enter a Username:").grid(row=0, column=0, padx=10, pady=5, sticky="e")
username_entry = tk.Entry(root, width=30)
username_entry.grid(row=0, column=1, padx=10, pady=5)

tk.Label(root, text="Enter a Password:").grid(row=1, column=0, padx=10, pady=5, sticky="e")
password_entry = tk.Entry(root, show="*", width=30)
password_entry.grid(row=1, column=1, padx=10, pady=5)

create_button = tk.Button(root, text="Create Guest User", command=create_guest_user)
create_button.grid(row=2, column=0, columnspan=2, pady=10)

user_count_button = tk.Button(root, text="Show User Count", command=display_user_count)
user_count_button.grid(row=3, column=0, columnspan=2, pady=10)

date_button = tk.Button(root, text="Show Current Date", command=display_date)
date_button.grid(row=4, column=0, columnspan=2, pady=10)
```

Figure 27, Second Half of Admin Create Guest User.

```
# Admin Create Guest User
# Simple Program to Create a Guest User within a Warehouse Management System
# Jack Teller


import tkinter as tk
from tkinter import messagebox
import csv
import datetime

# Path to the CSV file
guestFile = "user_date.csv"

# Load user data from the CSV file
def load_users_from_csv():
    try:
        with open(guestFile, mode="r") as file:
            reader = csv.reader(file)
            for row in reader:
                if len(row) == 2:  # Ensures rows have both username and password
                    user_database[row[0]] = row[1]
    except FileNotFoundError:
        with open(guestFile, mode="w") as file:  # Creates the file if it doesn't exist
            pass

# Save user data to the CSV file
def save_user_to_csv(username, password):
    with open(guestFile, mode="a", newline="") as file:
        writer = csv.writer(file)
        writer.writerow([username, password])

# User database
user_database = {}
load_users_from_csv()

def create_guest_user():
    username = username_entry.get().strip()
    password = password_entry.get().strip()

    if not username or not password:
        messagebox.showwarning("Input Error", "Username and password cannot be empty.")
        return


main_menu_button = tk.Button(root, text="Main Menu", command=main_menu)
main_menu_button.grid(row=5, column=0, columnspan=2, pady=10)

# Runs the main event loop
root.mainloop()
```

Ln: 31 Col: 0

Data Storage:

Login Page:
This code creates a CSV file (`users.csv`) and then uses it to store user credentials. Each line in the file represents a username and password pair, separated by a comma (e.g., `admin,1234`). The program reads this data into a Python dictionary (`users`) when it starts, allowing quick in-memory access to validate logins. New users are added both to the dictionary and appended to the CSV file during registration. This method is simple, readable, and portable and since the code creates the file if it doesn't already exist, there will be no issues running this on a computer that doesn't already have profiles saved on it.

Admin view Borrow Requests:
This code creates a CSV file (`borrowRequests.csv`) to store borrow requests, each containing ID, Name, Manufacturer, and Status. The program reads this data into a list of dictionaries for in-memory management. Administrators can update request statuses or delete rejected requests, with changes written back to the CSV file. The list of requests can be refreshed and searched by ID, Name, or Manufacturer. The file is created if it doesn't exist, ensuring portability.

User Search:
This code creates a CSV file (User data.csv) to store user data with fields for ID, Name, and Producer. The program reads this data into a list of dictionaries for in-memory searching. Users can search for items by ID, Name, or Producer and display the results. Items can also be added to a favorites list, which is saved in a separate CSV file (favorites.csv). The favorites file is created if it doesn't exist, ensuring portability.

View Favorites Page:
This code reads the CSV file ('Favorites.csv') that was created within the regular user search function when a user is prompted to add specific products to their favorite products before returning to the search function. The data is presented in a table that includes the product's name, ID, and producer. Products are only added within the regular user search function and cannot be edited within the view favorites function.

User Buy/Borrow request function:
These functions add a copy of the data presented in the overall list to two CSV files ('UserHistory.csv' and 'BorrowList.csv') The first file ('UserHistory.csv') cannot be edited and is presented on the View History page. This is so the user can view a history of all the items they have requested at any given time. The second file ('BorrowList.csv') is viewed by the page admin in the Admin View Requests page, where the admin can approve or deny these requests.

View History Page:
This page reads the CSV file ('UserHistory.csv') that is created within the search function any time a user requests to buy or borrow an item from the warehouse. Each line in this file represents the ID of a product, the name of the product, and the producer of the product, each separated by a comma. It is presented in a table on the view history page, and entries cannot be manually added or removed by the user. The file is only edited within the search function when a user requests to buy or borrow an item. This data

Admin Create Guest User Page:
This function adds the user information data (username and password) and stores it in the dictionary *user_database.* When the function is called, it pulls the previous users' information to ensure that the new username and password have not been used prior. It then adds a new key-value pair to the dictionary (The key being the username and the value being the password). All of this ensures the uniqueness of the guest users' information.

Admin Remove User:
Usernames, roles, and passwords are stored in 3 lists: usernames, userroles, and userpasswords. The admin user can select one of these items to be removed from the list. These items are added to the list when the admin creates a guest user. It is stored in regular lists, and not a CSV file, because we had problems in editing the file. For now, this data storage is a temporary measure. We plan on implementing CSV storage with this file as one of our improvements in the next phase.

Admin Manage Warehouse:
The data is stored in lists titled namelist, pidlist, and producer list. These lists can be edited by the admin user. It is stored in regular lists, and not a CSV file, because we had problems in writing the file. For now, this data storage is a temporary measure. We plan on implementing CSV storage with this file as one of our improvements in the next phase.

Improvements For Main Menu: I added a logout function that allows you to log out of your account without closing the entire software. I also added a display profile function that displays the profile information. It shows you not only what account you're currently logged into but also the type of account so that you know what kind of permission that you have access to. Finally, I added a function that allows you to access the log of accounts, from the admin account, for who has logged in so that you can keep track of who has logged into the system.

Figure 29, addMenuFeatures

```
def addMenuFeatures(root, userType, currentUser):
    if userType == "admin":
        tk.Button(root, text="Log Out", command=lambda: logout(root)).pack()
        tk.Button(root, text="Display Profile", command=lambda: displayProfile(currentUser)).pack()
        tk.Button(root, text="View Login Log", command=lambda: viewLoginLog(root)).pack()
    else:
        tk.Button(root, text="Log Out", command=lambda: goBackToLogin(root)).pack()
        tk.Button(root, text="Display Profile", command=lambda: displayProfile(currentUser)).pack()
```

Figure 30, Logout and displayProfile

```
def logout(root):
    for widget in root.winfo_children():
        widget.destroy()
    setupGui(root)

def displayProfile(currentUser):
    userType = "Admin" if currentUser == "admin" else "User"
    messagebox.showinfo("Profile Info", f"Username: {currentUser}\nType: {userType}")
```

Figure 31, viewLoginLog

```
def viewLoginLog(root):
    # Open a window to view login logs
    if not os.path.exists(loginLogFile):
        messagebox.showinfo("Info", "No login logs available.")
        return

    logWin = tk.Toplevel(root)
    logWin.title("Login Log")
    logWin.geometry("500x300")
    tk.Label(logWin, text="Login Log").pack()

    logText = tk.Text(logWin, wrap=tk.WORD, state=tk.DISABLED, width=60, height=15)
    logText.pack()

    try:
        with open(loginLogFile, "r") as f:
            logData = f.readlines()

        logText.config(state=tk.NORMAL)
        logText.delete(1.0, tk.END)
        for entry in logData:
            logText.insert(tk.END, entry)
        logText.config(state=tk.DISABLED)
    except Exception as e:
        messagebox.showerror("Error", f"Error reading login log: {e}")
```

Improvements for Login Page: I added a reset password so that a user can reset their password if they choose to or need to. I also added a login attempt counter so that after 3 attempts, the system locks you from logging into that account. Finally, I added a log that goes through and keeps every login attempt and what account was logged into or out of.

Figure 32, resetPassword

```
def resetPassword(root):
    def updatePassword():
        username = usernameEntry.get().strip()
        oldPassword = oldPwEntry.get().strip()
        newPassword = newPwEntry.get().strip()

        if username not in users or users[username] != oldPassword:
            messagebox.showerror("Error", "Invalid username or old password.")
        else:
            users[username] = newPassword
            saveUser(username, newPassword)
            messagebox.showinfo("Success", "Password updated successfully!")
            resetWin.destroy()

    resetWin = tk.Toplevel(root)
    resetWin.title("Reset Password")
    resetWin.geometry("300x200")
    tk.Label(resetWin, text="Username").pack()
    usernameEntry = tk.Entry(resetWin)
    usernameEntry.pack()
    tk.Label(resetWin, text="Old Password").pack()
    oldPwEntry = tk.Entry(resetWin, show="*")
    oldPwEntry.pack()
    tk.Label(resetWin, text="New Password").pack()
    newPwEntry = tk.Entry(resetWin, show="*")
    newPwEntry.pack()
    tk.Button(resetWin, text="Update Password", command=updatePassword).pack()
```

Figure 33, limitLoginAttempts

```
def limitLoginAttempts(user):
    if user not in loginAttempts:
        loginAttempts[user] = 0
    loginAttempts[user] += 1

    if loginAttempts[user] >= MAX_ATTEMPTS:
        blockedUsers[user] = True  # Block user after MAX_ATTEMPTS
        messagebox.showerror("Error", "Too many failed login attempts! Your account is now blocked.")
        return False
    return True
```

Figure 34, logLogins

```
def logLogin(user, success):
    try:
        with open(loginLogFile, "a", newline="") as file:
            writer = csv.writer(file)
            writer.writerow([user, "Success" if success else "Failure", datetime.now()])
    except Exception as e:
        messagebox.showerror("Error", f"Error logging login: {e}")
```

Improvements to the View Favorites Page:

Through brainstorming and discussing with my group, we identified the three additional features for the view favorites page: removeFavorites, searchFavorites, and sortFavorites. Since there is a way to add products to the list of favorites, there should be a way to remove them too. The removeFavorites function allows the user to manually select the product in the list of favorites and press the “Remove Selected Product” button. The user will be prompted to confirm the removal, and if no product was selected, the user will be prompted to choose one before trying to remove it. The searchFavorites and sortFavorites are helpful for the user to find their desired item faster. There is a search widget for the user to enter the name, ID, or producer of the product they wish to find and by clicking the headings, the list will be sorted alphabetically or numerically.

Figure 35, Remove Favorites

```
65  def removeFavorites():
66      # get selected item
67      select_item = info_tree.selection()
68      if not select_item:
69          messagebox.showwarning("No Selection", "Please select a row to remove.")
70          return
71      # get the values of the selected items
72      select_values = info_tree.item(select_item, "values")
73      confirm = messagebox.askyesno("Confirm Removal", f"Are you sure you want to remove?")
74      info_tree.delete(select_item)
75      # update csv file
76      with open(filename, mode='r') as csvfile:
77          rows = list(csv.reader(csvfile))
78      #filter out row to be removed
79      rows= [row for row in rows if row != list(select_values)]
80      #rewrite csv file
81      with open(filename, mode='w', newline='') as csvfile:
82          writer = csv.writer(csvfile)
83          writer.writerows(rows)
```

Figure 36, Search Favorites

```
87  def searchFavorites():
88      search_query = search_entry.get().lower()
89      if not search_query:
90          messagebox.showwarning("Empty Search", "Please enter a search term.")
91          return
92      clearTreeview()
93      # read the csv file and display matching rows
94      with open(filename, mode='r') as csvfile:
95          reader = csv.reader(csvfile)
96          next(reader)
97          for row in reader:
98              if any(search_query in cell.lower() for cell in row):
99                  info_tree.insert("", tk.END, values=row)
100
```

Figure 37, Sort Favorites

```
101 # global variable to track sorting state
102 sort_order = {"Product ID": False, "Product Name": False, "Product Producer": False}
103
104 def sortFavorites(column):
105     data = [info_tree.item(child, "values") for child in info_tree.get_children()]
106     column_index = ["Product ID", "Product Name", "Product Producer"].index(column)
107     reverse = sort_order[column]
108     sort_order[column] = not reverse
109     if column == "Product ID":
110         data.sort(key=lambda x: int(x[column_index]) if x[column_index].isdigit() else 0, reverse=reverse)
111     else:
112         data.sort(key=lambda x: x[column_index].lower(), reverse=reverse)
113     clearTreeview()
114     for row in data:
115         info_tree.insert("", tk.END, values=row)
```

Improvements to the User Buy/Borrow functions:

Within the buy and borrow commands, the following improvements were made;

First, the two buy and borrow buttons were merged into just one function. The user is prompted with a pop-up window to choose whether to buy or borrow an item. Depending on their answer to the question, the variable “BorB” (which stands for Buy or Borrow), which is reflected by a line in the CSV noting if the item was requested “buy” or “borrow”.

The next improvement is that it is now easier for the user to note the duration of their requested borrow. If the user selects Borrow, they are met with another prompt to enter the number of days they wish to borrow the item for. This is reflected in the variable “DurB” (which stands for Duration of Borrow.) If the user selects “Buy,” the DurB variable is set to “N/A”.

Finally, more information is noted for each borrow. The date, which is retrieved from the system time and is stored in the variable “dateB” (which stands for Date of Buy/Borrow.) This is called when the buy/borrow button is pressed, so it records only the time on the system. It also only records the date, and not the minutes or hour of the request. This is also added to the UserHistory.CSV, and is reflected in the user history page.

Figure 38, Improved version of the Buy/Borrow command.

```
def requestToB(item=None):
    try:
        if item is None:
            selected_item = resultsList.get(tk.ACTIVE)
            if not selected_item:
                messagebox.showwarning("Error", "Please select an item to request")
                return
            item = dict(zip(["ID", "Name", "Producer"], [i.split(": ")[1] for i in selected_item.split(", ")]))
        fileExists = os.path.exists("BorrowList.csv")
        # commands to select whether to buy or borrow an item
        BuyOrBorrowWindow()
        DateB = datetime.now()
        DateB = DateB.date()
        print(DateB)
        with open("BorrowList.csv", "a", newline="") as file:
            write = csv.writer(file)
            if not fileExists:
                write.writerow(["ID", "Name", "Producer", "Buy or borrow", "Borrow duration", "Date of request", "Approval by Admin"])
            write.writerow([item["ID"], item["Name"], item["Producer"], BorB, f"{DurB} days", DateB, "PENDING"])

        # this segment adds a copy to the user history list

        with open("UserHistory.csv", "a", newline="") as file:
            write = csv.writer(file)
            if not fileExists:
                write.writerow(["ID", "Name", "Producer", "Buy or borrow", "Borrow duration", "Date of request",])
            write.writerow([item["ID"], item["Name"], item["Producer"], BorB, f"{DurB} days", DateB,])

        messagebox.showinfo("Buy/Borrow list", "Item requested!")
        print(f'BorB is equal to "{BorB}"')
        print(f'DurB is equal to "{DurB}"')
    except Exception as errorName:
        messagebox.showerror("Error", f"An error occurred: {errorName}")
```

Figure 39, options window and subfunctions for the improved buy/borrow command.

```
# creating a custom class for a 2 option window.
class OptionsWindow(simpledialog.Dialog):
    def body(self, master):
        self.result= None
        tk.Label(master, text="Are you requesting this item to buy or to borrow?").grid(row=0)
        tk.Button(master, text="Buy", command=BuyCmd).grid(row=1)
        tk.Button(master, text="Borrow", command=BorrowCmd).grid(row=2)
        tk.Label(master, text="Press OK to confirm your choice!").grid(row=3)

# function to call the options window
def BuyOrBorrowWindow():
    OptionsWindow(root, title="Buy or Borrow this item?")


# fuctions for the buttons to buy or borrow -- and setting of the appropriate variables
def BuyCmd():
    global BorB
    global DurB
    BorB = "Buy"
    DurB = "N/A"
def BorrowCmd():
    global BorB
    global DurB
    BorB = "Borrow"
    DurB = simpledialog.askstring("Input","How many days would you like to borrow this item for?", parent = root)
```

Improvements to User View History:

On the view history page, the following improvements were made;

First, the information view was extended so that the user could view more specific information about their purchase history. The field was included to show whether or not the user had bought or borrowed an item, the duration of their borrow (if applicable), and the date on which they requested the item.

Next, a manual refresh button was added. This is so that the user can view updated information without having to close and reopen the page. The refresh function works by clearing the entire field, and then recalling the file.

Lastly, a delete function was added for users wishing to delete selected items from their history. It works by finding the ID of the selected item, filtering that item out of the list, and then rewriting the list. After all of this, the refresh function is called so that the user can see the change in real time.

Figure 40, Expanded fields for tree views

```
tree['columns'] = ['ID','Name','Producer','Buy or Borrow','Borrow Duration','Date of Request']
# columns
tree.column("#0", width=0, minwidth=0)
tree.column("ID", anchor="w", width=120, minwidth=25)
tree.column("Name", anchor="center", width=120, minwidth=25)
tree.column("Producer", anchor="w", width=120, minwidth=25)
tree.column("Buy or Borrow", anchor="w", width=120, minwidth=25)
tree.column("Borrow Duration", anchor="w", width=120, minwidth=25)
tree.column("Date of Request", anchor="w", width=120, minwidth=25)

#headings
tree.heading("#0", text="", anchor="w")
tree.heading("ID", text="Product ID", anchor="w")
tree.heading("Name", text="Product Name", anchor="center")
tree.heading("Producer", text="Product Producer", anchor="w")
tree.heading("Buy or Borrow", text="Buy or Borrow", anchor="w")
tree.heading("Borrow Duration", text="Borrow Duration", anchor="w")
tree.heading("Date of Request", text="Date of Request", anchor="e")
```

Figure 41, ListClear and ListRefresh functions

```
# Cearing the tree view (sub-function of refresh)
def ListClear():
    for item in tree.get_children():
        tree.delete(item)


# refreshing the tree view for edits made in other windows -- this is also called to instantly refresh the view after deleting an entry
def ListRefresh():
    ListClear()
    with open(filename, 'r',) as file:
        reader = csv.reader(file)
        for row in reader:
            tree.insert("", "end", values=(row))
```

Figure 42, Delete History function

```
def delHistory():
    select_row = tree.selection()[0] #selected row
    RIndex = tree.index(select_row)


    with open("UserHistory.csv","r") as file:
        reader = csv.reader(file)
        rows = list(reader)

    del rows[RIndex]

    # rewrites the file with
    with open("UserHistory.csv", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerows(rows)

    ListRefresh() #refreshes the list

# this line is for debugging -- uncomment it if you need to.
#    print(RIndex)

    messagebox.showinfo("History", "Item Deleted! Refresh List if change is not present")
```

Figure 43, Added buttons to support new functions.

```
tk.Label(root,text="Press refresh to see values.").grid(row=1, column=0)

btn_refresh = tk.Button(root, text="Refresh List", command= ListRefresh)
btn_refresh.grid(row=2, column=0)

btn_return = tk.Button(root, text="Return to Main Menu", command= mainMenu)
btn_return.grid(row=3, column=0)

btn_delete = tk.Button(root, text="Delete item from history", command= delHistory)
btn_delete.grid(row=4, column=0)
```

Improvements to The Create Guest User Page

For this page, we collectively decided on three improving functions: display_user_count, display_date, and mainMenu. The display user count function activates by simply utilizing the length function to view how many users are stored inside the user database. For display_date, the datetime module had to be imported into the code and activated using both .now and .strftime. Lastly, for mainMenu, we called Ethan's Main Menu function from his section of the code.

Figure 44, Display User Count

```
#Function which shows the # of guest users
def display_user_count():
    user_count = len(user_database)
    messagebox.showinfo("User Count", f"There are currently {user_count} guest users.")
```

Figure 45, Display Date

```
import datetime
#Function which displays current date
def display_date():
    current_date = datetime.datetime.now().strftime("%Y-%m-%d")
    messagebox.showinfo("Current Date", f"Today's date is: {current_date}")
```

Figure 46, Return to Main Menu

```
# This is where main menu is called
def main_menu():
    root.destroy()
```

Improvements to the Warehouse Management Function

For the warehouse management page, we changed how the data was stored from lists to CSV, has a feature to view the warehouse history, and a confirmation for deletion of products just in case of user error. Changing data storage from a list to a CSV file allows for easier integration. Also, allowing the warehouse history to be viewed is useful for admins to see what has been changed. Finally, there is now a confirmation feature to allow a user to make sure their input is correct.

Figure 47, CSV Storage

```
8   PRODUCT_CSV = 'products.csv'
9   HISTORY_CSV = 'product_history.csv'
10
11  # Ensure the CSV file and history file exist with proper headers if not already there
12  def makecsv():
13      if not os.path.exists(PRODUCT_CSV):
14          with open(PRODUCT_CSV, mode='w', newline='') as file:
15              writer = csv.writer(file)
16              writer.writerow(['Product Name', 'Product ID', 'Producer'])  # Write headers
17
18      if not os.path.exists(HISTORY_CSV):
19          with open(HISTORY_CSV, mode='w', newline='') as file:
20              writer = csv.writer(file)
21              writer.writerow(['Timestamp', 'Action', 'Product Name', 'Product ID', 'Producer'])
22
23  # Load product data from CSV
24  def load_data():
25      products = []
26      if os.path.exists(PRODUCT_CSV):
27          with open(PRODUCT_CSV, mode='r', newline='') as file:
28              reader = csv.reader(file)
29              next(reader)  # Skip the header row
30              for row in reader:
31                  products.append(row)
32      return products
33
34  # Write product data to CSV
35  def save_data(products):
36      with open(PRODUCT_CSV, mode='w', newline='') as file:
37          writer = csv.writer(file)
38          writer.writerow(['Product Name', 'Product ID', 'Producer'])  # Write headers
39          writer.writerows(products)
```

Figure 48, Warehouse History

```
42  def log_history(action, name, pid, producer):
43      timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
44      with open(HISTORY_CSV, mode='a', newline='') as file:
45          writer = csv.writer(file)
46          writer.writerow([timestamp, action, name, pid, producer])
47
```

Figure 49, Deletion Confirmation

```
132     def delete():
133         pid2 = pid.get()
134         try:
135             pid2 = int(pid2)
136             products = load_data()
137             for product in products:
138                 if int(product[1]) == pid2:
139                     if messagebox.askyesno("Confirm Deletion", f"Are you sure you want to delete {product[0]} (ID: {product[1]})?"):
140                         products.remove(product)
141                         save_data(products)
142                         log_history('Deleted', product[0], pid2, product[2])
143                         message("Deletion complete.")
144                     return
145             message("Product does not exist.")
146         except ValueError:
147             message("Please input a valid ID.")
```

Improvements to use Admin Remove User Page

For the user delete page, we added a feature to view the passwords of all users as the person using this function is an admin, and needs access to the passwords to delete a user and therefore can be trusted with the passwords. Also, we added a log to keep track of which users were deleted and when they were deleted using datetime. The final improvement was changing the way data was stored from lists to a csv file for better data storage, and it allows easier integration with new code.

Figure 50, Password Access

```
65      def viewpasswords():
66          password_list = "\n".join(userpasswords)
67          message(f"User   Passwords:\n{password_list}")
```

Figure 51, Removed Accounts Log

```
69  def removedlog():
70      if not os.path.exists(REMOVED_USERS_FILE):
71          message("No users have been removed yet.")
72          return
73
74      log_entries = []
75      with open(REMOVED_USERS_FILE, mode='r', newline='') as file:
76          reader = csv.reader(file)
77          for row in reader:
78              if row:  # Make sure the row is not empty
79                  log_entries.append(f":User  {row[0]}, Removed At: {row[1]}")
80
81      if log_entries:
82          log_message = "\n".join(log_entries)
83          message(f"Removed Users Log:\n{log_message}")
84      else:
85          message("No users have been removed yet.")
```

Figure 52, Reading/Saving From A CSV

```python
# File names for storing user data and removed user logs
USER_DATA_FILE = 'user_data.csv'
REMOVED_USERS_FILE = 'removed_users_log.csv'

# Load user data from the CSV file
def userdata():
    if not os.path.exists(USER_DATA_FILE):
        return [], [], []  # Return empty lists if the file does not exist

    usernames, passwords, roles = [], [], []
    with open(USER_DATA_FILE, mode='r', newline='') as file:
        reader = csv.reader(file)
        for row in reader:
            if row:  # Ensure the row is not empty
                usernames.append(row[0])
                passwords.append(row[1])
                roles.append(row[2])
    return usernames, passwords, roles

# Save user data to the CSV file
def savedata():
    with open(USER_DATA_FILE, mode='w', newline='') as file:
        writer = csv.writer(file)
        for i in range(len(usernames)):
            writer.writerow([usernames[i], userpasswords[i], userroles[i]])

# Log removed user data
def logremoved(username):
    with open(REMOVED_USERS_FILE, mode='a', newline='') as file:
        writer = csv.writer(file)
        # Write the username and the timestamp of removal
        writer.writerow([username, datetime.now().strftime("%Y-%m-%d %H:%M:%S")])

# Load user data
usernames,userpasswords,userroles=userdata()

```

Improvements to User Search

The interface now includes a refresh button, and search results are displayed vertically rather than horizontally for improved readability. A pop-up notification has been introduced to provide immediate feedback when items are added to favorites. Finally, the GUI has been made more appealing by making the buttons horizontal instead of pancaked vertically, and a dynamic search bar has been implemented to update results in real-time as the user types.

Figure 53, New dynamic search using treeview

```
def performSearch(event=None):
    search_type = searchOption.get()
    search_query = searchEntry.get().strip()

    #This is because a issue happened that when the search was blank no items would show up
    if not search_query:
        loadAllUsers()
        return

    search_results = searchItems(search_type, search_query)

    for row in tree.get_children():
        tree.delete(row)
    for item in search_results:
        tree.insert("", tk.END, values=(item["ID"], item["Name"], item["Producer"]))
```

Figure 54, New vertical treeview

```
# Search results treeview
treeFrame = tk.Frame(root)
treeFrame.pack(pady=10, fill=tk.BOTH, expand=False)

tree = ttk.Treeview(treeFrame, columns=("ID", "Name", "Producer"), show="headings", height=10)
tree.heading("ID", text="ID")
tree.heading("Name", text="Name")
tree.heading("Producer", text="Producer")
tree.pack(side="left", fill=tk.BOTH, padx=10, pady=10)

scrollbar = ttk.Scrollbar(treeFrame, orient="vertical", command=tree.yview)
tree.configure(yscroll=scrollbar.set)
scrollbar.pack(side="right", fill="y")
```

Figure 55, Improved GUI

```
# Main application window you can rezie it if you want but this is pretty good size
root = tk.Tk()
root.title("Regular User Search")
root.geometry("600x500")

# Search frame
searchFrame = tk.Frame(root)
searchFrame.pack(pady=10)

# Search Options/ buttons for diffrent types of data
tk.Label(searchFrame, text="Search By:").grid(row=0, column=0, padx=5)

searchOption = tk.StringVar(value="ID")
tk.Radiobutton(searchFrame, text="ID", variable=searchOption, value="ID").grid(row=0, column=1, padx=5)
tk.Radiobutton(searchFrame, text="Name", variable=searchOption, value="Name").grid(row=0, column=2, padx=5)
tk.Radiobutton(searchFrame, text="Producer", variable=searchOption, value="Producer").grid(row=0, column=3, padx=5)

# Search entry
tk.Label(searchFrame, text="Enter Search Query:").grid(row=1, column=0, padx=5)
searchEntry = tk.Entry(searchFrame)
searchEntry.grid(row=1, column=1, columnspan=3, padx=5, pady=5)
searchEntry.bind("<KeyRelease>", performSearch)

# Here are all the cool buttons
buttonFrame = tk.Frame(root)
buttonFrame.pack(pady=10)

# Removed the search button
tk.Button(buttonFrame, text="Add to Favorites", command=addToFavorites).grid(row=0, column=0, padx=5)
tk.Button(buttonFrame, text="Clear Search", command=clearSearch).grid(row=0, column=1, padx=5)
tk.Button(buttonFrame, text="Refresh", command=refreshSearch).grid(row=0, column=2, padx=5)
tk.Button(buttonFrame, text="Main Menu", command=root.destroy).grid(row=0, column=3, padx=5)
```

Improvements to Admin view Requests

The application now features live refresh in the search box, eliminating the need for a dedicated search bar and improving usability. Additionally, the CSV file dynamically updates whenever a request is rejected or accepted, allowing real-time data. To help error handling and organization, more error checking has been implemented, and a dedicated CSV file has been added to track all rejected requests for better management and record-keeping.

Figure 56, Live refresh

```
def refresh(tree, query=""):
    for row in tree.get_children():
        tree.delete(row)

    items = openFile()
    if query:
        items = [req for req in items if query.lower() in req["Name"].lower()]

    for req in items:
        tree.insert("", "end", values=(req["ID"], req["Name"], req["Producer"], req["Status"]))
```

Figure 57, Messages to let admin's know that they have successfully rejected or accepted something.

```
def buttonHandler(tree, action):
    selected = tree.selection()
    if not selected:
        messagebox.showwarning("Error!", "Please select an req to perform this action.")
        return

    reqID = tree.item(selected[0], "values")[0]
    if action == "accept":
        newStatus = "Approved"
        if updateStatus(reqID, newStatus):
            messagebox.showinfo("Success", f"Request {reqID} accepted successfully.")
    else:  # this is essentially action == "reject"
        updateStatus(reqID, "Rejected")
        deleteRequest(reqID)
        messagebox.showinfo("Success", f"Request {reqID} rejected and deleted successfully.")

    refresh(tree)
```

Figure 58, Dynamic searching

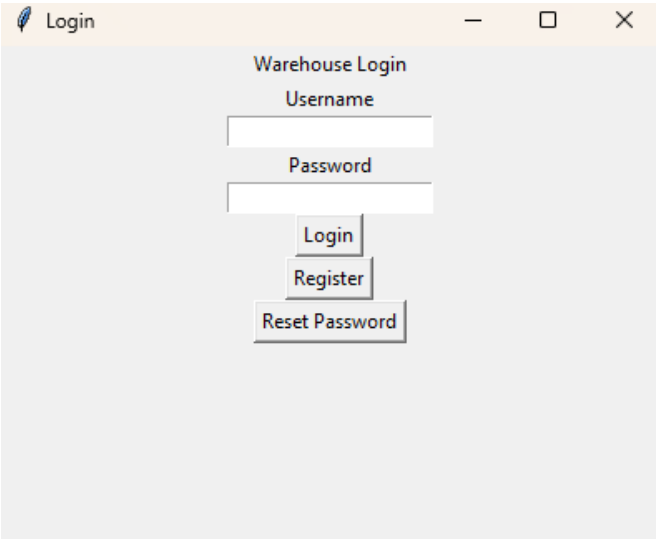
```
# dynamically search the treeview when the user types in the search box. Here is a link for where I found how to do this https://www.youtube.com/watch?v=mSpLnnXeiIc
def searchRequests(event=None):
    field = searchArea.get()
    query = dynamicSearch.get().strip()
    items = openFile()
    results = [req for req in items if query.lower() in req[field].lower()]

    for row in tree.get_children():
        tree.delete(row)

    for req in results:
        tree.insert("", "end", values=(req["ID"], req["Name"], req["Producer"], req["Status"]))

dynamicSearch.bind("<KeyRelease>", searchRequests)
tk.Button(root, text="Back to Main Menu", command=mainMenu).pack(pady=10)
```

Figure 59, Add rejected items to CSV

```
def deleteRequest(reqID):
    items = openFile()
    rejected_items = openFile(rejectedFilePath)
    with open(filePath, mode='w', newline='') as file:
        write = csv.DictWriter(file, fieldnames=["ID", "Name", "Producer", "Status"])
        write.writeheader()

        for req in items:
            if req['ID'] == reqID and req['Status'] == "Rejected":
                rejected_items.append(req)
            else:
                write.writerow(req)

    with open(rejectedFilePath, mode='w', newline='') as file:
        write = csv.DictWriter(file, fieldnames=["ID", "Name", "Producer", "Status"])
        write.writeheader()
        write.writerows(rejected_items)
```

Visually reported stored data for login page:
This code creates a login functionality inside a Tkinter GUI application. Upon clicking the Login button, the login function looks up the entered username and password from the stored user list. It gives an error message for users who are blocked due to several previous failed attempts. For the user not found, it logs the attempt, and after multiple such failed attempts, blocks that user. Successful logins display either the admin or user menu depending on the username. The interface will also include input fields for username and password, and buttons for login, registration, and password reset, which link to their respective functions.

Figure 60, Login Page GUI



Visually reported stored data for Main Menu:
The main menu is a dynamic Tkinter window that changes depending on the user's type: admin or regular user. It starts by clearing any existing widgets in the window, then it shows a label identifying what menu it is and adds buttons depending on the user's role. For an admin, it would include buttons to manage stock, add users, and so on, while for a regular user, it would include options like 'View History' or 'Search Items'. Each button has an action that is executed once clicked.

Figure 61, Admin Main Menu GUI

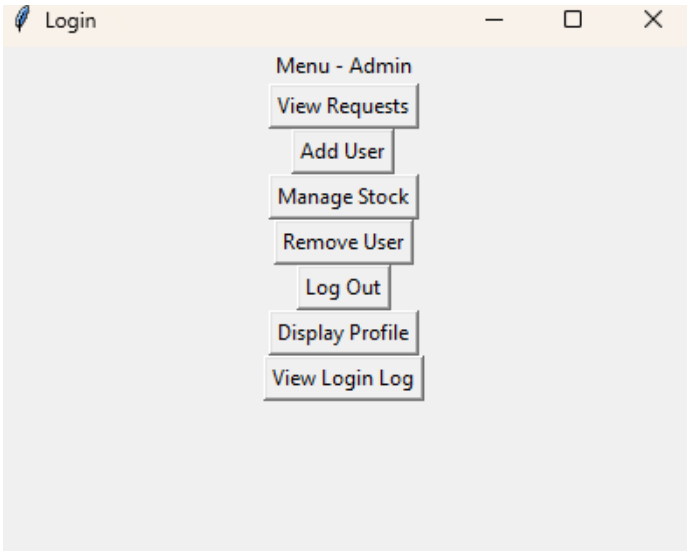
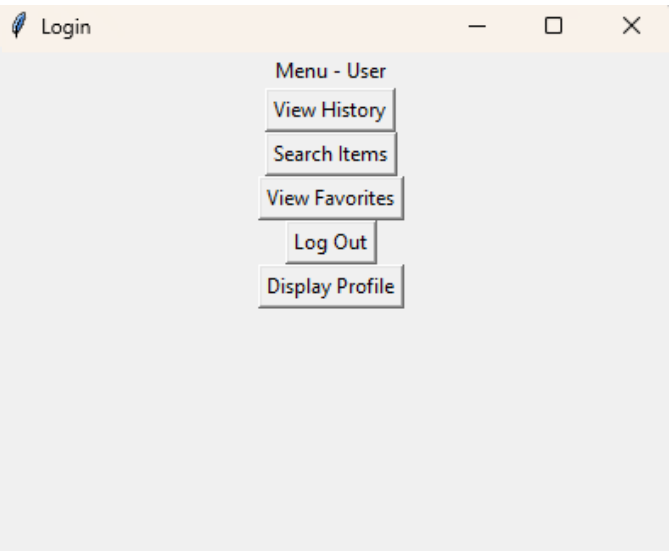


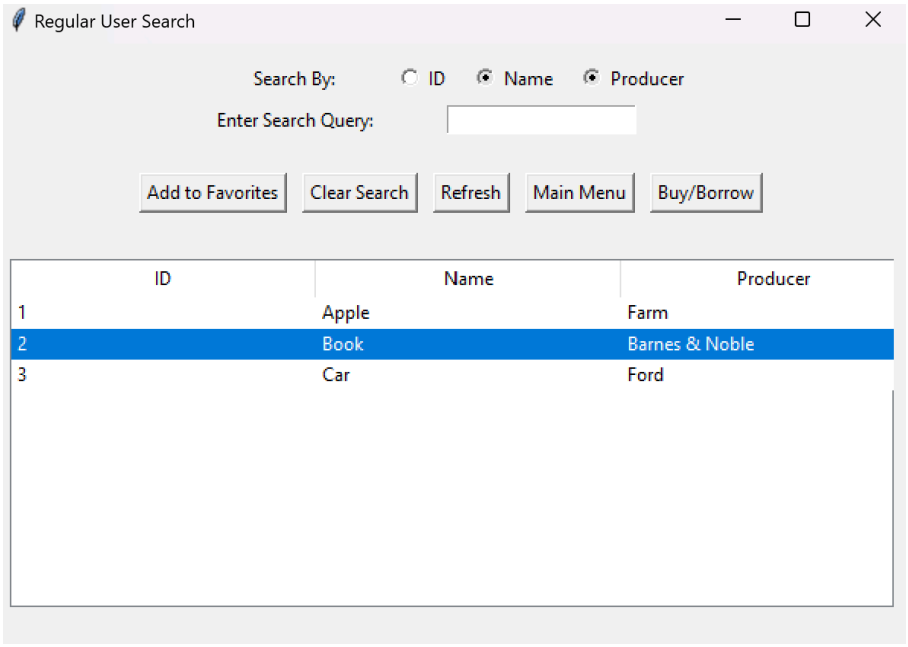
Figure 62, User Main Menu GUI



User Search

The user search program allows users to search items from a CSV file by ID, Name, or Producer, and displays results in a treeview widget. Users can type queries into a search bar which dynamically updates the displayed items based on the users input. The search results are displayed vertically, and users can add items to their favorites list. The program creates a GUI using Tkinter, defining functions for loading and searching items in the CSV, and using a treeview widget to display the results.

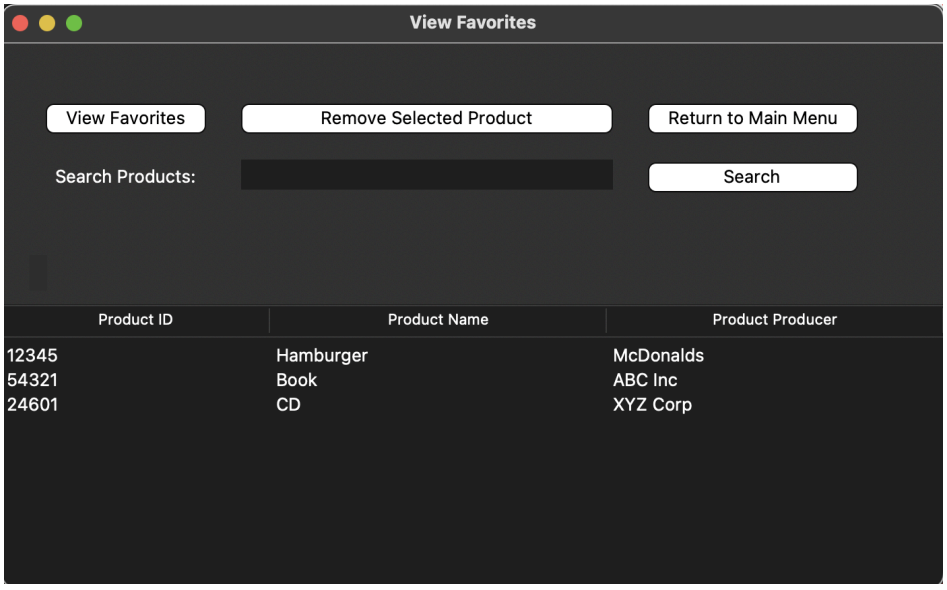
Figure 63, User Search GUI



View Favorites

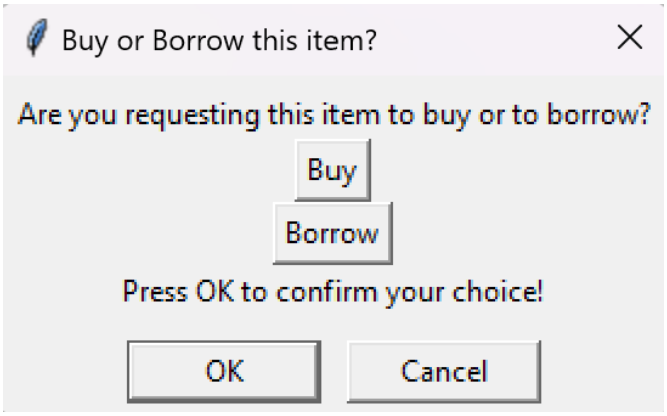
The view favorites function allows the user to view all the items that they favorited in the user search function. The view favorites button refreshes the treeview and the favorites.csv file that the data is stored in. The remove selected product allows the user to select a product and remove it from the treeview and csv. Clicking on the headers of the list allows the user to sort the items numerically or alphabetically. The return to the main menu destroys the view favorites function and returns the user to the main menu page.

Figure 64, View Favorites GUI



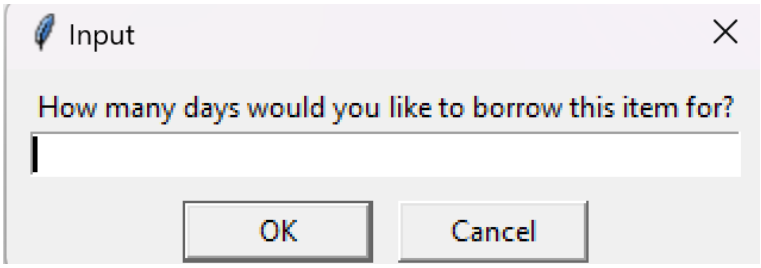
The buy/borrow popup feature allows users to request items for either purchase or to borrow. When a user selects an item, a custom dialog prompts them to choose between buying or borrowing the item.

Figure 65, Buy or Borrow window



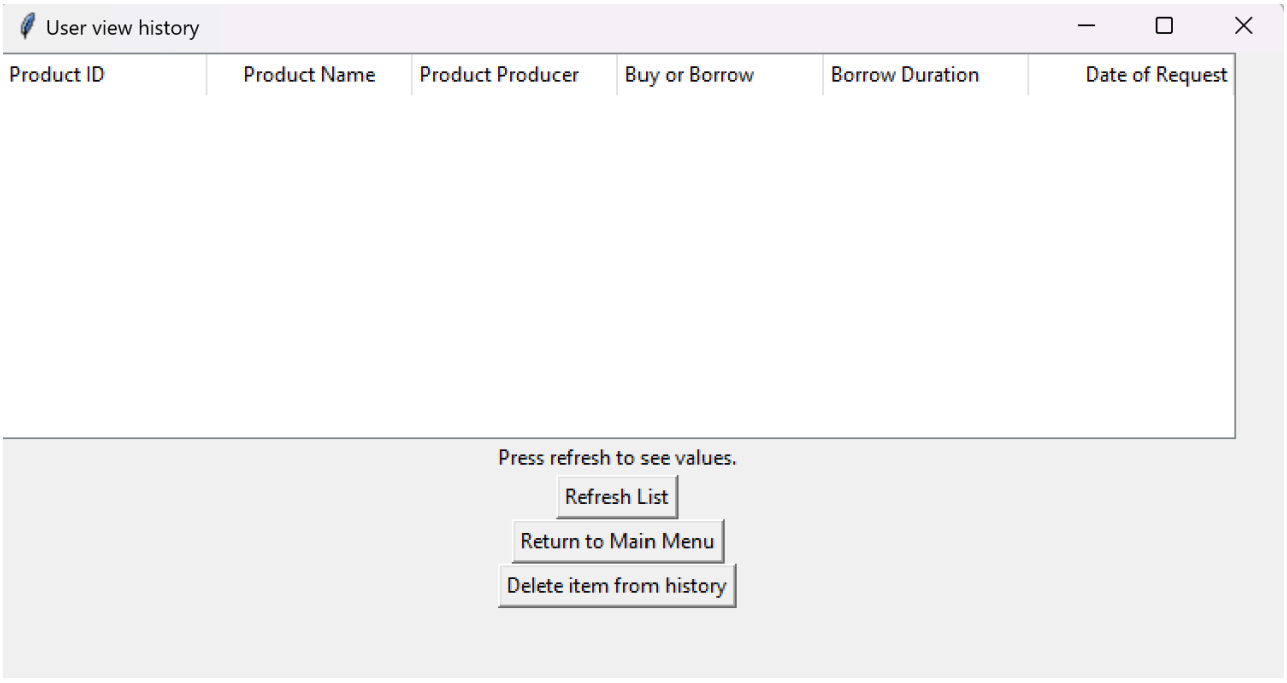
If borrowing is selected, the user specifies the borrow duration. The requests are then recorded in a BorrowList.csv and UserHistory.csv file, with the request status marked as PENDING until approved by an admin.

Figure 66, borrow input window



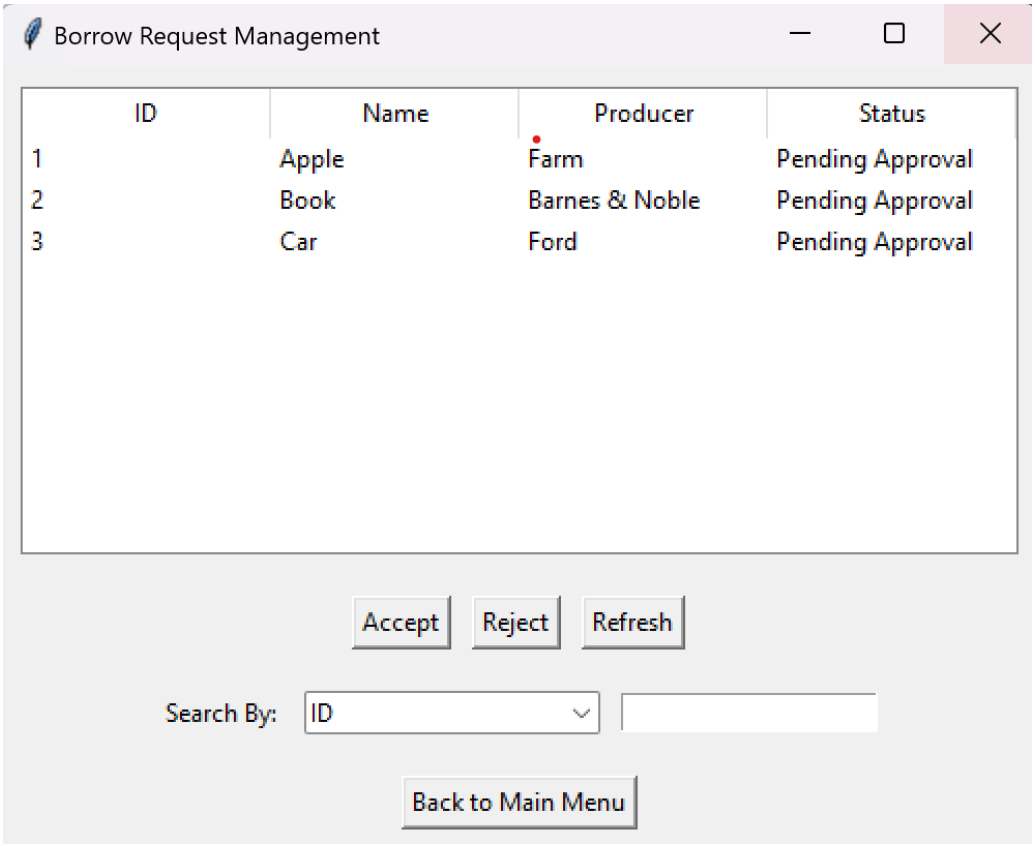
The View History window uses a tkinter tree display to show the user all the items they have requested. It displays the product ID, Name, and Producer. It also displays if the user wanted to buy or borrow the item, the duration of the item, and the date they requested it. The refresh list button re-reads the UserHistory.csv file, and the delete item removes the selected item from the file. Returning to the main menu can be done by clicking the return to main menu button.

Figure 67, ViewHistory Window



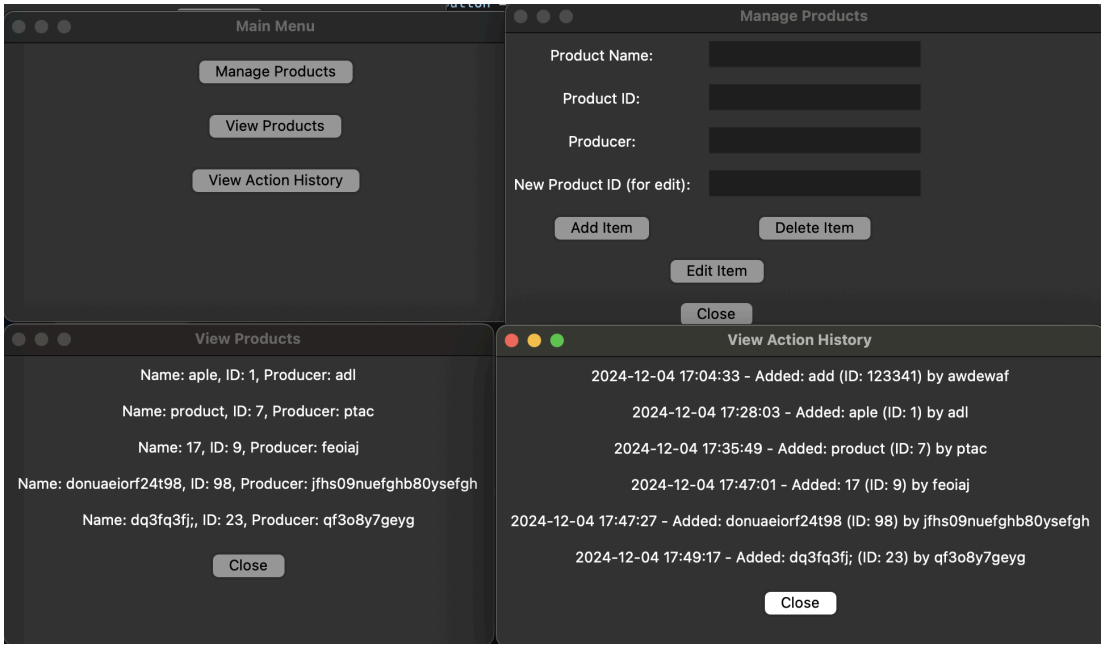
Borrow requests use Tkinter to manage borrow requests. It provides a GUI for administrators to view, approve, or reject borrow requests stored in a CSV file. The application features dynamic searching, live refresh of the displayed data, and error handling for missing files to inform the admin. It reads requests from a primary CSV file and moves rejected requests to a separate CSV file while deleting, updating the status of requests as needed. The implementation uses Tkinter for the GUI with treeviews for displaying data and buttons for performing actions, while using CSV for all stored data.

Figure 68, Borrow Request GUI



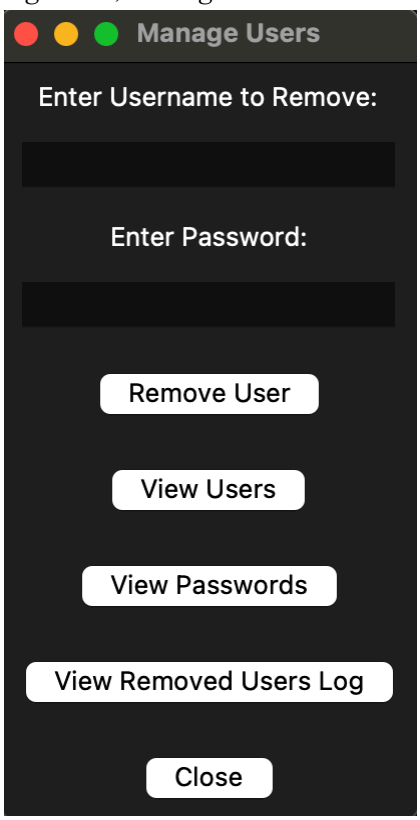
The Manage Products window allows a user to add, delete, and edit products by inputting a product name, product id, producer, and new product id to edit a product. Then, you can press the view products button and a messagebox will appear showing all the products currently in the warehouse. Also, by pressing view action history, you can see all the changes to the products in the warehouse and when those changes happened.

Figure 69, Manage Products Windows.



The manage users menu allows you to remove a user by entering a username and a password then pressing the “Remove User” button. Also, you can press the “View Users” and “View Passwords” buttons to see each respective user's name and passwords. Also, you can press “View Removed Users Log” to see the past removed users and when they were removed.

Figure 70, Manage Users GUI



Admin Create Guest User

The Admin Create Guest User Function intends to allow the admin to create a user who doesn't have the same permissions as themself. It begins by checking if the provided username and password are valid. It then verifies that the username and password are not already in use. Assuming it passes these checks, the function adds the information to the system's database and writes the data to a CSV for permanent storage. Lastly, it pops up with a text box to display if the user was successfully added or if it encountered an error. This ensures the ability to add guests who do not have the same permissions as the admin account.

Figure 71, Guest User Create GUI